

Multi-Modal Transportation Simulation Framework: A Routing Game Approach

Dan Calderone¹, Agrima Khanna², Abhishek Dubey², and Lillian Ratliff¹

June 8, 2024

Contents

1 INPUTS	2
2 RUN INSTRUCTIONS	3
3 EXAMPLES	6
4 SIMULATION OVERVIEW	8
4.1 Simulation Object	8
4.2 Initialization Steps	8
4.3 Fitting Ondemand Latency Curves	10
4.4 Main Simulation Loop	10
5 ARCHITECTURE DETAILS	10
5.1 WORLD Object	12
5.2 GRAPH & FEED Objects	14
5.3 NETWORK Objects	15
5.3.1 SEG Objects	16
5.4 PERSON Object	17
5.4.1 TRIP Object	18
5.5 ONDEMAND object	19
5.5.1 GROUP object	20
5.5.2 RUN object	23
5.6 CONVERTER object	23
5.7 DASHBOARD	24
5.7.1 TRACE	26
5.7.2 SLIDER	27
5.7.3 BUTTON	27
5.8 RAPTOR object	28
6 METHODOLOGY	30
6.1 Travel Time Costs	30
6.1.1 Routing Game Formulation - Primal & Dual	30
6.1.2 Dual Algorithm: Driving Edge Cost Updates	31
6.1.3 Determining Driving Latency Functions	31
6.2 Ondemand Cost Estimation	31
6.2.1 Dual Algorithm: Ondemand Group Cost Updates	32
6.2.2 Determining Ondemand Latency Functions	32

6.3	Trip Cost Computation	32
6.4	Trip/Route Planning	33
6.4.1	Dynamic programming between trip segments	33
6.4.2	Shortest Path via Dijkstra/A* search - modes: drive, walk	33
6.4.3	RAPTOR implementation - modes: gtfs	33
6.4.4	VRP solver - modes: ondemand	34
7	OUTPUT FORMATS	34
7.1	Simulation Data	34
7.1.1	By mode	34
7.1.2	Ondemand data	34
7.2	Dashboard	35
8	GENERATING INPUTS	36
8.1	Generating Group Bounding Regions	36
8.2	Generating Population Data	37
8.3	Generating Group & Driver Run Data	37

1 INPUTS

The following files are required to run the simulation.

INPUTS/pop_data.csv
 INPUTS/regions.geojson
 INPUTS/regions_pickup.geojson
 INPUTS/regions_dropoff.geojson
 INPUTS/carta_gtfs.zip
 INPUTS/gtfs_trips.obj

The following files are optional.

INPUTS/background_traffic.obj

Details for each are given below.

- INPUTS/pop_data.csv
 (or similar file) Contains population data for constructing agents in simulation. (Generated by Agrima's code).
- INPUTS/regions.geojson
 (or similar file) Contains information about group ondemand regions. Details for generating .geojson files given in Sec. 8.1.
- INPUTS/regions_pickup.geojson
 (or similar file) Contains information about group pickup regions. Details for generating .geojson files given in Sec. 8.1.
- INPUTS/regions_dropoff.geojson
 (or similar file) Contains information about group dropoff regions. Details for generating .geojson files given in Sec. 8.1.
- INPUTS/carta_gtfs.zip
 Contains stored gtfs feed information for CARTA.

- INPUTS/gtfs_trips.obj
Contains precomputed optimal trips for fixed line CARTA transport generated using the RAPTOR object. Generation detailed in Sec. 5.8.
- INPUTS/background_traffic.obj (OPTIONAL)
(or similar file) Contains background traffic information.

2 RUN INSTRUCTIONS

The following is details for running the simulator in the Jupyter notebook MULTIMODAL_SIM.ipynb. The code depends on the functions and structures defined in `multimodal_class.py`.

• PRELIMINARIES

The Instructions for connecting with the VRP solver module.

STEP 1: run each of these in a separate terminal

```
TERMINAL 1:  
export SERVER='carta-cav'  
ssh -L 8081:localhost:8081 "$SERVER"  
  
TERMINAL 2:  
export SERVER='carta-cav'  
ssh -L 26025:localhost:26025 "$SERVER"
```

STEP 2: update the following two lines at the top of

```
multimodal_class.py  
to be the appropriate paths on your system  
path_to_VRPsolver = '/Users/dan/Documents/transit_webapp/'  
path_to_multimodalsim = '/Users/dan/Documents/multimodal/'
```

• Imports

The following lines import the appropriate functions and structures.

```
[1]: %load_ext autoreload  
%autoreload 2  
from multimodal_class import *
```

• Constructing File Names

The following constructs the appropriate file names.

```

• [4]: max_num_people = 300; regions_name = 'C';
ondemand_cost = 0; num_iters = 6;
num_drivers_per_group = {'group0':int(8),'group1':int(8)}; drivers_per_group = '8n8';

name = 'regions' + regions_name;
name = name + '_drivers' + drivers_per_group;
name = name + '_cost' + str(ondemand_cost);
name = name + '_people' + str(max_num_people);
name = name + '_iters' + str(num_iters);

geojson_file = 'INPUTS/newregions8.geojson'; # + str(regions_num) + '.geojson';
save_fits_file = 'INPUTS/' + name + '.pickle';
load_fits_file = save_fits_file;
save_outputs_folder = 'OUTPUTS/' + name
dashboard_folder = 'DASH/' + name;

csv_file_path = 'INPUTS/new_pop_data8.csv'

print('geojson_file: ', geojson_file)
print('load_fits_file: ', load_fits_file)
print('save_fits_file: ', save_fits_file)
print('save_outputs_folder: ', save_outputs_folder)
print('dashboard_folder: ', dashboard_folder)

```

• Initial Parameters Inputs

```

[6]: params = {'csv_file_path': csv_file_path, #'INPUTS/PRE_micro_0.csv',
'modes':['drive','walk','gtfs','ondemand'],
'time_window': [21600,36000], # in seconds. 6 AM to 10 AM
'bnds':[np.array([-85.3394, 34.9458]), # bottom left
        np.array([-85.2494, 35.0658])], # top right
'groups_regions_geojson': geojson_file, #'INPUTS/regions3.geojson',
'max_num_people':max_num_people,
# 'groups_regions_pickup_geojson': geojson_pickup_file, #'INPUTS/regions3.geojson',
# 'groups_regions_dropoff_geojson': geojson_dropoff_file, #'INPUTS/regions3.geojson',
'num_drivers_per_group': num_drivers_per_group, #{'group0':int(10),'group1':int(4),'group2':int(4)},
'region_assignment': region_assignment,
'monetary_costs':{'ondemand':0,'gtfs':1.5},
# 'background_congestion_file': 'INPUTS/background_traffic1.obj', # OPTIONAL - LOADS BACKGROUND TRAFFIC
##### PRECOMPUTED GTFS DATA #####
'load_ondemand_fits': load_fits_file, #'INPUTS/load_file.pickle',
'save_ondemand_fits': save_fits_file, #'INPUTS/save_file.pickle',
'gtfs_feed_file': 'INPUTS/carta_gtfs.zip',
'gtfs_precomputed_file': 'INPUTS/gtfs_trips.obj'}

```

• Generate Population

```

[7]: if not os.path.exists(save_outputs_folder):
    os.mkdir(save_outputs_folder);
    os.mkdir(save_outputs_folder+'/figs');

[8]: if False: #not os.path.exists(csv_file_path):
    df = GENERATE_SAMPLE_POPULATION('data/pop/lodes_combinations_upd.parquet',filter_region_path=geojson_file,sample_num = 10000)
    df.to_csv(csv_file_path); #, index_label='tag')'INPUTS/new_pop_data1.csv'

```

• Initialize Simulation

```
[9]: # OPTION-1: FULL ONE-SHOT SETUP
# BLAH = WORLD(params, full_setup = True)
# OPTION-2: STEP-BY-STEP SETUP
WORLD1 = WORLD(params)
WORLD1.initGRAPHnFEEDS();
WORLD1.initNETWORKS();
WORLD1.initCONVERTER();
WORLD1.initSTATS()
WORLD1.initONDEMAND();
WORLD1.initPEOPLE();
WORLD1.initBACKGROUND();
```

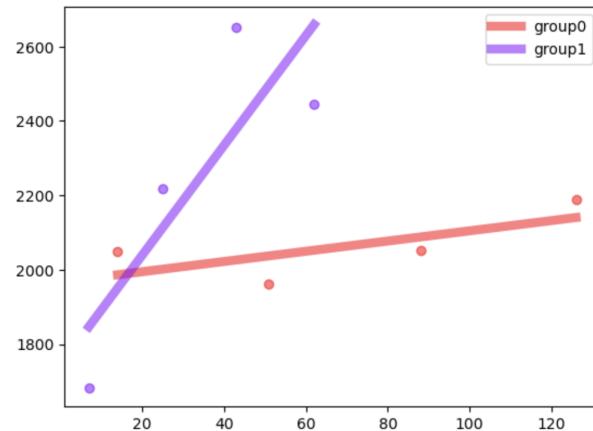
- Plot ODs & Group Regions

```
[11]: from multimodal_class import *
WORLD1.plotPRELIMINARIES(include_demand_curves=False, save_file=save_outputs_folder+'/figs/od_regions.png', figsize=(5,5));
OUTPUTS/regionsC_drivers8n8_cost0_people300_iters6/figs/od_regions.png
```

- Fit Congestion Demand Models

```
[12]: from multimodal_class import *
WORLD1.load_fits = False;
# counts = {'num_counts':8,'num_per_count':1}
counts = {'num_counts':4,'num_per_count':1}
WORLD1.fitModels(counts=counts,force_initialize=True);
```

```
[13]: colors = {'group0':[1,0,0,0.5], 'group1':[0.5,0,1,0.5], 'group2':[0,0,1,0.5], 'group3':[1,0.5,0.,0.5]};
WORLD1.ONDEMAND.plotCongestionModels(colors=colors, save_file=save_outputs_folder+'/figs/demand_curves.pdf')
```



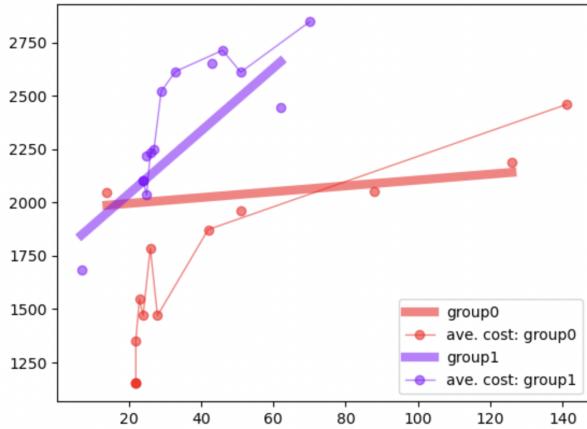
```
[14]: # WORLD1.save_fits = True; #se; # must be set to True
# WORLD1.save_fits_file = save_fits_file;
# WORLD1.saveFits();
```

- Run Simulation

```
[16]: from multimodal_class import *
num_iters = 10;
WORLD1.SIM(num_iters = num_iters, restart = False, constants = ['drive','gtfs','walk']);
WORLD1.initUNCONGESTED()
```

- Plot Ondemand Convergence

```
[18]: colors = {'group0': [1,0,0,0.5], 'group1': [0.5,0,1,0.5], 'group2': [0,0,1,0.5], 'group3': [1,0.5,0.,0.5]};
WORLD1.ONDEMAND.plotCongestionModels(colors=colors,with_converge = True, save_file=save_outputs_folder+'/figs/demand_curves_wconvergence.pdf')
```



- Generate Outputs

```
[19]: from multimodal_class import *
WORLD1.generateOutputs(); #randomize=['money','switches'];
mode = 'ondemand'
dfs_to_show = [mode]
WORLD1.printOutputs(show_active=True,dfs_to_show = dfs_to_show, row_count = 100);
WORLD1.saveOutputs(save_outputs_folder,typ='pickle',overwrite=True)
```

- Generate Dashboard

```
[22]: from multimodal_class import *
WORLD1.generateLayers();
WORLD1.plotLAYERS(folder = dashboard_folder,overwrite = True,verbose=False)

WORLD1.DASH = DASHBOARD(WORLD1.dash_folder);
WORLD1.DASH.makeGrid();
WORLD1.DASH.addOutputs(WORLD1.OUTPUTS,use_active=True,filter_for_zeros=True)
WORLD1.DASH.addImages(WORLD1.layers)

WORLD1.DASH.show()
import plotly.io as pio
pio.write_html(WORLD1.DASH.fig, file=dashboard_folder+'/dash.html', auto_open=True)
```

3 EXAMPLES

Dashboard examples

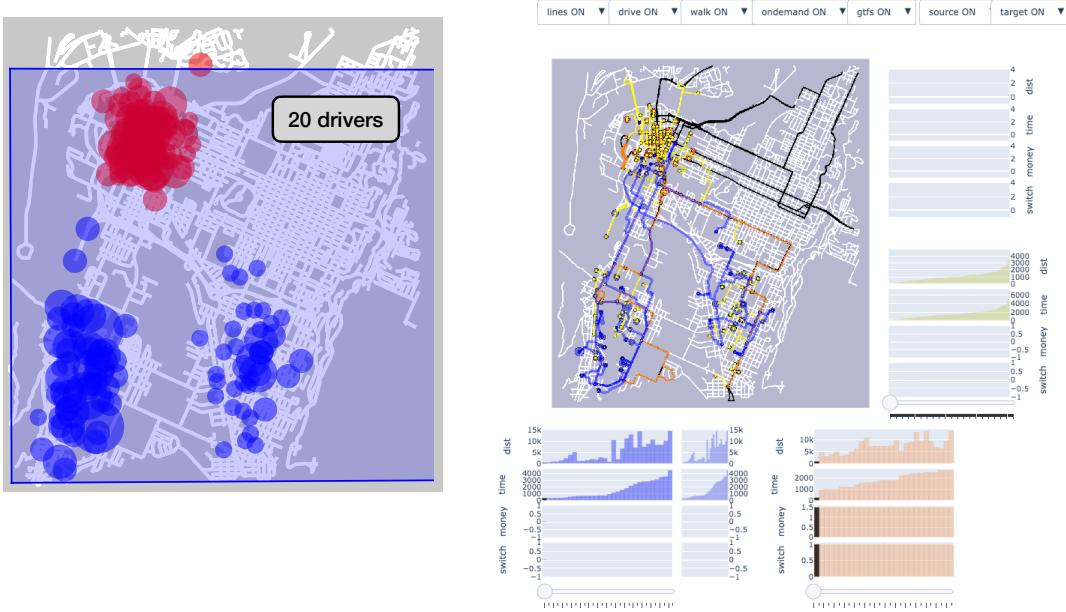


Figure 1: Dashboard example for example with one region with 20 ondemand drivers.

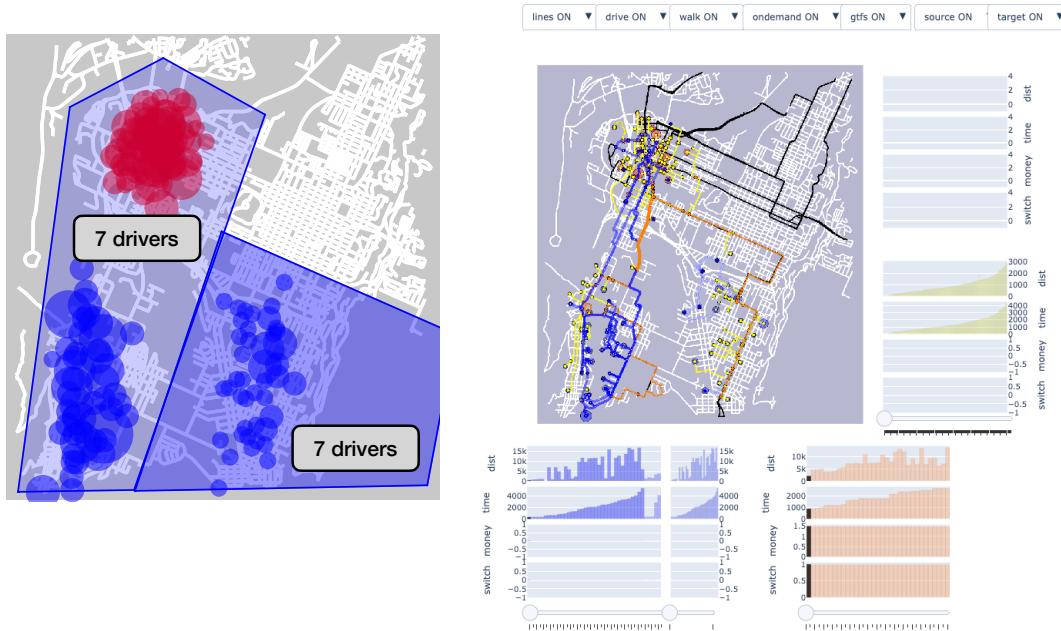


Figure 2: Dashboard examples for example with two regions with 7 drivers in each region.

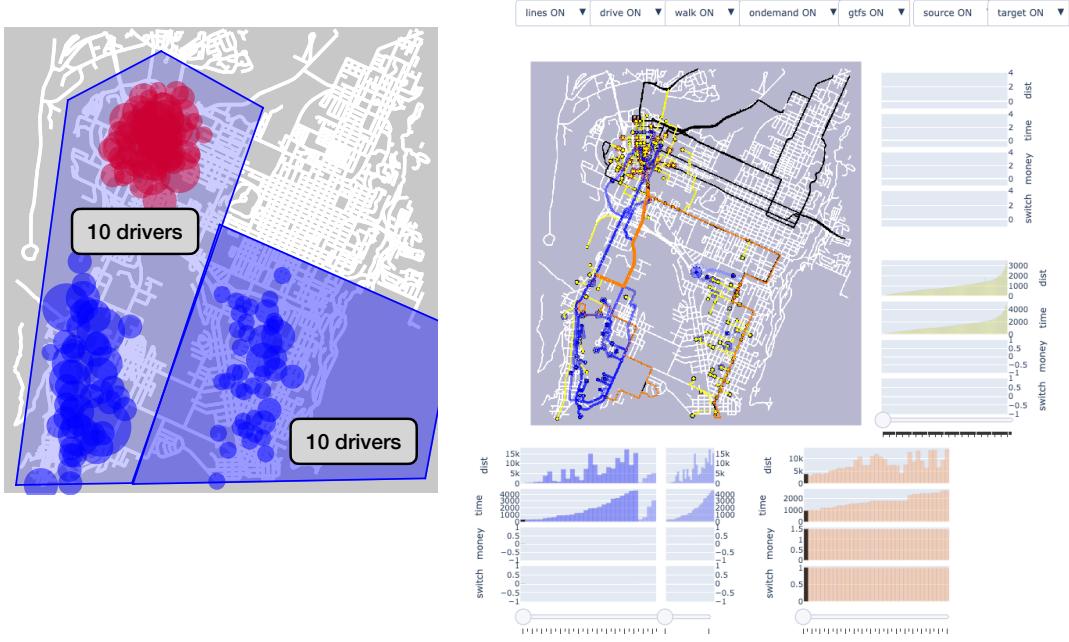


Figure 3: Dashboard example for example with two regions with 10 drivers in each region.

4 SIMULATION OVERVIEW

In this section, we detail the software architecture for the simulator. In the following subsections, we detail the various objects their attributes and functions. A brief overview of each object and its position in the architecture is given here. An overview of the steps in running each simulation iteration are given in Section ??.

4.1 Simulation Object

Figure 8 gives an overview of the main simulation object. More details are given in Section ???. The **NETWORKS** objects contain details of each mode network including trip segments in use by agents and traffic on each mode (details in Sec. 5.3). The **GRAPHS** and **FEED** objects contains OSMNX graphs and gtfs feed information. The **PERSON** objects in the **PEOPLE** dict contain information for each agent in the simulation (details in Sec. 5.4). The **ONDEMAND** object and **GROUP** objects inside contain information about the ondemand microtransit mode (details in Sec. 5.5 and 5.5.1). The **CONVERTER** converts between nodes in different mode networks (details in Sec. 5.6).

4.2 Initialization Steps

An overview of the initialization steps for the simulation are given in Fig. 5. The **GRAPH** and **FEED** are initialized from OSMNX graph structures and gtfs feed (loaded in a zip file). Population data can be generated artificially or generated from census data (details in Sec. 8.2). The ondemand data information is loaded from a **.geojson** file generated from **geojson.io** (details in Sec. 8.1) and from dataframes that contain group and driver run information (details in Sec. 8.3).

The **NETWORK** and **CONVERTER** objects are generated in the initialization. The individual agents (**PERSON** objects) are generated from the population data, initial potential routes are generated, appropriate nodes are added to the node **CONVERTER**, and agents are assigned to the appropriate ondemand group.

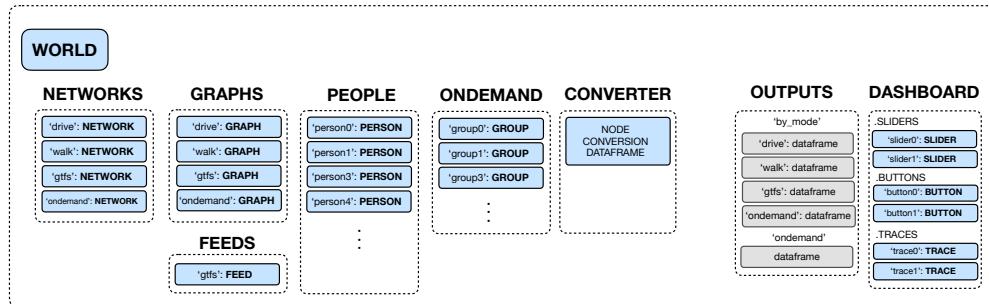


Figure 4: Overview of simulation architecture

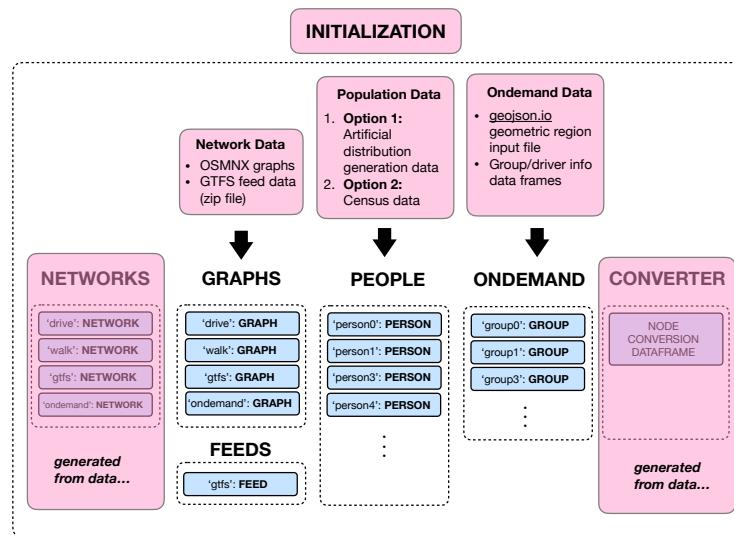


Figure 5: Overview of simulation architecture

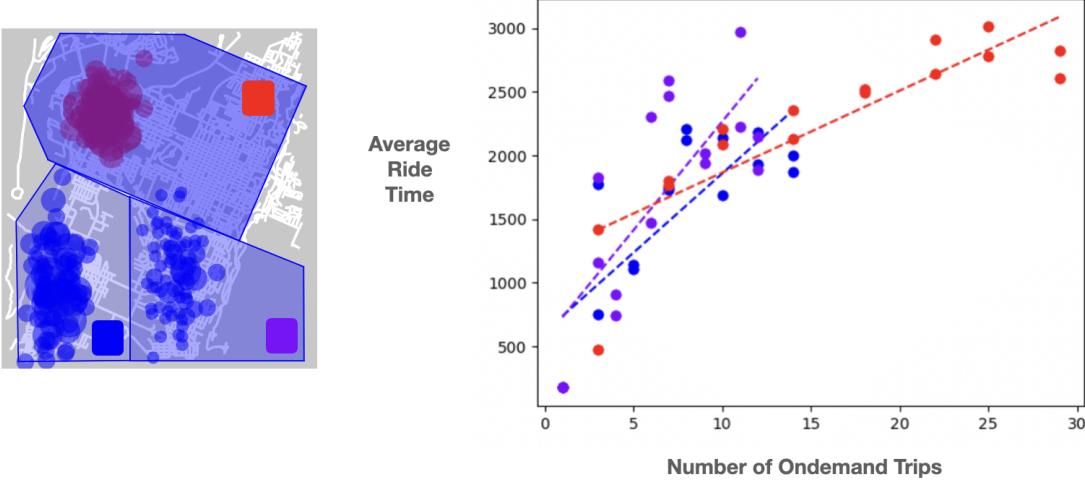


Figure 6: Curve fitting examples

4.3 Fitting Ondemand Latency Curves

After the populations are generated and agents are assigned to the appropriate ondemand group, demand curves must be fit for each ondemand group in order to estimate the groups travel time based on the number of users requesting rides during the simulation process. This is done by the function

```
WORLD.fitModels(counts ={'num_counts':10,'num_per_count':4})
```

which in turn calls the functions

```
ONDEMAND.generateCongestionModels
GROUP.generateCongestionModel
```

In order to fit the demand curve for each group, the VRP solver is called for different numbers of possible agents within the group region at random. The number of agents to plan trips for is chosen at regular intervals over the total possible number of trips. The number of intervals is determined by the input `counts['num_counts']`. The number of random samples to take at each specific number of trips is determined by the input `counts['num_per_count']`. The demand curves are fit using a simple linear model. (This could fairly easily be updated to use a higher order curve.) We give an example of the fitting curves for three ondemand regions in Fig. 6. In this example, the inputs are `counts={‘num_counts’:8,’num_per_count’:2}`.

4.4 Main Simulation Loop

The main simulation loop is detailed in Fig. 7. In each iteration loop, the first step is to update the optimal routes and costs for all the trip segments in each `NETWORK` object. For the ondemand network the `ONDEMAND` object is called to update the optimal VRP solutions for each driving group. This step requires a call to the VRP routing package (details in Sec ??).

5 ARCHITECTURE DETAILS

This section contains details of each of the objects in the software architecture. We start with a summary of the objects listed below.

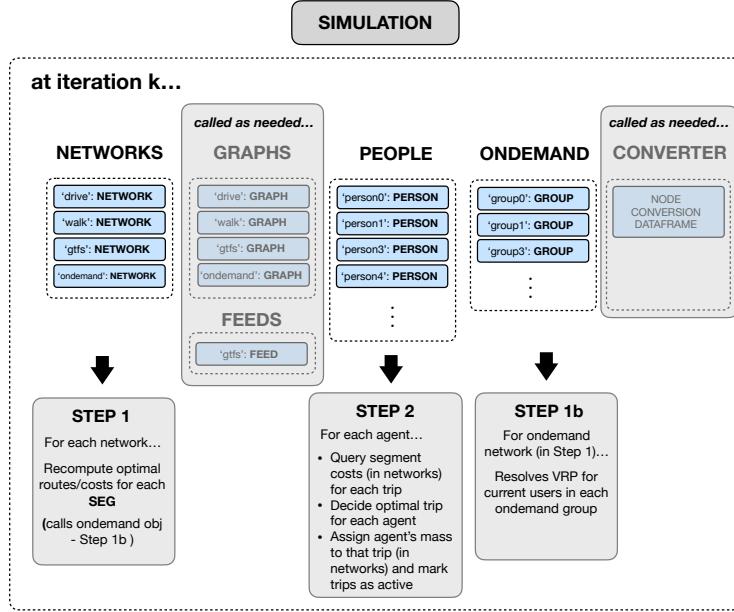


Figure 7: Overview of simulation architecture

SUMMARY

- **WORLD**: Overall simulator object.
- **GRAPH**: NetworkX graph objects that detail the road network for specific transportation modes.
- **FEED**: GTFS feed information for fixed line mode.
- **NETWORK**: Contains information about traffic flow and individual trip segments for a specific transportation mode. Recomputes optimal routes for each segment at each iteration.
 - **SEG**: Contains information about optimal routes between origin and destination nodes within a particular transportation mode.
- **PERSON**: Preference and choice information for a specific agent in the population including preferences for each cost component, origin and destination nodes, possible trips to choose from, and current optimal trip choice and optimal routes.
 - **TRIP**: Contains information about segment combinations that form a single trip origin to destination. May include segments from multiple transportation networks.
- **ONDEMAND**: Contains information specific to the ondemand transportation mode.
 - **GROUP**: Contains information for ondemand transportation group including number of vehicles and other parameters, current driver run assignments, etc.
 - **RUN**: Contains information about specific driver runs for vehicles within an ondemand group.
- **CONVERTER**: Dataframe that converts node tags between different transportation modes.
- **DASHBOARD**: Object that generates and controls the visualization dashboard from data
 - **TRACE** Object that controls a trace on a plot in the dashboard.

Attributes		
Field	Value	Description
NETWORKS	dict	dict of NETWORK objects
GRAPHS	dict	dict of OSMNX GRAPH objects
RGRAPHS	dict	dict of OSMNX (reversed) GRAPH objects
FEEDS	dict	dict of stored gtfs feed info
ONDemand	obj	ONDemand object
PEOPLE	dict	dict of PERSON objects
gpsDF		dataframe with ondemand group info
driversDF		dataframe with ondemand driver info
DELIVERYDF		
OUT		(preloaded population information)
PRE		(preloaded population information)
NODES		(preloaded population information)
LOCs		(preloaded population information)
SIZES		(preloaded population information)
VEHS		(preloaded population information)
PRELOADS		(preloaded population information)

Functions			
Function	Description	Function	Description
<code>_init_</code>	Initializes simulation. (If <code>full_setup = True</code> , also calls the other * functions)	<code>fitModels</code>	Fit ondemand congestion curves (calls <code>ONDemand</code> function)
<code>initGRAPHSnFEEDS*</code>	Initializes GRAPH and FEED objects from OSMNX data and gtfs feed file	<code>saveFits</code>	(Optional) saves ondemand curve information
<code>initNETWORKS*</code>	Initializes NETWORK objects	<code>loadFits</code>	(Optional) loads precomputed ondemand curve information
<code>initCONVERTER*</code>	Initializes CONVERTER object	<code>graphFromFeed</code>	construct NetworkX graph from gtfs feed
<code>initSTATS*</code>	Loads population states (Interface with Agrima's code)	<code>trimGraph</code>	cut down graph based on geographic region
<code>initONDemand*</code>	Initializes ONDemand object	<code>SIM</code>	run main simulation loop
<code>initPEOPLE*</code>	Initializes PERSON objects for each agent (stats for indiv agents loaded in <code>initSTATS</code>)	<code>makeDashboard</code>	construct dashboard
<code>initBACKGROUND*</code>	(Optional) Loads any background traffic info in <code>self.preloads_file</code>	<code>generateLayers</code>	generate layers for dashboard plots
<code>initUNCONGESTED</code>	Computes costs for trips without congestion (for comparison purposes)	<code>plotLayer</code>	plot a layer and save image for dashboard
<code>createGroupsDF</code>	Creates data frame with ondemand group information	<code>plotLayers</code>	plot all layers and save as images for dashboard plots
<code>createDriversDF</code>	Creates data frame with ondemand driver run information	<code>generateGraphPresets</code>	(generate plotting presets for dashboard)
<code>plotPRELIMINARIES</code>	Plot initial population information - geographic region, origins and destinations, etc.	<code>generateOutputs</code>	generate dataframes with output data
<code>add_base_edge_masse</code>	adds background traffic to edges if loaded initBackground	<code>printOutputs</code>	display outputs in notebook
		<code>saveOutputs</code>	save output data frames in pickle file
		<code>LOAD</code>	load precomputed simulation data (may not work)
		<code>SAVE</code>	save simulation data

Figure 8: Overview of simulation architecture

- **SLIDER** Object that controls a slider control on dashboard.
- **BUTTON** Object that controls a button control on dashboard.
- **RAPTOR**: Class that implements RAPTOR algorithm to solve for shortest path with transfers within a fixed-line transportation network given a GTFS feed. (currently run offline).

In the following sections, we detail each object.

5.1 WORLD Object

The simulator is run in a **WORLD** object whose overall structure is summarized in Fig. 8.

WORLD: function descriptions

- **WORLD.__init__:**
Initializes the overall **WORLD** simulation object. If the `full_setup = True`, then also calls the `WORLD.init*` functions given below. If `full_setup = False`, the other initialization functions must be called separately.
- **WORLD.initGRAPHSnFEEDS:**
Loads graphs (from OSMNX) and feed (from .zip file). Computes Networkx graph from gtfs feed and also computes reverse graphs (for later computation). Calls `WORLD.trimGraph` to cut graphs down to lie within inside geographic bounding region. Also computes a combined graph from all networks together stored in `WORLD.GRAPHS['all']`.
- **WORLD.initNETWORKS:**
Initializes **NETWORK** objects for each mode.

- **WORLD.initCONVERTER:**

Initializes CONVERTER object for node conversation between modes.

- **WORLD.initSTATS:**

(Should be updated to be more user friendly.) Catch all function that loads in precomputed population statistics information, generates some ondemand information using `WORLD.createGroupsDF` and `WORLD.createDriversDF`, and adjusts some population information.

- **WORLD.initONDEMAND:**

Initializes ONDEMAND object.

- **WORLD.initPEOPLE:**

Initializes all agents (PERSON objects) in simulation.

- **WORLD.initBACKGROUND:**

Loads background traffic and adds it to the appropriate edges in the graph.

- **WORLD.initUNCONGESTED:**

Computes cost of trips without congestion for comparison purposes. Uncongested trips for `ondemand` mode are the same as driving trips.

- **WORLD.createGroupsDF:**

(Should be updated to be more user friendly.) Creates dataframes with group information.

- **WORLD.createDriversDF:**

(Should be updated to be more user friendly.) Creates dataframes with driver run information

- **WORLD.plotPRELIMINARIES:**

Plots preliminary setup of simulation. Particularly, origin-destination pairs and group geographic regions.

- **WORLD.fitModels:**

Loads congestion curves (if precomputed in `WORLD.load_fits`) file. If not, calls `ONDEMAND.generateCongestionModels` to fit demand curves for each group.

- **WORLD.saveFits:**

Saves ondemand congestion curve data.

- **WORLD.loadFits:**

Loads ondemand congestion curve data if specified in `WORLD.load_fits` file.

- **WORLD.graphFromFeed:**

Generates a NetworkX graph from a gtfs feed. Used for plotting, etc.

- **WORLD.trimGraph:**

Cuts graph down to be within geographic bounding box.

- **WORLD.SIM:**

Runs main simulation loop for simulation. If input `restart=True`, then the simulation iteration number in `WORLD.main['iter']` is reset to 0. (Otherwise, the simulation continues from the current iteration.) The parameter `WORLD.main['alpha']` contains the iteration update step for updating the cost variables for each network step. (For more details, see Sec. ??).

- **WORLD.load:**
(Doesn't work at the moment.) Loads previous simulation run.
- **WORLD.save:**
(Doesn't work at the moment.) Saves full current simulation run.
- **WORLD.makeDashboard:**
Construct DASHBOARD object.
- **WORLD.generateLayers:**
Generates image layer information for DASHBOARD visualizer. Layer image passed to WORLD.plotLAYERS, WORLD.plotLayer for plotting.
- **WORLD.plotLayer:**
Plots individual image layer for DASHBOARD.
- **WORLD.plotLAYERS:**
Plots image layers for DASHBOARD. Images are stored in folder WORLD.dash_folder.
- **WORLD.generateOutputs:**
Generate output dataframes. WORLD.OUTPUTS is a dict with key tags ['by_mode', 'ondemand']. For mode in ['drive', 'walk', 'gtfs', 'ondemand'], WORLD.OUTPUTS['by_mode'][mode] is dataframe with simulation statistics about the specific mode. Each row gives information about a specific trip segment. WORLD.OUTPUTS['ondemand'] is a dataframe with more statistics specific to the ondemand mode. More details are given in Sec. 7.1
- **WORLD.printOutputs:**
Prints dataframe outputs to the screen. dfs_to_show gives the specific modes to show. If 'groups' contained in dfs_to_show, prints WORLD.OUTPUTS['ondemand']. row_count gives the number of rows to print.
- **WORLD.saveOutputs:**
Saves dataframe outputs to input filename.
- **WORLD.generate_graph_presets:**
- **WORLD.generate_polygons:**
- **WORLD.add_base_edge_masses:**

5.2 GRAPH & FEED Objects

The GRAPH objects are Open Street Map NetworkX graph objects that store transportation grid information for each mode. For the `drive`, `walk`, and `ondemand` modes the NetworkX object is used for shortest path computations. For the `gtfs` mode, the NetworkX object is generated from the a GTFS feed with custom function `load_feed_as_graph` and is primarily used to facilitate plotting. For the fixed line transite mode, a FEED object is loaded from a standard GTFS feed.

GRAPHS			FEEDS		
mode	Type	Description	mode	Type	Description
'drive'	OSMNX graph	loaded using OSMNX	'gtfs'	GTFS feed	loaded from gtfs.zip file
'walk'	OSMNX graph	loaded using OSMNX			
'gtfs'	OSMNX graph	generated using GTFS feed			
'ondemand'	OSMNX graph	loaded using OSMNX (same as 'drive')			

Figure 9: GRAPH and FEED objects

5.3 NETWORK Objects

Each NETWORK object contains information about potential trip segments within a specific transportation mode. The primary attribute of the NETWORK object is a dictionary, `segs`, which contains SEG objects with information about the current optimal route between a start node and end node within the mode network. The keys are tuples with the form `(start_node, end_node)`. The `UPDATE` function recomputes the optimal routes for each active segment within the mode network. The other attributes and functions are detailed in Fig. 10.

Attributes		
Field	Value	Description & Details
mode	str	network mode from ['drive','walk','gtfs','ondemand']
GRAPH	NetworkxG	graph for network
segs	dict	SEG objects - keys have form (start, end)
edge_masses	dict	history of edge masses on each edge - edge keys
edge_costs	dict	history of costs on each edge - edge keys
current_edge_masses	dict	current mass on each edge - edge keys
current_edge_costs	dict	current cost on each edge - edge keys
edge_cost_poly	dict	poly cost func on each edge - edge key: [a0,a1,a2,...]
people	tag	list of agents using network
active_segs	list	list of active seg tags

Functions		
Function		Description
<code>__init__</code>		generates network structures
<code>UPDATE</code>		updates shortest paths and optimal cost info for each SEG in NETWORK
<code>createEdgeCosts</code>		generate edge cost functions (based on geography edge information from OSMNX graphs)
<code>createEdgeDists</code>		compute distances on each edge in the network
<code>removeMassFromEdges</code>		removes all mass from edges in the network - for each simulation iteration
<code>addSegMassToEdges</code>		add population mass for a trip segment to appropriate edges in the network.
<code>computeUncongestedEdgeCosts</code>		Setting edge costs for specified graph to the uncontested edge costs
<code>computeUncongestedSegCosts</code>		compute uncontested segment costs for comparison purposes

Figure 10: NETWORK object

NETWORK: function descriptions

- **NETWORK.__init__:** Initializes the NETWORK object. Computes the distances along each edge in the network and generates the latency functions on each edge. For the gtfs mode, loads the precomputed gtfs data and travel times.
- **NETWORK.UPDATE:** For each edge in the network, updates the current estimate of travel time based on the previous travel time and current edge mass usage (see Section ??) for details. After the edge costs are updated, recomputes the optimal routes and costs for each segment in the network.

- **NETWORK.createEdgeDists:** Computes the distances along each edge in the network.
- **NETWORK.createEdgeCosts** Generates the latency function polynomials each edge in the network. Polynomials of the form $p(x) = a_0 + a_1x + a_2x^2 + \dots$ stored as a list of the form [a0, a1, a2, \dots]
- **NETWORK.removeMassEdges:**
Removes the current mass on each edge in the network.
- **NETWORK.addSegMassToEdges:**
Adds population mass to each edge in a particular trip segment in the network.
- **NETWORK.computeUncongestedEdgeCosts:**
Computes uncongested edge travel times for each edge in the network.
- **NETWORK.computeUncongestedTripCosts**
Computes optimal travel time for each segment without congestion (used for comparison purposes and metrics.)

5.3.1 SEG Objects

The **SEG** object is detailed in Fig. 11. The attributes contain information about the start and end node for the segment, the current population mass using the, the trip and agent ids that use the segment, the current costs for each factor, and the current optimal path. Segments in the ondemand network contain additional information about pickup and dropoff times as well as booking id and the ondemand group the segment is assigned to among other things. The functions for each segment compute the optimal path between the start and end node. The **planDijkstraSeg** uses the implementation of shortest path computations using Dijkstra for NetworkX objects given the current congestion on the network. The **planGTFSeg** uses the (precomputed) implementation of RAPTOR for shortest path computation with timed transfers.

SEG: function descriptions

- **SEG.__init__:**
Initializes trip segment within a network and computes an initial optimal route and travel cost.
- **SEG.planSeg:**
Plans routes for the segment. Calls either **SEG.planDijkstraSeg** or **SEG.planGTFSeg** depending on the type of network.
- **SEG.planDijkstraSeg:**
Plans shortest paths through the network using implementations of Dijkstra from NetworkX. Used for 'drive' or 'walk' modes.
- **SEG.planGTFSeg:**
Plans shortest paths for transit segments using a precomputed implementation of RAPTOR. Used for 'gtfs' mode.
- **SEG.pathCost:**
Computes the cost of a particular path through a given graph.
- **SEG.create_chains:** (only for gtfs) - generates a list of stops (in order) for a fixed line trip.
- **SEG.list_inbetween_stops:** (only for gtfs) - lists all stops (used or not) along a fixed line trip
- **SEG.gtfs_to_transit_nodesNedges:** (only for gtfs) - constructs nodes and edges in NetworkX graph for fixed line trip

SEG

Attributes

Field	Value	Description
seg_id	str	segment id tag, form = (start node, end node)
mode	str	mode of segment
mass	float	population mass using segment
source	node	source node in mode graph
target	node	target node in mode graph
factors	list	list of cost factors
active	bool	segment active or not
trip_ids	list	trip ids that use this segment
people	list	agents that use this segment
costs	dict	costs of each factor
path	list	history of optimal paths
current_path		current optimal path

Field	Value	Description
booking_id*		booking id for ondemand
group*		assigned ondemand group
run_id*		current run id within ondemand group
typ*		
pickup_time_window_start*	float	time: start of pickup window
pickup_time_window_end*	float	time end of pickup window
dropoff_time_window_start*	float	time: start of drop-off window
dropoff_time_window_end*	float	time: end of drop-off window
pickup_time_scheduled*	float	scheduled pickup time
dropoff_time_scheduled*	float	scheduled drop-off time
num_passengers*	int	number of other passengers in ondemand vehicle
pickup_node_id*	node	node id for pickup node
dropoff_node_id*	node	node id for drop-off node
am*	int	number of ambulatory agents using segment
wc*	int	number of wheelchair agents using segment
delivery_history*		
delivery*		

Functions

Function	Description
<code>_init_</code>	initialize SEG within a network
<code>planSeg</code>	compute optimal cost of segment, uses <code>planDijkstraSeg</code> & <code>planGTFSeg</code>
<code>planDijkstraSeg</code>	computes the optimal cost for segment where Dijkstra (shortest path) is required
<code>planGTFSeg</code>	computes the optimal cost for segment where RAPTOR is required. Uses precomputed data
<code>pathCost</code>	compute cost along a path
<code>create_chains*</code>	*only for gtfs - generates a list of stops in order for a fixed line trip
<code>list_inbetween_stops*</code>	*only for gtfs - list all stops (used or not) along a fixed line trip
<code>gtfs_to_transit_nodesNedges*</code>	*only for gtfs - constructs nodes and edges in NetworkX graph for fixed line trip

Figure 11: SEG object

5.4 PERSON Object

The PERSON object contains information about an agent in the network and is responsible for updating that agent's choice based on the lowest cost of each trip. The attributes and functions are detailed in Fig. 12. The main attribute `trips` is a dictionary of TRIP objects (detailed below) that each detail a possible trip the agent could take from their origin to destination. Each trips consist of several connected segments each possibly using a different transportation mode. The keys are tuples of the order of the mode combinations, for example ('ondemand', 'gtfs', 'walk') for a trip that takes ondemand to a fixed line transit stop, takes fixed line to a node close to the destination, and then walks from the final transit node to the destination. Right now the simulator supports the following mode combinations.

- ('drive',)
- ('walk',)
- ('ondemand',)
- ('walk', 'gtfs', 'walk')
- ('walk', 'gtfs', 'ondemand')
- ('ondemand', 'gtfs', 'walk')
- ('ondemand', 'gtfs', 'ondemand')

The `weights` contains the weights the agent puts on each factor for the trip. The `UPDATE` function queries the current cost of each trip and picks the optimal trip for that agent at the current iteration. More details on the object functions are given below.

Attributes		
Field	Value	Description & Details
person_id	str	person id tag
mass	float	Population mass
costs	list	costs for each trip option
weights	list	WEIGHTS dicts
logit_version	string	logit function to use in computing costs
trips	dict	dict of TRIP objects detailing trips (tags are mode)
trips_to_consider	list	list of all possible mode sequences - SEE BELOW
opts	list	list of mode sequences to consider
current_choice	tag	current mode sequence choice
current_cost	float	current cost of travel
cost_traj	list	history of travel cost info for agent
choice_traj		current optimal path
trips_to_consider:	possible mode sequences...	
	('drive'), ('walk','gtfs','walk'), ('walk'), ('walk','gtfs','ondemand') ('ondemand') ('ondemand','gtfs','walk') ('ondemand','gtfs','ondemand')	

Functions		
Function	Description	
<code>_init_</code>	generates agent parameters, computes agents start and end nodes for each segment of each trip including closest applicable fixed transit lines and applicable ondemand delivery groups. Also adds appropriate nodes to the node CONVERTER.	
<code>UPDATE</code>	Queries cost of each trip option for the agent and deciding which is the optimal.	

Figure 12: PERSON object

PERSON function descriptions

- **PERSON.`_init_`:**

Initialize information for each agent in the simulation. Loads pre-generated preference data from PRE object for individual agent. For each trip available to the agent, computes the start and stop nodes for each segment and generates the TRIP object. Currently, only allows one start and one stop node per segment. (This could be expanded to include more potential start and stop nodes for each segment at the cost of simulation complexity.) Each new start and stop node is added to the node CONVERTER object for later use. For ondemand modes, ondemand delivery groups are also selected using ONDEMAND.`selectDeliveryGroup` given the start and stop node for the ondemand segment.

- **PERSON.UPDATE:**

The UPDATE function loops through each trip option that an agent can take, runs the `queryTrip` function to determine the cost of trip, chooses the optimal trip and assigns the agent's population mass to the trip in the appropriate networks and assigns those segments in the appropriate network as active. There is an option, `takeall`, which assigns the population mass to every possible trip rather than just the optimal one in order to show all trips an agent can take.

5.4.1 TRIP Object

The TRIP object contains details about a specific trip an agent can take. Details are given in Fig. ???. The `structure` attribute contains a list of TRIP_SEG dicts, one for each segment in the trip. The structure of the dict for each segment is illustrated here, containing details for that trip segment, specifically possible start and end nodes for that segment.

TRIP		
Attributes		
Field	Value	Description
cost	value	current cost of trip plan
traj	list	list of optimal transition nodes between modes
active	bool	Is the agent using this trip plan
structure	list	list of TRIP_SEG dicts (should be consistent with mode sequence tag)

TRIP_SEG		
Functions		
Field	Value	Description
mode	string	Mode of trip segment
start_nodes	list	list of possible start nodes for that segment
end_nodes	list	list of possible end nodes for that segment
path	list	list of nodes in optimal path for segment
delivery_group	string	ondemand group for segment
opt_start	node tag	optimal start node for segment
opt_end	node tag	optimal end node for segment

Function	Description
<code>__init__</code>	generates the current trip
<code>queryTrip</code>	computes the optimal cost for the trip given the current login choice model
<code>applyLogitChoice</code>	combines factors using the assigned logic choice model

Figure 13: TRIP segment details object

The main function `queryTrip` computes the current cost of each trip. Note this function assumes that the segment information for each component of the trip have been updated in the corresponding network.

- **TRIP.__init__:**

Generates the TRIP object given the mode sequence and start and stop nodes for each segment.

- **TRIP.queryTrip:**

Queries the cost of each segment in the trip. (If at the time of the query, the segment does not exist in the appropriate transportation network it is generated.) The main loop, loops through each segment in the `structure` attribute in reverse order starting with the final segment and proceeding backwards. For each segment the optimal cost of traveling between each start and end node for that segment is computed using all factors, agents weighting preferences and a logit choice function. (At the moment segments, only contain one start and stop node.) The optimal cost as well as the optimal start and stop nodes for each segment are computed using a dynamic programming style calculation as illustrated here in Fig. 25.

- **TRIP.applyLogitChoice:**

Computes weighted combination of costs for logit choice model.

5.5 ONDEMAND object

The ONDEMAND object contains information specific to the ondemand transportation mode. The main attribute `groups` is a dictionary of GROUP objects that have information for each group of ondemand drivers.

ONDEMAND		
Attributes		
Field	Value	Description
groups	dict	dict of GROUP objects
grpsDF	dataframe	dataframe with group information
group_polygons	dict	dict of polygon regions for each group
driversDF	dataframe	dataframe with driver run information
num_groups	int	number of groups
booking_ids	list	list of booking ids
fit	dict	polynomial curve information for ondemand curves for groups

Functions		
Function	Description	
<code>__init__</code>	initializer ONDEMAND object	
<code>selectDeliveryGroup</code>	assign ondemand passenger to a particular ondemand group	
<code>generateCongestionModels</code>	generate congestion models for each of the ondemand groups	
<code>plotCongestionModels</code>	plot congestion curves	

Figure 14: Details of GROUP object

ONDEMAND function descriptions

- **ONDEMAND.__init__**

Generates the ONDEMAND object and generates each ondemand GROUP from the group data contained in DELIVERYDF. DELIVERYDF [‘grps’] is a dataframe containing information about the ondemand groups. For a particular group with tag, for example, ‘group0’, DELIVERYDF [‘drivers’][‘group0’] contains information about the driver runs for that particular group. The RUN objects for each driver run are generated in the GROUP object.

- **ONDEMAND.selectDeliveryGroup**

Assigns ondemand trip segments (defined by start and stop nodes) to a specific ondemand group. In general, assumes that there is only one ondemand group that will accommodate that trip segment, ie. assumes that the groups have been split up by geographic region. Each group is defined by a pickup area and a dropoff area denoted by polygons defined in the columns [‘pickup_polygon’] and [‘dropoff_polygon’] of the ONDEMAND.grpsDF dataframe. Currently a rider is assigned to an ondemand group if either pickup and dropoff node are in either the pickup or dropoff areas. (The denotation of “pickup” or “dropoff” is arbitrary.) This could be adjusted to allow pickups in only the pickup area, etc. if desired.

- **ONDEMAND.generateCongestionModels:**

Generates the ondemand congestion for each group assuming that all possible rides have been assigned to the appropriate groups. Calls the function GROUP.generateCongestionModel for each group separately.

- **ONDEMAND.plotCongestionModels:**

Plots the approximated demand curve for each congestion model for each group.

5.5.1 GROUP object

The GROUP object contains information about a group of ondemand drivers

GROUP

Attributes

Field	Value	Description	Field	Value	Description
group	str	group id tag	booking_ids	list	list of booking ids for group trips
group_ind	int	number of group	time_matrix	np.array	
date	datetime	date and time of simulation	actual_num_segs	int	current number of ondemand group segs
loc	tuple	(lat,lon) location of group depot	expected_cost	list	list - history of expected costs for group
depot	OSMNX node	(lat,lon) location of depot node	current_expected_cost	float	current expected cost for group (based on demand curve)
depot_node	OSMNX node	depot node in ondemand network	actual_average_cost	float	actual average cost for riders in group
polygon	np.array	pickup + drop-off region for group	driver_runs		
pickup_polygon	np.array	pickup region for group	runs		list of RUN objects with driver run info
dropoff_polygon	np.array	drop-off region for group	driversDF		dataframes with driver information
total_num_groups	int	total number of groups	PDF		current payload data frame
polygon_color	tuple	color for plotting group region	MDF		current manifest data frame
fit	list		current_PDF		current payload data frame
			current_MDF		current manifest data frame

Functions

Function	Description	Function	Description
<code>_init_</code>	Initialize group object	<code>constructPayload</code>	construct payload input for VRP solver
<code>generateCongestionModel</code>	fit demand curve for group given region and possible passengers	<code>routeFromStops</code>	construct full route from node stops
<code>payloadDF</code>	construct dataframe from payload information	<code>routesFromManifest</code>	construct routes thru network from stops in manifest
<code>manifestDF</code>	construct dataframe from manifest information	<code>singleRoutesFromManifest</code>	construct individual routes from stops in manifest
<code>updateTravelTimeMatrix</code>	update travel time matrix for active trips based on current traffic	<code>assignCostsFromManifest</code>	assign costs to agents using ondemand from manifest

Figure 15: Details of GROUP object

GROUP function descriptions

- **GROUP.__ init__:**

Generates GROUP object. Driver run information loaded in through the dataframe `driversDF`. RUN objects for each driver run are also created.

GROUP.generateCongestionModel:

Generates congestion demand curve for ondemand group. Should be run after all rider trips have been assigned. `counts` variable contains information about how many test cases to run to determine congestion model. `counts['num_counts']` is the total number of sample groups to compute travel time for. `possible_trips` is the full list of possible rides that the ondemand group could be responsible for. Rider counts for sample rider groups are chosen at intervals between $0.1 \times$ number of total possible rides and $0.9 \times$ number of total possible rides using the lines

```
num_pts = counts['num_counts'];
num_per_count = counts['num_per_count'];
num_trips = len(possible_trips);
sample_counts = np.linspace(0.1*num_trips, 0.9*(num_trips-1), num_pts)
```

For each count in `sample_counts`, count samples are chosen from possible trips and the average travel time is computed using the VRP solver. For each count, this sampling and travel time computation is repeated `counts['num_per_count']` times.

The resulting rider counts and travel times are used to fit a demand curve for the group as illustrated in Fig. 16 where the horizontal-axis is the number of rides in each sample and the vertical axis is the

average time in seconds. These curves are used in the simulation to estimate the expected travel time given the rider demand.

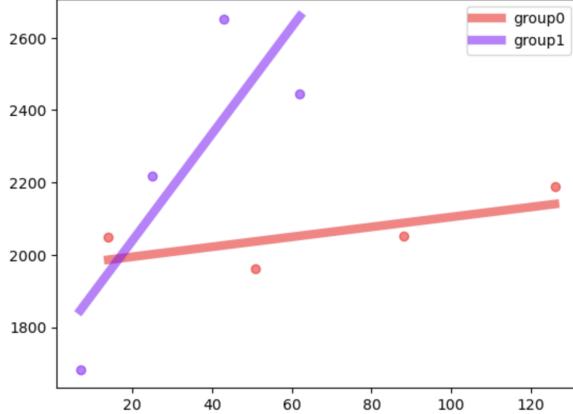


Figure 16: Sample demand curves

- **GROUP.updateTravelTimeMatrix:**

Updates the travel time matrix for the group for computation in the VRP solver. Uses the 'drive' mode graph so as to take into account the current traffic conditions.

- **GROUP.constructPayload:**

Constructs payload to pass to the VRP solver for current active riders in the group. VRP solver is called using the function

```
manifests = optimizer.offline_solver(payload)
```

- **GROUP.payloadDF:**

Constructs a dataframe to store the payload information.

```
PDF = GROUP.payloadDF(payload,GRAPHS,include_drive_nodes = True);
```

- **GROUP.manifestDF:**

Constructs a dataframe with the manifest information from the VRP solver.

```
MDF = GROUP.manifestDF(manifests,PDF)
```

- **GROUP.routeFromStops:**

Constructs a full driving route from (the ordered list of nodes from) the manifest generated by the VRP solver. **GROUP.singleRoutesFromManifest:**

Generates routes for driver runs in manifest generated by VRP solver. Used in `GROUP.assignCostsFromManifest` to assign costs to agents taking ondemand within group.

- **GROUP.assignCostsFromManifest:**

Give a manifest from the VRP solver, computes the costs for each ondemand segment taken by an agent and updates the cost information in that segment in `NETWORK.segs`. This step is important so that individuals can simply query the appropriate segment in `NETWORK.segs` to get their cost information for using ondemand for a segment of their trip.

RUN		
Attributes		
Field	Value	Description
run_id	str	run id tag
date	date	date of simulation
start_time	datetime	start time of run
end_time	datetime	end time of run
am_capacity	int	max num of ambulatory passengers
wc_capacity	int	max num of wheelchair passengers
group	str	group id of run
group_ind	int	number of group
booking_ids	list	list of booking ids for run
loc	tuple	group location
depot	node	depot node for group
depot_node	node	depot node for group
expected_cost	list	list of expected costs for run
current_expected_cost	float	current expected cost for run
actual_average_cost	float	actual average costs for
costs	dict	--
total_passengers	int	max number of passengers during run
max_num_passengers	int	max number of passengers during run
ave_num_passengers	float	ave num of passengers at any time

Field	Value	Description
DF	list	
VMT	list	history of vehicle miles traveled
PMT	list	history of ave. personal miles traveled
VMTbyPMT	list	history of ratio of VMT to PMT
current_VMT	float	current vehicle miles traveled
current_PMT	float	current ave. personal miles traveled
current_VMTbyPMT	float	current ratio of VMT to PMT
VTT	list	history of vehicle travel time
PTT	list	history of ave. personal travel time
VTTbyPTT	list	history of ratio of VTT to PTT
current_VTT	float	current vehicle travel time
current_PTT	float	current ave. personal travel time
current_VTTbyPTT	float	current ratio of VTT to PTT

Figure 17: Details of RUN object

- **GROUP.routesFromManifest:**

Generates all routes for each driver run in the manifest generated by the VRP solver. (Only used in sorting data and plotting after the fact.)

5.5.2 RUN object

The RUN object contains information about a specific driver run within a group. It does not contain any specific functions.

5.6 CONVERTER object

The CONVERTER object converts node tags between the different network modes. CONVERTER.NDF is a data frame with columns for each mode with a graph and/or a gtfs feed. In the current setup, we have the following columns: ['drive', 'walk', 'gtfs', 'feed_gtfs', 'ondemand']. Each row of this dataframe represents a separate node with id tag in each graph or feed given by the value in the appropriate column. CONVERTER.NINDS is a dictionary (of dictionaries). For each mode, CONVERTER.NINDS[mode] is a dictionary with keys given by nodes in that mode network. CONVERTER.NINDS[mode][node] then gives the appropriate (row) index for that node in CONVERTER.NDF. The indices for CONVERTER.NDF have the form 'node0', 'node1', etc. We can then do the following to convert between nodes in different mode networks. For example suppose, we want to convert a node with tag node_tag1 from the mode1 to mode2. We can use the following commands.

```
index = self.NIND[mode1][node_tag1]; # this returns the index of the appropriate row self.NDF
node_tag2 =self.NDF.loc[index][mode2]; # this returns the new node tag in the mode2 network
```

CONVERTER function descriptions

- **CONVERTER.__init__:**

Sets up converter object to convert between nodes in different networks.

CONVERTER		
Attributes		
Field	Value	Description
NDF	dataframe	dataframe with node conversion info - each contains all the tags (in ea. network) for a node
NINDS	dict	dict for locating the appropriate row in NDF for a node tag from a specific network.

Functions		
Function	Description	
<code>_init_</code>		
<code>convertNode</code>	convert node label between two networks	
<code>findClosestNode</code>	find the closest node in a particular network given a node in another	
<code>addNodesByType</code>		
<code>addNodeToConverter</code>	add a node to the node converter, finding closets nodes in other	

Figure 18: Details of CONVERTER object

- **CONVERTER.convertNode:**

Converts between node tags in two different mode networks. (Details in the explanation above)

- **CONVERTER.findClosestNode:**

When a new node is added to the node converter, this function finds the closest node within each network. (At the moment there is no check to make sure the nodes are less than some maximum distance from each other. This should be fixed.)

- **CONVERTER.addNodeToConverter:**

This function adds many precomputed nodes to the node converter including all the stop nodes on the transit lines and origin and destination nodes for each agent in the population.

- **CONVERTER.addNodesByType:**

Adds a new node to the node converter if it is not there already.

5.7 DASHBOARD

The DASHBOARD object gives a visual display of the simulation results, including a graphical display of the routes on a transportation map and also various cost metrics for the given routes.

Plotly (Python) for interactive static html: The code is built on the interactive plotting libraries plotly and plotly.express [1] and particularly their ability to print interactive plots to static .html webpages. (see <https://plotly.com/python/>) We specifically did not use any of any DASH functionality so that the dashboard can easily be displayed as a static html webpage.

Attributes		
Field	Value	Description
folder	str	folder for dashboard files
DATAINDS	dict	indices of traces, figs, buttons, etc in the dashboard data
SLIDERS	dict	dict of SLIDER objects
BUTTONS	dict	dict of BUTTON objects
TRACES	dict	dict of TRACE objects
slider_names	list	names of slider objects
button_names	list	names of button objects
button_display_names	list	display names for buttons
factors	list	factors to plot
pitgrps		(stores subplot locations)
subplots		(stores subplots)
imgpad	list	padding
slider_grid_locs	list	grid location of sliders
slider_hpads	list	horizontal padding - sliders
slider_vpads	list	vertical padding - sliders
slider_grid_lengths	list	slider lengths - in grid units
slider_lengths	list	slider lengths
button_lengths	list	button lengths
button_locs	list	location of buttons
button_hpads	list	horizontal padding of buttons
colors	dict	color info for plotting layers
edgecolors	dict	edge color info for plotting layers
opacities	dict	opacities for plotting layers
width	int	width of dashboard
height	int	height of dashboard
numx	int	num of columns
numy	int	num of rows
row_heights	list	row heights
col_widths	list	column widths

Functions	
Function	Description
<code>_init_</code>	initialize dashboard object
<code>makeGrid</code>	make grid of subplots - core of dashboard construction
<code>addSliders</code>	add sliders to dashboard
<code>addButtons</code>	add buttons to dashboard
<code>sortingData</code>	sort data for constructing traces on dashboard and generating outputs
<code>connectControls</code>	connect controls (sliders, buttons, etc) to dashboard
<code>addTraces</code>	add traces to the different plots on the dashboard
<code>addOutputs</code>	construct output dataframes from sorted data
<code>show</code>	display dashboard

Figure 19: Details of DASHBOARD object

DASHBOARD function descriptions

- **DASHBOARD.`_init_`:**
Sets up dashboard object for visualization and display.
- **DASHBOARD.`makeGrid`:**
Generates grid structure for dashboard
- **DASHBOARD.`show`:** Generates dashboard. Updates axes. Adds sliders, buttons, and traces and connects controllers.
- **DASHBOARD.`addTraces`:**
Adds traces to dashboard.
- **DASHBOARD.`addSliders`:**
Adds sliders to the dashboard.
- **DASHBOARD.`addButtons`:** Adds buttons to the dashboard.
- **DASHBOARD.`addOutputs`:**
Generates outputs.
- **DASHBOARD.`sortingData`:**
Sorts data for presentation on the dashboard.
- **DASHBOARD.`connectControls`:**
Connects controls (after generation)

5.7.1 TRACE

The TRACE object stores information for each plot and image in the dashboard figures. The current software supports three types of traces 'bar', 'box', and 'image' traces.

- 'bar': Traces on bar charts
- 'box': traces on box plots.
- 'image': Images on image plots.

Attributes		
Field	Value	Description
fig	figure	figure to show trace on
image	image	image data (for images)
x	dict	x data for trace (for plots)
y	dict	y data for trace (for plots)
sliders	dict	sliders that control trace
buttons		buttons that control trace
loc	list	grid loc of TRACE
dataind	list	index of trace data within the figure
filename	list	filename for loaded traces

Functions		
Function	Description	
<code>__init__</code>	initialize trace object	
<code>add</code>	adds trace to the given plot on the dashboard	

Figure 20: Details of TRACE object

- **TRACE.__init__:**
Initializes a trace object.
- **TRACE.add:**
Adds the trace object to the appropriate figure.

5.7.2 SLIDER

SLIDER			Attributes		
Field	Value	Description	Field	Value	Description
loc	tuple	grid loc of slider	currentvalue	int	current slider value
xanchor	str	horizontal anchor for slider	steps	list	info for each step of slider
yanchor	str	vertical anchor for slider	datainds	list	ref indices in figure data for traces referenced in the slider
active	bool	slider active or not	turnoninds	list	which positions turn on which traces
length	float	length of slider	num_steps	int	num of steps for sliders
pad	dict	dict of padding information			

Functions		
Function		Description
<code>__init__</code>		initialize slider for dashboard
<code>addTrace</code>		adds trace to the SLIDER
<code>addSteps</code>		add information for a particular step of the slider

Figure 21: Details of SLIDER object

- **SLIDER.__init__:**

Initializes slider object for dashboard

- **SLIDER.addTrace:**

Adds trace to slider. `dataind` is the reference index for the trace in the figure data. `turnoninds` is the indices of the positions of the slider where the trace is turned on. Can be more than one index if the trace turns on at more than one slider position.

- **SLIDER.addSteps:**

Sets up information for each step of the slider.

5.7.3 BUTTON

- **BUTTON.__init__:**

Initialize BUTTON object.

- **BUTTON.addTrace:**

Add data index from trace to the button.

- **BUTTON.add:**

Generate button object that toggles traces added to the button.

Attributes		
Field	Value	Description
name	str	identifier tag for button
display_name	str	name to display near button
typ	str	type of button
loc	tuple	grid location for button
xanchor	str	anchor values for button
yanchor	str	anchor values for button
currentvalue	tuple	current value of button
datainds	float	datainds associated with button
active	bool	whether the button is active
direction	str	direction of button
pad	dict	dict of pad values for button

Functions		
Function	Description	
<code>_init_</code>	initialize button object	
<code>addTrace</code>	add trace to button	
<code>add</code>	construct button for dashboard	

Figure 22: Details of BUTTON object

5.8 RAPTOR object

Separate Notebook: MULTIMODAL_RAPTOR.ipynb

The RAPTOR object is run from a separate notebook and used to precompute the optimal timed transfer paths through the fixed line network for predetermined number of possible transfers. Run times are given in the notebook and a print out is shown in Fig. 23

Attributes		
Field	Value	Description
gtfs_file	str	filename for gtfs .zip file - ex. 'carta_gtfs.zip'
save_file	str	filename for file to save computed RAPTOR trips to - ex. 'gtfs_trips_NEW.obj'
feed	gtfs feed	loaded gtfs feed
transfer_limit	int	num of allowed transfers that passengers can make
add_footpath_transfers	bool	True/False - does routing consider foot transfers

Functions		
Function	Description	
<code>_init_</code>	initializes RAPTOR object - loads gtfs feed from .zip file	
<code>calculateGTFSrips</code>	wrapper function - calls raptor_shortest_path	
<code>raptor_shortest_path</code>	main function - runs RAPTOR calculations from each stop	
<code>get_trip_ids_for_stop</code>		
<code>get_trip_profile</code>		
<code>stop_times_for_kth_trip</code>		
<code>compute_footpath_transfers</code>	computes possible foot transfers between stops (run once initially)	
<code>get_trip_lists</code>	(not used at the moment)	
<code>SAVE</code>	saves RAPTOR to RAPTOR.save_file	

Figure 24: Details of RAPTOR object

RECOMPUTE GTFS FEED

for 868 stops in CARTA transit

Run times: (roughly 1 hour per transfer all owed)

- 2 transfers = 2 hrs, 3 transfers = 3 hrs, 4 transfers = 3.85 hrs, 5 transfers = 4.85 hrs, 6 transfers = 6 hrs...

```
[*]: recompute = True;
params = {}
params['gtfs_file'] = 'INPUTS/carta_gtfs.zip';
params['save_file'] = 'data/gtfs/gtfs_trips3.obj'
params['transfer_limit'] = 3; #default if not specified here is 1.
params['add_footpath_transfers'] = True; # default if not specified here is False;
if recompute:
    RAPTOR1 = RAPTOR(params)
    RAPTOR1.calculateGTFS(); # SOLVED =
```

回 ↑ ↓ ← → ⌂ ⌃ ⌄

```
preparing to implement raptor for... 868 bus stops
time to compute foot transfers: 14.67206883430481
stop number: 0
time to add stop is... 12.489600896835327
stop number: 1
time to add stop is... 12.301872253417969
stop number: 2
time to add stop is... 12.262991189956665
stop number: 3
time to add stop is... 12.290733814239502
stop number: 4
time to add stop is... 12.308046817779541
stop number: 5
time to add stop is... 12.208059787750244
```

SAVING PRECOMPUTED GTFS...

```
[*]: saveraptor = False;
if saveraptor:
    RAPTOR1.SAVE();
```

Figure 23: Details of RAPTOR object

RAPTOR function descriptions

- **RAPTOR.__init__:** Initializes RAPTOR object - loads gtfs feed from .zip file
- **RAPTOR.calculateGTFStrips:** Main wrapper function - initially computes possible foot transfers and then calls RAPTOR.raptor_shortest_path which runs the main computation.
- **RAPTOR.raptor_shortest_path:**
Main function - runs RAPTOR calculations from each stop. Number of potential transfers is given by RAPTOR.transfer_limit.
- **RAPTOR.SAVE:** Saves computed trip info to RAPTOR.save_file
- **RAPTOR.get_trip_ids_for_stops:**
- **RAPTOR.get_trip_profile:**
- **RAPTOR.stop_times_for_kth_trip:**
- **RAPTOR.compute_footpath_transfers:**
Computes possible foot transfers between stops. Run once initially by RAPTOR.calculateGTFStrips.

6 METHODOLOGY

6.1 Travel Time Costs

At each iteration of the simulation, an estimate of travel time is kept based on current congestion in the network. For the driving network this estimate is kept for each edge/street; for the ondemand network this estimate is kept for each ondemand group. (The travel times for fixed line and walking are currently taken to be independent of congestion.) At each iteration, the travel times are updated based on a dual formulation of classical routing games. This update procedure is detailed here.

6.1.1 Routing Game Formulation - Primal & Dual

In a classical multi-commodity routing game, agents are modeled as a continuous population mass that spreads out over various routes in a network. We give a brief exposition here; more details can be found in [2] and the many references there in.

To formulate a routing game optimization problem we consider a directed graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ with nodes \mathcal{V} and edges \mathcal{E} . We can write the node-edge incidence matrix of the graph as $E \in \mathbb{R}^{|\mathcal{V}| \times |\mathcal{E}|}$.

The routing game optimization problem can be formulated as follows

$$\min_{x_{ij}} \quad \sum_e \int_0^{x_e} \ell_e(u) du \quad (1a)$$

$$\text{s.t. } x = \sum_{ij} x_{ij}, \quad Ex_{ij} = S_{ij}m_{ij}, \quad x_{ij} \geq 0 \quad \forall i, j \quad (1b)$$

Here the indices ij refer to different origin-destination pairs; $x_{ij} \in \mathbb{R}^{|\mathcal{E}|}$ is vector of mass flows on each graph edge from the population traveling from i to j and $x \in \mathbb{R}^{|\mathcal{V}|}$ is a vector of the total mass flows on each edge (from all populations combined). m_{ij} is the total population mass traveling from i to j . S_{ij} is a source-sink vector indicating the origin and destination node. By conservation of mass (under some basic assumptions like the graph is simply connected) the constraint $Ex_{ij} = S_{ij}m_{ij}$ ensures that x_{ij} represents the mass traveling from i to j on each edge. Each function $\ell_e(x_e)$ is a latency function that gives the travel time on edge e when population mass $x_e \in \mathbb{R}_+$ is traveling on that edge. $\ell_e(\cdot)$ is usually a positive, increasing

function of its argument in order to encode congestion. The objective function ensures that the effective edge cost for an infinitesimal mass agent in the simulation are the latencies functions at the current level congestion. This is ensured by the KKT conditions of the optimization problem. Specifically at optimal, stationarity with respect to x_{ij} is given by

$$\ell(x)^\top - v_{ij}^\top E - \mu_{ij}^\top = 0 \quad (2)$$

where $\ell : x \mapsto \mathbb{R}^{|\mathcal{E}|}$ is vector of all the individual latency functions on each edge, $v_{ij} \in \mathbb{R}^{|\mathcal{V}|}$ is a dual variable associated with the incidence matrix constraint that encodes the optimal cost-to-go information for population ij , and $\mu_{ij} \in \mathbb{R}_+^{|\mathcal{E}|}$ is the dual variable associated with the positivity constraint that encodes the inefficiency of traveling on a particular edge. By complementary slackness, we also have that $\mu_{ij}^\top x_{ij} = 0$, ie. no mass in the optimal vector x_{ij} is on an inefficient edge. This KKT condition can also be used as a certificate for testing the optimality of any particular route. For an indicator vector $\xi \in \mathbb{R}^{|\mathcal{E}|}$ that encodes a path from i to j , we have that $E\xi = S_{ij}$. We can then multiply ξ by 2 to get that

$$\ell(x)^\top \xi = v_{ij}^\top E\xi + \mu_{ij}^\top \xi = v_{ij}^\top S_{ij} + \mu_{ij}^\top \xi$$

Here $\ell(x)^\top \xi$ is the cost (travel time) of route ξ ; $v_{ij}^\top S_{ij}$ is the optimal travel cost from i to j (which is the same for all routes); $\mu_{ij}^\top \xi > 0$ is the inefficiency of route ξ . If ξ only uses edges that have strictly positive mass in x_{ij} , then $\mu_{ij}^\top \xi = 0$ and ξ is an efficient route.

The dual form of problem (1) can be written as

$$\min_{c, v_{ij}, \mu_{ij}} v_{ij}^\top S_{ij} - \sum_e \int_{\ell_e(0)}^{c_e} \ell_e^{-1}(u) du \quad (3a)$$

$$\text{s.t. } c^\top \geq v_{ij}^\top E + \mu_{ij}^\top, \quad \mu_{ij} \geq 0, \quad \forall ij \quad (3b)$$

Here $c \in \mathbb{R}^{|\mathcal{E}|}$ is a new variable that represents the estimated travel time on each edge. The dual problem can be interpreted as solving for the cost variable information c, v_{ij}, μ_{ij} as opposed to the mass flow information.

6.1.2 Dual Algorithm: Driving Edge Cost Updates

There many algorithms for solving these type of problems (see [2]). We make use of a dual algorithm for updating the estimated edge costs that can be summarized as follows

$$\text{INSERT ALGORITHM HERE} \quad (4)$$

For more details we refer the reader to [2].

6.1.3 Determining Driving Latency Functions

The above optimization problem is dependent on the latency functions $\ell_e(x_e)$. We compute these separately for each edge based on geographic features of each street including street length, speed limit, number of lanes, etc. We use a well-known cubic model given in [2].

$$\text{INSERT DRIVING LATENCY FUNCTION}$$

where we compute the parameters as follows

$$\text{INSERT DRIVING LATENCY PARAMETER DEFINITIONS}$$

6.2 Ondemand Cost Estimation

Inspired by the routing game formulation in the section, we employ a similar dual cost update scheme to find the equilibrium travel time costs for ondemand groups.

6.2.1 Dual Algorithm: Ondemand Group Cost Updates

At each simulation iteration, we update our current estimate of the travel time for each group using the following formula inspired by 4.

$$\text{INSERT ONDEMAND COST UPDATE STEP} \quad (5)$$

where $\phi_g(\cdot)$ is the estimated travel time for each group based on the number of passengers that group is servicing. Intuitively, here at each step the algorithm compares the estimated travel time cost for the group with the number of passengers who decided to use it (at that iteration) and then updates the cost accordingly to be closer to accurate. Intuitively, it is important that cost updates slowly so that the population is able to converge, rather than bouncing around as it would if the cost was updated drastically.

6.2.2 Determining Ondemand Latency Functions

The ondemand latency functions are estimated empirically from data using the VRP solver. For an ondemand group with a given geographic region and a list of potential passengers, random groups of different numbers of those customers are selected and routes are planned for them in order to determine the approximate travel time for an average passenger in the group as a function of how many other passengers are also using it. Details for when/how this is done in simulation are given in Sec. 4.3. The current model fits a simple linear group latency function but more complicated curves could easily be implemented and used.

More sophisticated ways of selecting the random sample groups could also be done in the following way. The cost of other possible trips for agents in the group region could be computed and agents could be ordered by the longest travel time they would accept in order to take ondemand. Selecting agents according to this ordering would provide a more accurate picture of what the latency curve for the group would look like.

6.3 Trip Cost Computation

The overall cost of a trip segment is a combination of travel time (as estimated in using the methods in Sec. 6.1) and other factors. Right now, the code considers the following cost factors

travel time	monetary cost	number of mode switches	conveniences
-------------	---------------	-------------------------	--------------

Each agent has a cost weighting for each factor as well. We include details of how each cost factor and agent's weights are determined here.

- **Travel Time:** Travel time is estimated using the computations detailed in Sec. 6.1
- **Monetary Cost:** Monetary cost can be adjusted in the simulation. Driving costs are currently taken to be based on price of gas for trip. (Parking prices could easily be added as well.) Price per ondemand trip can be adjusted in simulation as well. Agent weights placed on monetary cost can be computed relative to median income information for demographic regions.
- **Switches:** This factor includes the number of mode switches + the number of switches on a fixed line. For example, switches= 5 for a trip that includes walking to the fixed line transit, making two bus transfers, and then taking ondemand from the end of the fixed line. The switching cost for a single trip segment is initially taken to be 1. If the segment is a fixed line segment, the number of bus transfers is added as well. ToDo: add a model for weight agent's put in cost on number of switches.
- **Convenience:** (not used at the moment) This is an abstract category that can be added to include other factors.

The overall cost for a particular trip segment is determined using a weighted cost function

$$(\text{Cost})_{\text{agent}}^{\text{seg}} = \sum_{\text{factor}} (\text{Weight})_{\text{agent}}^{\text{factor}} (\text{Cost})_{\text{factor}}^{\text{seg}}$$

The overall trip cost is given by summing up the costs of each trip segments.

$$\left(\text{Cost}\right)_{\text{agent}}^{\text{trip}} = \sum_{\text{seg}} \left(\text{Cost}\right)_{\text{agent}}^{\text{seg}} = \sum_{\text{seg}} \sum_{\text{factor}} \left(\text{Weight}\right)_{\text{agent}}^{\text{factor}} \left(\text{Cost}\right)_{\text{factor}}^{\text{seg}}$$

6.4 Trip/Route Planning

Individual trips consisting of multiple mode segments are planned using a dynamic programming formulation. For each trip, potential start and stop nodes for each segment are encoded in a TRIP object (see Sec. 5.4.1 for details).

6.4.1 Dynamic programming between trip segments

The code supports a dynamic programming framework that computes the optimal start and stop nodes for each segment (along with optimal routes for each segment) as illustrated in Fig. 25.

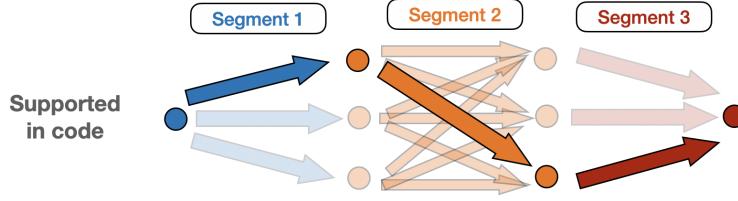


Figure 25: Dynamic programming computation of trip costs.

Currently, all trip segments are only constructed with one start and one stop node for each segment as illustrated in 26.

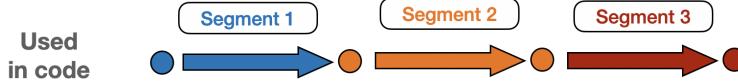


Figure 26: Simple trip segment structure used in code currently

(Adding more start and stop nodes for each trip segment may provide a significant increase in computation time.)

6.4.2 Shortest Path via Dijkstra/A* search - modes: drive, walk

Shortest path planning for driving (and walking) modes are implemented using Dijkstra/A* search using built in functions in NetworkX/OSMNX [3]. The edges weights for shortest path computations are given by the current estimate of the travel costs on each edge based on the current picture of congestion. These edge costs are updated using the dual descent framework detailed in Sec. 6.1.1. At the moment, this congestion model is only implemented for the driving mode and the walking travel times are taken to be fixed (congestion free).

6.4.3 RAPTOR implementation - modes: gtfs

Route planning for fixed line transit is done using the RAPTOR algorithm [4] which plans routes that respect timed-switches between transit lines. Our implementation was based on the discussion and code provided by Kuan Butts [5] with modifications. Our current RAPTOR module is not real time and thus must be

precomputed for a given time interval and thus currently does not model congestion of any kind on fixed line transit. Updating the code to run a real-time version could be a significant improvement to the simulation engine.

6.4.4 VRP solver - modes: ondemand

ToDo: add details of VRP solver from Mike's code.

7 OUTPUT FORMATS

7.1 Simulation Data

Output info from the simulation is stored in a dict of data frames in `WORLD.OUTPUTS`.

7.1.1 By mode

For `mode` in `'drive'`, `'walk'`, `'gtfs'`, and `'ondemand'`, `WORLD.OUTPUTS['by_mode'][mode]` contains a data frame with data for each mode. Each row corresponds to an individual trip segments. The columns are shown here.

Column	Description	Column	Description
<code>seg_id</code>	Segment id	<code>uncongested_distance</code>	distance with no congestion
<code>mode</code>	Mode tag	<code>uncongested_travel_time</code>	travel-time w/no congestion (secs)
<code>trips_ids</code>	list of trips that use segment	<code>group_id*</code>	group id
<code>active</code>	True/False: is segment active	<code>run_id*</code>	driver run id
<code>start_node</code>	start node (in mode network)	<code>pickup_time_start*</code>	start of pickup window
<code>end_node</code>	end node (in mode network)	<code>pickup_time_end*</code>	end of pickup window
<code>start_loc</code>	start location	<code>dropoff_time_start*</code>	start of dropoff window
<code>end_loc</code>	end location	<code>dropoff_time_end*</code>	end of dropoff window
<code>people</code>	people using trip segment	<code>pickup_time_scheduled*</code>	scheduled pickup time
<code>distance</code>	length of segment (meters)	<code>dropoff_time_scheduled*</code>	scheduled dropoff time
<code>travel_time</code>	travel-time of segment (secs)	<code>pickup_time*</code>	actual pickup time
<code>money</code>	monetary cost segment (\$)	<code>dropoff_time*</code>	actual dropoff time
<code>switches</code>	num of switches	<code>num_passengers*</code>	num of passengers in veh.
		<code>line_id**</code>	bus line id
		<code>bus_trip_id**</code>	spec. bus trip id

*: only included for `ondemand` mode. **: only included for `gtfs` mode.

7.1.2 Ondemand data

`WORLD.OUTPUTS['ondemand']` contains a data frame with extra data about the ondemand mode. Each row corresponds to a specific driver run within a specific group. The columns are given here.

Column	Description
<code>run_id</code>	driver run id
<code>group_id</code>	group id
<code>VMT</code>	vehicle miles traveled
<code>PMT</code>	personel miles traveled
<code>VMT/PMT</code>	ratio of VMT to PMT
<code>VTT</code>	vehicle travel time
<code>PTT</code>	average personel travel time
<code>VTT/PTT</code>	ratio of VTT to PTT
<code>distance</code>	distance of driver route
<code>travel_time</code>	travel time of driver route
<code>money</code>	monetary cost for ea. passenger
<code>total_passengers</code>	total passengers in run
<code>ave_num_passengers</code>	ave. num of passengers at any time in run
<code>max_num_passengers</code>	max num of passengers at any time during run

7.2 Dashboard

Fig. 27 shows the dashboard layout with various controls and images labeled. The sliders change which specific trip segments are highlighted on the map and in the bar charts that show the cost information. The ondemand metrics are broken down by group as well and displayed in the small side plot. (ToDo: Add side plot display for specific fixed transit lines/trips as well.)

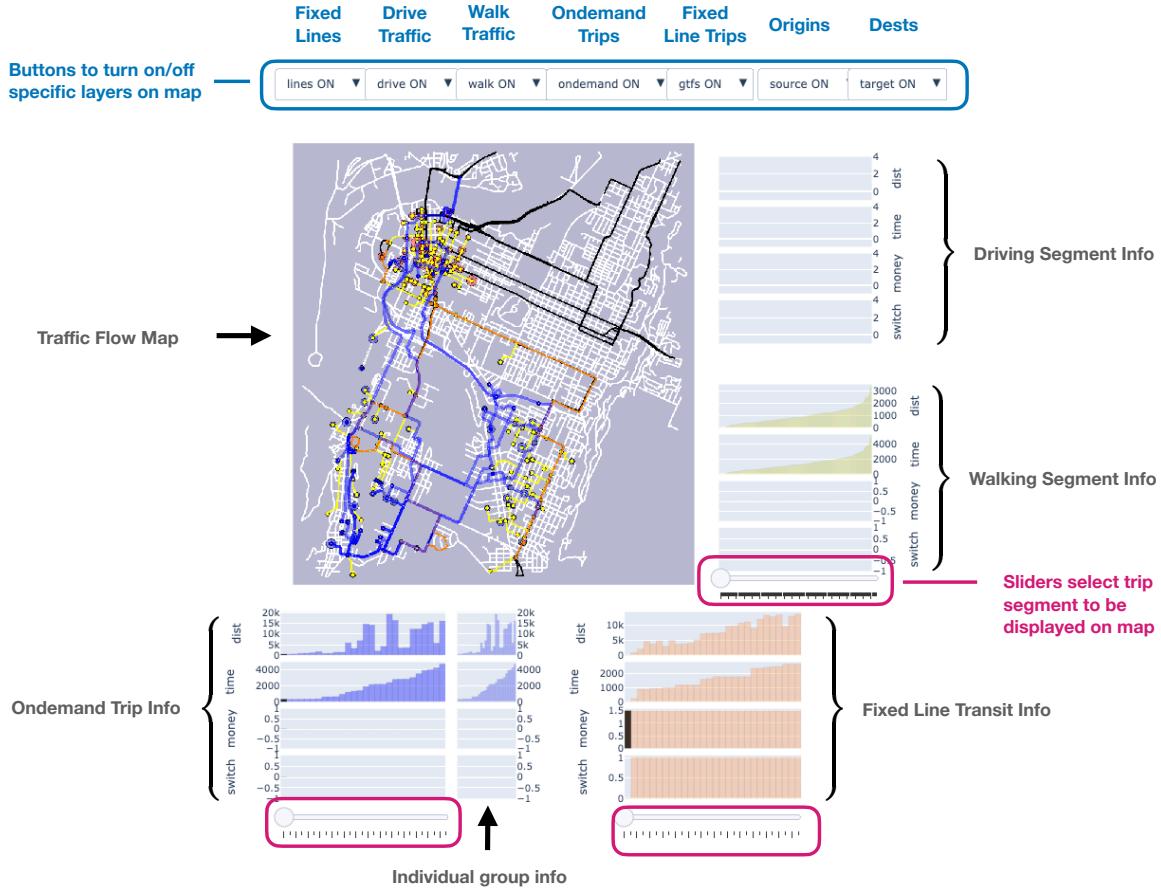


Figure 27: Details of dashboard setup.

8 GENERATING INPUTS

8.1 Generating Group Bounding Regions

The geographic boundary regions for each group can be generated using the external url geojson.io. These regions can be convenient input by hand using at the url geojson.io. The input interface is pictured in the figure below. After drawing input regions (using the toolbar options on the right), the data can be saved as a GeoJSON using the “Save” option in the top left as shown.

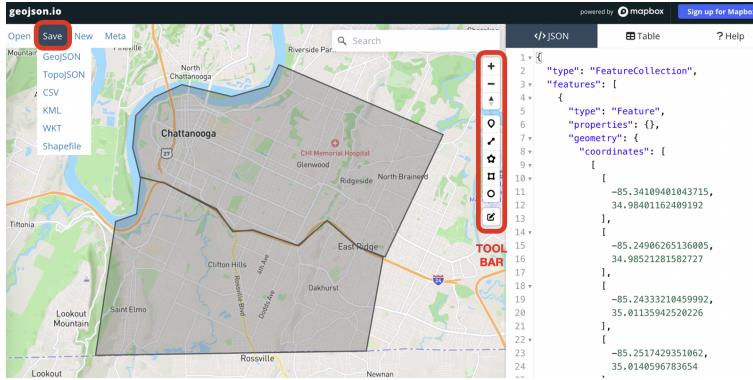


Figure 28: Input interface from geojson.io

From the .geojson file, they can be loaded in to the appropriate format using the custom function `generate_polygons` where the .geojson file is stored in `regions.geojson`.

```
polygons = generate_polygons('from_geojson', path='regions.geojson')
```

8.2 Generating Population Data

(Needs to be written based on Agrima's code.)

8.3 Generating Group & Driver Run Data

Needs to be updated to be more user friendly. Currently implemented by functions

```
WORLD.createGroupsDF
WORLD.createDriversDF
```

References

- [1] “Plotly — plotly.com,” <https://plotly.com/python/>, [Accessed 28-05-2024].
- [2] M. Patriksson, *The traffic assignment problem: models and methods*. Courier Dover Publications, 2015.
- [3] “Shortest Paths &x2014; NetworkX 3.3 documentation — networkx.org,” https://networkx.org/documentation/stable/reference/algorithms/shortest_paths.html, [Accessed 28-05-2024].
- [4] D. Delling, T. Pajor, and R. F. Werneck, “Round-based public transit routing,” *Transportation Science*, vol. 49, no. 3, pp. 591–604, 2015.
- [5] K. Butts, “RAPTOR transit routing algorithm basics — kuanbutts.com,” <https://kuanbutts.com/2020/09/12/raptor-simple-example/>, [Accessed 28-05-2024].