One-to-Many Simulator Interface User Guide

Updated: August 9th, 2022

PI: Joseph Y. J. Chow
New York University

Co-PI: Kaan Ozbay
New York University

Co-PI: Xuegang (Jeff) Ban
University of Washington

Contents

        - def simulatorsListboxEvent(self)
        - def setInputDirSelected(self)
        - def setOutputDirSelected(self)
        - def setConfigFileSelected(self)
        - def cancelSelected(self)
        - def confirmSelected(self)
    - graphVerification.py
        - def processSUMONetwork()
        - def verifyGraph()
    - matsimHelper.py
        - def readMATsimInput()
    - openFileFinder.py
        - def openFilefinder()
    - removeExtraEdges.py
        - def removeExtraEdges()
    - runSimulation.py
        - def runInterface()
    - selectRidesharing.py
        - class Leg
            - def __init__(self)
        - class RidesharingAgent
            - def __init__(self)
        - def mapMatsimNodesToRideshareNodes()
        - def calculateDistance()
        - def timeToSeconds()
        - def getRegionAgents()
    - setDirectory.py
        - class SetDirectory
            - def __init__(self)
            - def openDirectoryFinder(self)
            - def cancelSelection(self)
            - def confirmSelection(self)
    - simulator.py
        - class Region
            - def __init__(self)
        - class Simulator
            - def __init__(self)

- sumoHelper.py
  - def readSUMOInput()

➢ User Guide

Installation Guide

1. Create a new project in your Python IDE and import all of the interface files.
2. Run the following commands in the terminal. These commands will install all of the necessary Python libraries if you do not already have them installed.
   - pip install beautifulsoup4
   - pip install matplotlib
   - pip install networkx
   - pip install JPype1
   - pip install lxml
3. Once the libraries are installed, find main.py and run this file to launch the interface.

Library Documentation

Library	Purpose	Documentation Link
BeautifulSoup4	XML file parsing.	https://beautiful-soup-4.readthedocs.io/en/latest/#
Matplotlib	Used to plot graphs and create rectangle selector.	https://matplotlib.org/stable/index.html#
Networkx	Used to generate graphs.	https://networkx.org/documentation/stable/index.html
JPype1	Used to connect Python code with Java (MATSim and ridesharing simulator).	https://jpype.readthedocs.io/en/latest/
LXML	Used by	None.

	BeautifulSoup4.	

File Breakdown

Note:

- An arrow in the "Name" category indicates the class the method belongs to.
- N/A under parameters means that no parameters are needed.

main.py

- Purpose:
  - Main file for the interface.
  - Contains the code that consists of the main window of the user interface.
  - Connects all of the interface functionality provided by other files with the user interface.

- Breakdown:

Last Modified: 8/4/22 @ 12PM

Name	Parameters	Purpose
class MainWindow	**root**: Tkinter root object. **simObjDict**: Dictionary with simulator names as keys and simulator objects as values.	• Contains the code that creates the main window of the user interface. ○ Instance is created in the main function to generate the window.
def __init__(self) ➔ *MainWindow*	N/A	• Init method of MainWindow class. • All attributes are defined here. • All Tkinter objects of the main window of the user interface are defined here.
def operateRideshare(self) ➔ *MainWindow*	N/A	**TEST FEATURE ONLY.** • Used to call runRideshare() from runRideshare.py. ○ Button was used to independently activate ridesharing simulation, but this is no longer necessary as

		the ridesharing simulator is called by def operateSimulation(self) now. • Kept for testing purposes only.
def testGraph(self) ➔ *MainWindow*	N/A	**TEST FEATURE ONLY.** • Used to call processSUMONetwork() and verifyGraph() from graphVerification.py. • Performs graph verification of all MATSim sub-regions with their assigned SUMO regions. • verifyGraph() returns a dictionary with region name as the key and True / False as the value, indicating whether or not they are the same isomorphic graphs.
def testRideshare(self) ➔ *MainWindow*	N/A	**TEST FEATURE ONLY.** • Used to call getRegionAgents() from selectRidesharing.py. • Processes MATSim plans XML output for agents that use the specified sub-region and identifies random 10% to feed into the ridesharing simulator. • This function **does not** run the ridesharing simulator.
def outputContents(self) ➔ *MainWindow*	N/A	**TEST FEATURE ONLY.** • Used to print the contents of simulator and region objects.
def operateSimulation(self) ➔ *MainWindow*	N/A	• Used to call runInterface() from runSimulation.py. • Main implementation of MATSim to ridesharing simulator connection. ○ Runs one iteration of the MATSim simulation, processes the data from this simulation for ridesharing agent data, feeds data into ridesharing simulator and runs the ridesharing simulator, and produces an output containing travel times.

def createDirectoryWindow(self) ➔ *MainWindow*	**type**: String indicating if the directory is an input or output directory. • "Input" and "Output" are currently the only options.	• Used to call SetDirectory class from setDirectory.py. • Creates a window prompting the user to identify where the directory for either inputs or outputs is located.
def loadMATsimNetwork(self) ➔ *MainWindow*	N/A	• Processes MATSim's network XML file for node and edge data and plots them for user visualization of the MATSim network.
def resetSelection(self) ➔ *MainWindow*	N/A	• Clears node and edge selections for the current region as well as any rectangles created on the visualizer.
def clearNetwork(self) ➔ *MainWindow*	N/A	• Clears the network visualizer and resets any region selections made..
def onSelection(self) ➔ *MainWindow*	**eclick**: Matplotlib event indicating mouse was clicked. Used to get coordinates of click. **erelease**: Matplotlib event indicating where mouse was released. Used to get coordinates of release.	• Creates a rectangle patch to visualize the selection region and identifies the nodes that fall within the region.
def addNodeButtonClicked(self) ➔ *MainWindow*	N/A	• Used to call AddNodeButton class from addNodeButton.py. • Creates a window for users to input a list of nodes to the current region.
def createRegionClicked(self) ➔ *MainWindow*	N/A	• Used to create a new region with current region name and current region node and edge selections. ○ Adds region name to the listbox and adds region object to the self.regions dictionary.
def loadRegionClicked(self)	N/A	• Used to load the selected region (from the listbox) into the visualizer and displays the nodes from the region in

➔ *MainWindow*		the selected nodes listbox. ○ Loading a region allows the user to edit the region's selections. Changes are saved automatically.
def newRegionClicked(self) ➔ *MainWindow*	N/A	• Lets users create a new region and assign new node selections. ○ Default region name (if the user doesn't assign a name) is "Simulator Region [number of regions that exist]".
def removeRegionClicked(self) ➔ *MainWindow*	N/A	• Lets users delete a saved region.
def configureRegionClicked(self) ➔ *MainWindow*	N/A	• Used to call the ConfigureRegion class of configureRegion.py. • Creates a window that allows the user to specify the local simulator they want to assign to that particular region as well as any input and output files the interface needs to perform the simulation.
def main()	N/A	• Starts the interface.

addNodeButton.py

- Purpose:
  - Contains the code that creates the "Add Node" window along with functions that provide functionality.
- Breakdown:

Last Modified: 8/4/22 @ 12PM

Name	Parameters	Purpose
class AddNodeButton	**root**: Tkinter root object. **obj**: Region object. **positions**: Dictionary with node IDs as keys and	• Contains the code that creates the add node window of the user interface. ○ Instance is created in the MainWindow class upon

	tuple with node. coordinates as the value. **destinationListbox**: Tkinter listbox object. **graph**: Networkx graph object.	clicking the add node button.
def __init__(self) ➔ *AddNodeButton*	N/A	• Init method of AddNodeButton class. • All attributes are defined here. • All Tkinter objects of the add node window are defined here.
def addNodeClicked(self) ➔ *AddNodeButton*	N/A	• Used to add the node typed into the text box to the list of node selections.
def removeSelectionClicked(self) ➔ *AddNodeButton*	N/A	• Removes node from the list of node selections.
def resetSelectionClicked(self) ➔ *AddNodeButton*	N/A	• Resets node selection list.
def confirmSelectionClicked(self) ➔ *AddNodeButton*	N/A	• Confirms node selection and adds the nodes to the region's selection as well as all edges associated with the selected nodes.
def cancelSelectionClicked(self) ➔ *AddNodeButton*	N/A	• Closes the add node window.

configureRegion.py

- Purpose:
  - Contains the code that creates the "Configure Region" window along

with functions that provide functionality.

- Breakdown:

Last Modified: 8/4/22 @ 1PM

Name	Parameters	Purpose
class ConfigureRegion	**root**: Tkinter root object. **obj**: Region object. **simObjDict**: Dictionary with simulator names as keys and simulator objects as values.	• Contains the code that creates the configure region window of the user interface. ○ Instance is created in the Mainwindow class upon clicking the configure region button.
def __init__(self) ➔ *ConfigureRegion*	N/A	• Init method of the ConfigureRegion class. • All attributes are defined here. • All Tkinter objects of the configure region window are defined here.
def simulatorsListboxEvent(self) ➔ *ConfigureRegion*	selection: Tkinter listbox selection object.	• Identifies the selected local simulator and changes the label to reflect the selection.
def setInputDirSelected(self) ➔ *ConfigureRegion*	N/A	• Used to call SetDirectory class of setDirectory.py and specifies this should be an input directory.
def setOutputDirSelected(self) ➔ *ConfigureRegion*	N/A	• Used to call SetDirectory class of setDirectory.py and specifies this should be an output directory.
def setConfigFileSelected(self) ➔ *ConfigureRegion*	N/A	• Used to call openFilefinder() of openFileFinder.py and is used to identify the configuration file for a local simulator. ○ Currently the only one this is set up for is SUMO.
def cancelSelected(self)	N/A	• Closes the configure region window.

➔ *ConfigureRegion*		
def confirmSelected(self) ➔ *ConfigureRegion*	N/A	• Sets simulator name in the region object and adds region object to the assignedRegions list of the specified local simulator object.

graphVerification.py

- Purpose:
  - Contains the code that provides the MATSim and SUMO graph (network) verification functionality.
    - Determines if both MATSim and SUMO networks are the same isomorphic graphs.
- Breakdown:

Last Modified: 8/4/22 @ 2PM

Name	Parameters	Purpose
def processSUMONetwork()	**sumoNodes**: Dictionary with region name as the key and a dictionary as the value. The value has the node ID as a key and a tuple of node coordinates as the value. **sumoEdges**: Dictionary with region name as the key and a list of tuples with node coordinates as the value. **obj**: Region object.	• Processes the SUMO network XML file and identifies the nodes and edges.
def verifyGraph()	**regionsForSumo**: List of region objects assigned to the SUMO simulator object. **matsimGraphs**: Dictionary with region name as the key and dictionary as the value.	• Performs graph verification on the MATSim and SUMO regions. Uses the is_ismorphic function of the Networkx library. • Does not check if the graphs are to scale, just identifies if they are isomorphic.

	The value has node IDs as they key and a list of edges as the value. **nodes**: Dictionary with node IDs as keys and tuple with node coordinates as the value. **sumoNodes**: Dictionary with region name as the key and a dictionary as the value. The value has the node ID as a key and a tuple of node coordinates as the value. **sumoEdges**: Dictionary with region name as the key and a list of tuples with node coordinates as the value.	

matsimHelper.py

- Purpose:
  - Contains the code that processes MATSim XML configuration files.
- Breakdown:

Last Modified: 8/4/22 @ 2PM

Name	Parameters	Purpose
def readMATsimInput()	**file**: String representing the file path. **simInputData**: Tkinter listbox object. **simObj**: Simulator object. **simInputDataList**: List representing the contents of the simInputData listbox object.	• Processes MATSim's XML configuration file and saves the data it gets to the MATSim simulator object.

openFileFinder.py

- Purpose:

- Contains the code that opens the window for users to specify configuration files.

- Breakdown:

Last Modified: 8/4/22 @ 2PM

Name	Parameters	Purpose
def openFilefinder()	**simObj**: Simulator object. **simName**: String representing simulator name. **root**: Tkinter root object. **openSimLabel**: Tkinter label object. **fileName**: String representing file type. **fileEnd**: String representing the file extension. **simInputData (*Default = None*)**: Tkinter listbox object. **simInputDataList (*Default = None*)**: List representing the contents of the simInputData listbox.	• Used to open the window prompting the user to identify the configuration file.

removeExtraEdges.py

- Purpose:
  - Contains the code that processes sub-region selections and removes stored edges that are not connected to selected nodes.
- Breakdown:

Last Modified: 8/4/22 @ 2PM

Name	Parameters	Purpose
def removeExtraEdges()	**simObj**: Simulator object.	• When edge selections are stored, we store all of the edges associated with a selected node. To remove edges connected with nodes we didn't

		select, this function is called. A dictionary with region names and dictionaries of node IDs to their associated list of edges as the value is returned.

runSimulation.py

- Purpose:
  - Contains the code that runs MATSim, the code to process its outputs, and runs the ridesharing simulator (MATSim to rideshare implementation code).
- Breakdown:

Last Modified: 8/5/22 @ 11AM

Name	Parameters	Purpose
def runInterface()	**configFile**: String representing the MATSim configuration file. **simObjDict**: Dictionary with simulator names as keys and simulator objects as values. **nodes**: Dictionary with node IDs as keys and tuple with node	• Runs all of the parts of the MATSim to ridesharing simulator connection. ○ Currently it only runs one iteration of the MATSim simulation to get plans data, processes the plans file for agents to feed into the ridesharing simulator, and calls the ridesharing simulator to produce travel times for the selected agents. • Note: The process of

selectRidesharing.py

- Purpose:
  - Contains the code that provides part of the MATSim to rideshare implementation functionality.
    - Specifically, the part that processes MATSim's output plans XML.
- Breakdown:

Last Modified: 8/5/22 @ 11AM

Name	Parameters	Purpose

class Leg	**dep_time**: String indicating departure time. **trav_time**: String representing total travel time.	• Class containing the data for a particular leg of an agent's route. • Used only to organize data.
def __init__(self) ➔ *Leg*	N/A	• Init method of the Leg class containing attributes.
class RidesharingAgent	**id**: String representing agent ID.	• Class containing the data for a particular agent. • Used only to organize data.
def __init__(self) ➔ *RidesharingAgent*	N/A	• Init method of the RidesharingAgent class containing attributes.
def mapMatsimNodesToRideshareNodes()	**file**: String representing MATSim network XML file path.	• Used to map MATSim node IDs to the node IDs used by the ridesharing simulator. • Currently only works for the Sioux Falls network. ○ This function will work for any network files passed to it but the ridesharing simulator only supports Sioux Falls.
def calculateDistance()	**x1**: String representing first x-coordinate. **y1**: String representing first y-coordinate. **x2**: String representing second x-coordinate. **y2**: String representing second y-coordinate.	• Used to calculate the distances between two points on a 2D plane. • Used to identify which node is closest to an agent's origin or destination.
def timeToSeconds()	**time**: String representing time in HH:MM:SS format.	• Converts a time in HH:MM:SS format to seconds. • Assumes the start of the simulation is at 0 seconds.
def getRegionAgents()	**matsimObj**: Simulator object representing MATSim simulator. **matsimNodes**: Dictionary with node IDs as keys and tuple with	• For each region, this function passes through the plans XML and identifies agents that have both origins and destinations in the region. Once agents are identified, 10% are taken at random to feed into the ridesharing

	node **simObj**: Simulator object representing ridesharing simulator.	simulator. This 10% has its data written to a CSV file which the ridesharing simulator would read in.

setDirectory.py

- Purpose:
  - Contains the code that opens the window for users to specify input and output directories for MATSim and the local simulators.
- Breakdown:

Last Modified: 8/5/22 @ 11AM

Name	Parameters	Purpose
class SetDirectory	**root**: Tkinter root object. **obj**: Simulator **OR** Region object. **type**: String indicating whether or not the directory is for inputs or outputs.	• Contains the code that creates the set directory window of the user interface. ○ Instance is created in either the MainWindow or ConfigureRegion class when either the "Set Input Directory" or "Set Output Directory" buttons are clicked.
def __init__(self) ➔ *SetDirectory*	N/A	• Init method of the SetDirectory class. • All attributes are defined here. • All Tkinter objects of the SetDirectory class are defined here.
def openDirectoryFinder(self) ➔ *SetDirectory*	N/A	• Opens window prompting the user to identify the directory.
def cancelSelection(self) ➔ *SetDirectory*	N/A	• Closes the set directory window.
def confirmSelection(self) ➔ *SetDirectory*	N/A	• Confirms directory selection and saves it to the object passed.

simulator.py

- Purpose:
  - Contains the code that implements the Region and Simulator objects.
- Breakdown:

Last Modified: 8/5/22 @ 11AM

Name	Parameters	Purpose
class Region	**name**: String representing the region's name.	• Class storing data related to a defined sub-region of the MATSim network.
def __init__(self) ➔ *Region*	N/A	• Init method of the region class where attributes are defined.
class Simulator	**name**: String representing the simulator's name.	• Class storing data related to a particular local simulator.
def __init__(self) ➔ *Simulator*	N/A	• Init method of the simulator class where attributes are defined.

sumoHelper.py

- Purpose:
  - Contains the code that processes SUMO XML configuration files.
- Breakdown:

Last Modified: 8/5/22 @ 11AM

Name	Parameters	Purpose
def readSUMOInput()	**file**: String representing the SUMO XML configuration file. **simObj**: Region object representing the region being assigned to the SUMO simulator.	• Processes a SUMO XML configuration file to identify input files.

User Guide

Last Modified: 8/5/22 @ 11AM

1. Run the main.py file to launch the user interface.
2. Once a window titled "NYU MATsim Interface" loads, go to the

MATSim Settings frame and specify the input directory containing the MATSim configuration, plans, network, and the other input files you need to perform a MATSim simulation.
    a. Hit the “Set Input Directory” button.
3. Once the input directory is specified, you can specify the path to the MATSim configuration file.
    a. Hit the “Select Configuration” button.
4. Specify the output directory where the outputs of the MATSim simulation would be stored.
    a. Hit the “Set Output Directory” button.
5. Once MATSim’s settings are set, you can begin using the visualizer.
    a. Hit the “Load Network” button.
6. Upon hitting the “Load Network” button, a graph representing the simulation’s network will be displayed.
7. To select a sub-region, you can use either the “Add Node” window or the rectangle selector.
8. To use the “Add Node” window, hit the “Add Node” button and specify the list of nodes you want to add to the region.
9. To use the rectangle selector, left click on the visualizer and drag the mouse to create a rectangular selection.
10. You can use either tool to select nodes.
11. Once you’ve selected the nodes you want to assign to the region, you can name the region before hitting the “Create Region” button to save the selection. You only need to hit the “Create Region” button to save the region initially, to make edits to it later you do not need to hit the “Create Region” button again. Any changes made will automatically save.
    a. Changes to be made to previous selections using the “Load Region” button to reload the selected region or the “Remove Region” button to delete a selected region.
12. Set a localized simulator as well as any input or output files it needs using the “Configure Region” button.
13. To create a new region, hit the “New Region” button.
14. Once all of the selections are made, hit the “Begin Simulation” button to launch the interface.

Note: Currently the only supported simulator is the ridesharing simulator.