Documentation of Code

Purpose

This documentation is part of the Master Thesis “MATSim as a Panning Tool for Public Transport Networks” on https://github.com/SamarkEthz/MasterThesis. This document should be carried along the project. The purpose of the documentation is to facilitate getting to know the tool in order to collaborate. Moreover, it structures the code, which is over 25’000 lines long and links it to the thesis. The tool itself aims at designing transit networks in a given or a new scenario. Specifically, the code has been implemented for designing a metro network, but it is built generically so that other transport modes are possible as well. The user is highly encouraged to carefully read the thesis before working with the tool. After review of the underlying concepts and the evolutionary approach, the code will be significantly easier to follow. Moreover, this is just a rough description of how the code works. Comprehensive commenting is performed directly in the code itself, making it easier to relate.

Folder Structure

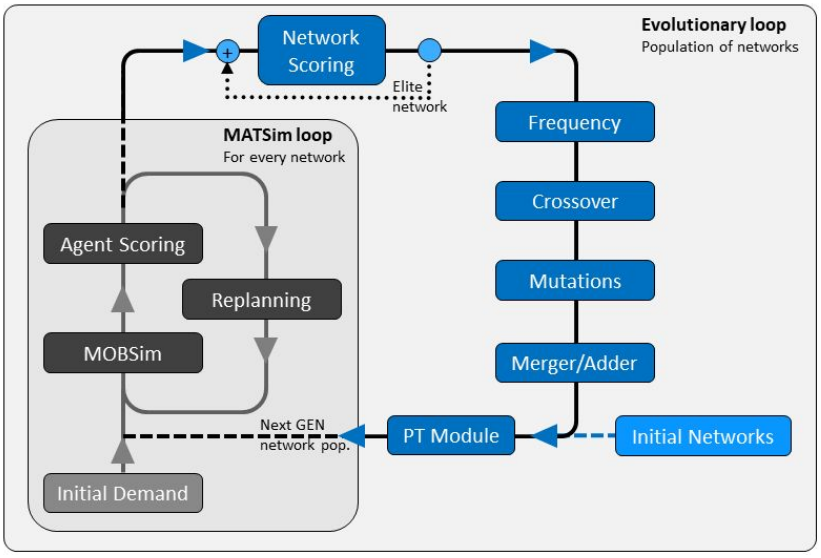
This tool was originally created in a Java project. This documentation considers the src folder the first hierarchy level. The latter contains which the .class respectively the compiled .java files of the code. The essential files are all provided on the GitHub repository above and include at least the src folder and the pom.xml file.
The entry to the actual file structure of the tool is *zurich_1pm*. Unfortunately, this has been named with the 1pm suffix. It does, however, not matter what scenarios are used in the project. Note that all file paths will therefore start with “zurich_1pm” irrespectively. A minimal zurich_1pm folder has also been put on GitHub. To use all of the classes of the code, the following list should be included within *zurich_1pm*:

- *Zurich_1pm_SimulationOutputEnriched:* Folder to hold the simulation output of reference sceanrios i.e. the base scenario, on which new networks should be planned.
- *bgImgMedium.png*: An image of the base scenario to draw resulting routes on
- *Evolution*: Folder, in which all newly designed networks are built and stored for processing.
- *cbpParametersOriginal*: Folder to store performance indicators of the reference scenario (e.g. average travel times, users of a specific transport mode). This is a processed extraction from the simulation output of the reference scenario. It contains cbp-Files, which will be discussed below.
- All the scenario files: Different base scenarios are available at IVT, ETH Zürich. All the scenario files must be placed in the *zurich_1pm* folder as shown exemplary in the GitHub version. The files include the configuration file for the MATSim simulations and information about the infrastructure and agent population using such.

Reference Simulations

The tool is intended for planning and analyzing new scenarios with respect to a given situation. Therefore, reference simulations have to be performed first! This can be done in the Class *Run_ZurichScenarioEnriched*. Here, the MATSim simulation resp. its controller can be configured. The arguments are by default [--model-type tour --fallback-behaviour IGNORE_AGENT *lastIteration*], where the last iteration of MATSim can be set by the user. After the simulation is complete, there is data available to compare to. It is recommended to set *lastIteration=100.* MATSim usually reaches equilibrium after 30-50 iterations so 100 given a nice and stable steady-state interval, that can be averaged to get as precise as possible values. The output of the simulations is automatically stored in the *Zurich_1pm_SimulationOutputEnriched* folder.

Code Structure



The code generally follows the schematic above. The procedure is built as an evolutionary loop to iteratively develop networks and services thereon. For more information on the evolutionary network planning procedures the user is referred to the thesis itself explaining every step in detail with the scientific background. The code controls the entire process through its *main* files. For a Zurich scenario this is the Class *NetworkEvolution* and for a smaller test scenario this is the Class *VC_Evolution*, where “VC” stands for “Virtual City”. At the beginning of the corresponding classes, an entire list of parameters can be set by the user. These are *all* parameters for the entire evolutionary process and will be passed on to the corresponding methods. The parameters are commented in detail within the code itself. The run arguments for running the NetworkEvolution or VC_Evolution are by default

[--model-type tour --fallback-behaviour IGNORE_AGENT NumberOfParallelNetworks NumberOfInitialPtLinesOnEachNetwork innerCityRadiusForTransportMode initialHeadway numberOfEvolutionGenerations nrIterationsToAverageSimulationResults nrMATSimIterationsWithinEachGeneration populationSize customExistingTransitScheduleModificationStrategy globalCostFactor]. A modification strategy of existing PT could for instance be removing all trams, where one would simply type *tram* as argument. The globalCostFactor scales all the costs in the system so that the metro is not overly expensive and the algorithm actually allows it to be built instead of dying out very fast due to unprofitability. Let's get started with the entire evolutionary loop now!

1. *NetworkEvolutionImpl.createMNetworks()*:
   In order to construct anything, we have to define or rather build a base infrastructure, on which the networks can be built. For the metro, this means defining a network of links and nodes in the MATSim convention as well as finding feasible stop links and facilities to integrate the metro in a current scenario. The resulting files are stored in *Evolution/Population/BaseInfrastructure,* which can be accessed by all network systems, that should be built. Basically, the base infrastructure provides the constraints an environment for building transit networks. In order to start the actual development, we need to create initial networks. This is done within the *NetworkEvolution* by the NetworkEvolutionImpl.createMNetworks() method. It creates instances of the MNetwork class, which can be looked at in detail in the class itself. Basically it stores a standard MATSim Network object along with a bunch of information. The most important data it carries along is the .RouteMap, that contains all routes of the underlying network. Once the network has been simulated, it also carries key performance indicators on the simulation output, which is required for the calculation of the fitness of a network, i.e. the welfare that has been created or destroyed by introducing additional infrastructure such as metro lines. The important part here is that the MNetworks are each a scenario for themselves and are treated as an individual solution candidate in competition with all other candidates. A number of MNetworks, usually 16, is built and all of the resulting candidates are placed in the network population MNetworkPop, which is the output of the just discussed method. Building an MNetwork means building its routes, transit lines and merging everything to a functional transit schedule, also known as the "PT module depicted in the loop above" (see MATSim documentation for how to build a transit schedule). All this is done within the *.createMNetworks()* method according to the user-defined parameters in the *NetworkEvolution* main class.
2. Now the MNetworkPop can enter the actual evolutionary loop. Before it can be simulated, it's most important features are stored in the folder *Evolution/Population/HistoryLog*, from where a simulation can be restarted anytime. In order to provide this functionality, the HistoryLog must store the .xml files of all MRoutes that each network contains. In addtion, it will make a safe copy of the transit schedule and its vehicles built to simulate this generation.
3. Next, the MNetworkPop enters the MATSim loop, where every individual MNetwork is simulated by the MATSim environment. To speed up this process, instead of calling *NetworkEvolutionRunSim.run()*, parallel threads can be executed by

RunnableRunSim(). This will be performed if the boolean “enableThreading” is set to true at the top of NetworkEvolution. The MATSim controller can be manipulated as desired within the RunSim class selected. By default, the code makes the configuration file point to the zurich_1pm folder for general files like the population and will to the Network folder in *Evolution/Population* for the specific files of this MNetwork candidate i.e. the transitSchedule and transitVehicles. The simulation output is stored within the individual MNetwork folders in the *Evolution/Population* directory. The folders are named self-explanatory as “Network”+Nr.

4. After the simulation, all MNetworks are scanned for events of interest, that happened during every iteration of the simulation. This is done to gather the condensed information to evaluate the welfare function for every network. The scanning takes place in two steps. First, the *NetworkEvolutionRunSim.runEventsProcessing* method scans the events files of the single iterations stored in the *Evolution/Population* folder to assess the usage of the metro services (commuter distance, number of users) on each line. Then, in *NetworkEvolutionRunSim.peoplePlansProcessingM()*, the output plans of all the agents are scanned to assess their travel behavior. This includes the number of agents of each mode, the average travel times within each mode, distances travelled etc. All these key parameters are stored within a cbp-file (cbp = cost-benefit-parameters), which is saved in the Network’s individual folder. The network welfare assessment only requires these condensed parameters from the simulation.
5. The actual welfare can now be calculated for each network by calling *NetworkEvolutionImpl.logResults(),* which will load the cbp-Parameters and assess the overall welfare by a comprehensive function elaborated in the thesis. In order to do so, it also has to assess the network characteristics such as route length, underground percentages or driven vehicle distance. These are responsible for the costs of the system and are opposed to the benefits brought by the new metro transit network. The results are stored in the corresponding cbp-files and partly also in the MNetworks.
6. Now, that the performance of all the networks is known, they can be developed in the evolutionary loop, which is the method *NetworkEvolutionImpl.developGeneration().* The elite network remains unchanged, while the other networks undergo frequency modification, mutations, crossover, merging and topping, which are all extensively described in the thesis and commented in the code (see *evo*-package). Once the development is complete, the MNetworkPop starts the evolutionary loop again for the next generation.
7. Last, after all generations have been simulated and hopefully great solutions have evolved, the NetworkEvolution class stores all parameters used for the simulation and shuts down.

Logging

A log file is built continuously during the simulation with outputs, where the user defines them in the code. The logger is implemented by the command *Log.write("write your log output here as a string");* By default, only errors, cautions, and key performance indicators of the networks (and routes) under development are written to the log file. The location of the log file is *zurich_1pm/Evolution/Population/LogDefault.txt* and can be changed as desired by specifying the location as one of its input arguments (see the class itself).

Recalling a Simulation

It is probable that during development of the code it will fail to execute and will probably fail on a NullPointerException :). A handy tool is the recall module to restart a simulation from any earlier generation. This functionality has been ensured by saving the most important files. To use the recall module, the used must simply set the boolean *recallSimulation=true* in the uppermost parameter list of the NetworkEvolution main class and specify the generation, for taking on development again. A possible improvement of the recall module is putting it after the MATSim loop section, so that when something fails along the way at least the entire MATSim simulation does not have to be repeated. For now, the recall module is called automatically instead of creating a new MNetworkPop. It will indeed create an MNetworkPop, but will do so by loading its parts (MRoutes+Schedules to MNetworks and MNetworks to the MNetworkPop) and adding them together.

Building an Own Scenario

For the underlying thesis, a smaller test scenario was built to develop and test some parts of the algorithm. Usually, the user would implement new transit services within a given scenario, whereas here it is built from scratch. As documented in the thesis, it is a square or rectangular city with an x-y-grid. The dimension, unit grid length, flow capacities etc. can be set by the user. This is done within *VC_NetworkImpl* in the *virtualCity*-package. In addition to several options on making the network more interesting (unsteady edges, rivers and bridges, no-build zones, tunnels), the user can choose to build transit services in the network. This can be done either by manual input or by letting the algorithm find shortest paths between random initial OD-pairs. It also includes the functionality to take existing infrastructure (facilities) and rebuild it within the realms of the the virtual city. Also, agent plans can be taken as input and they can be mapped into the city in order to simulate a population within it. This way, populations do not have to built painstakingly, but keep their well-tuned attributes and just change the location of their activities.

Visualizer Package

The visualizer package has two purposes. On the one hand, it can be used to plot the evolution of different network variables. On the other hand, it may be called to reevaluate events files and people plans. This may be useful, if changes have been made to the welfare function and the simulation output is to be reevaluated with the new function or if the plans are wished to be scanned for certain interests without having to run the entire simulation. The two most important classes in the visualizer package are explained in detail here because they are essential for the welfare calculation with respect to the reference scenario. Minimum one of the two classes MUST be performed before an evolutionary sim can be run.

After running the reference scenario in preparation of the evolutionary transit network designing process, it must be scanned for its key parameters, which are stored as cbp-files. This way, when eventually comparing a new network to the ref case, simply the differences in the cbp values have to be calculated. The cbp's for the reference scenario are stored in zurich_1pm/cbpParametersOriginal. They are explicitly calculated from the output files (selected agent plans) of the reference simulation by calling visualizer.VisualizerCBP_Original. Not only will it calculate them for each iteration, it will also average them to a global ref cbp value "cbpParametersOriginalGlobal.xml", which can be seen as benchmark simulation values after that. The user specifies the following arguments for running the class: [nrOfMATSimIterationToScan movingAverageIterNr populationFactor individualXglobal none iterationsToAverageGlobal], where movingAverageIterNr can be used to smoothen the cbp values of the individual iterations a little. In general, this is not interesting, as we can make a more global average anyways. It makes sense to keep iterationsToAverageGlobal at least 25 smaller than nrOfMATSimIterationToScan, because higher than that also the transient is part of the averaging, biasing the result. The populationFactor defines what share of the real-world population the agent population is. This is important for back-scaling the output parameters onto the entire population and assess the 1:1 real-world values.

VisualizerIterFluctuations: Similar to the VisualizerCBP_Original class, this class also has the ability to calculate cbp values. In fact, the user can decide between calculating the ones from the reference scenario simulation or from an evolutionary simulation. At the same time, it has many plotting sequences installed so that parameters like travel times, mode shares and induced benefits can be compared directly.

For further info, please consult the code directly.