PIPELINE FOR ROUTE CHOICE MODELLING WITH BFS-LE

MATTEO FELDER

Contents

1. Introduction and background 1
1.1. BFS-LE algorithm 1
1.2. Cost functions 2
1.3. Link penalty variant - LP-BFS-LE 2
2. Code and test data 3
2.1. Code 3
2.2. Test data 4
3. Pipeline 4
3.1. Network conversion 4
3.2. Importing the project in Eclipse 5
3.3. Running BFS-LE and LP-BFS-LE 5
References 6

1. Introduction and background

The Breadth First Search on Link Elimination (BFS-LE) algorithm is a route choice set generation algorithm developed by Michael Balmer. The main reference is [ARSB13]. Here we describe the pipeline to go from Adrian Meister's OSM sourced networks [MGA21] to generating and analysing non-chosen alternatives in the context of bicycle route choice modelling.

1.1. BFS-LE algorithm.

The BFS-LE algorithm is based on repeated least cost path computations. For a given OD pair, the algorithm computes the least cost path between the origin and the destination and adds the path to the choice set. Subsequently, it removes the links within this path one at the time, and calculates the respective least cost path in the network with one less link. If such a least cost path does not coincide with a previously computed route, it is added to the set of alternatives. This procedure, explained here for the least cost path in the original network, is then repeated for each newly found route and newly created network.

To organise this link elimination process, the authors in [ARSB13] make use of the following tree structure. Each node consists of a network. The root is given by the original network, while at level d, the nodes correspond to the networks for which d links have been removed. Moreover, the networks at level $d+1$ are determined by the least cost path in their parent node at level d, each corresponding to the network with one link from that route removed. Additionally, at each step the algorithm checks whether eliminating a certain link might lead to a network that is already part of the tree, and whether after deletion there still exists a path from O to D. Only networks satisfying this connectivity condition are added to the tree.

The set of alternatives is then generated by traversing this tree following a breadth first search approach. The algorithm thus starts at the root and at a given level, it explores all of its nodes, adding the corresponding least cost path to the set of alternatives if it is not yet contained in it, before passing to the next level. The algorithm stops when there are no further paths to be found for the OD pair, or when the required number of alternatives is generated. More precisely, if the required number of alternatives is reached for some node at level d, the algorithm still traverses the whole of level d, thus generating a possible larger set of alternatives, before randomly removing routes until the require choice set size is attained. This is to ensure that the composition of choice set does not depend on the processing order.

The algorithm is independent of the choice of the cost function, and we shall compare different choices below.

August 9, 2022

1.2. **Cost functions.** The least cost path calculations are based on a link cost function. Currently we have implemented the following multi-attributed link cost function, which is based on OSM-highway tags and any OSM-based cycling infrastructure. It is inspired by the open-source bicycle routing tool Brouter (http://brouter.de). More precisely, for any link a of length l_a the travel cost is of the form

$$c_a = (c(a) + o(a)) \cdot l_a + e(a)$$

where we refer to $c(a)$ as the cost factor, $o(a)$ as the one-way-penalty, and $e(a)$ as elevation cost. Currently the values of $c(a)$ are implemented as given in Table 1 and depend on whether the link is equipped with any bicycle infrastructure. Links with cycling infrastructure are those for which any combination of the following OSM-tags are used: `bicycle=yes`, `bicycle=official`, `bicycle=permissive`, `bicycle=designated`, `cycleway=track`, `cycleway=|lane|`, `cycleway=opposite_lane`, `cycleway=opposite`. Additionally, we extend this to links with either a cycle lane or separate cycling path, or links which are part of the "Velo Masterplan" as specified by the city of Zurich.

	c(a)	**c(a)**
`highway =`	**bicycle infra.**	**no bicycle infra.**
`cycleway`	1.0	-
`footway/ path/ pedestrian/ road/ track`	1.0	3.0
`primary(_link)`	1.2	3.0
`secondary(_link)`	1.1	1.6
`tertiary(_link)/ unclassified`	1.0	1.3
`residential / living-street / service`	1.0	1.1
`steps`	-	40.0

TABLE 1. Cost factor.

The one-way-penalty is applied to links which run in direction opposite to the direction of traffic without any appropriate cycling infrastructure. The OSM-tag `cycleway=opposite_lane` or `cycleway=opposite` indicated whether there is the required cycling infrastructure available, in which case

$$o(a) = 0.$$

If these tags are not used, we set

$$o(a) = \begin{cases} 50.0 & \text{if highway=primary(-link)} \\ 40.0 & \text{if highway=secondary(-link)} \\ 20.0 & \text{if highway=tertiary(-link)} \\ 6.0 & \text{otherwise.} \end{cases}$$

Furthermore, denote by $s(a)$ the average slope of link a. It is calculated as the difference in altitude between its start- and endnode, divided by its length. We set the elevation cost of a link to

$$e(a) = \begin{cases} (80 \cdot s(a) - 1.2) \cdot l_a & \text{if } s(a) \geq 0.015 \\ 0 & \text{otherwise.} \end{cases}$$

For instance, for a slope of $s(a) = 0.0275$ the additional cost is given by the length l_a of the link, and with each additional 1.25% of slope, the elevation cost increases by the value of the length of the link. Also, slopes lower than 1.5% (i.e. $s(a) = 0.015$) do not generate additional link cost through the elevation factor.

1.3. **Link penalty variant - LP-BFS-LE.** To ensure a greater degree of heterogeneity among the alternatives, we explore a link penalty variant of the BFS-LE algorithm (LP-BFS-LE). For this, before each new least cost path computation, the cost of each link in the network is updated according to the number of times a link has been used by the routes that are part of the choice set up to that point. More precisely, at the start, the choice set is empty, and the link cost function for any link a reads

$$c_a^{(0)} = c_a.$$

Assume now that we have already computed k alternatives. For the next least cost path computations we define the link cost function for a link a as

$$c_a^{(k)} = c_a + \mu \cdot n_a^{(k)} \cdot l_a$$

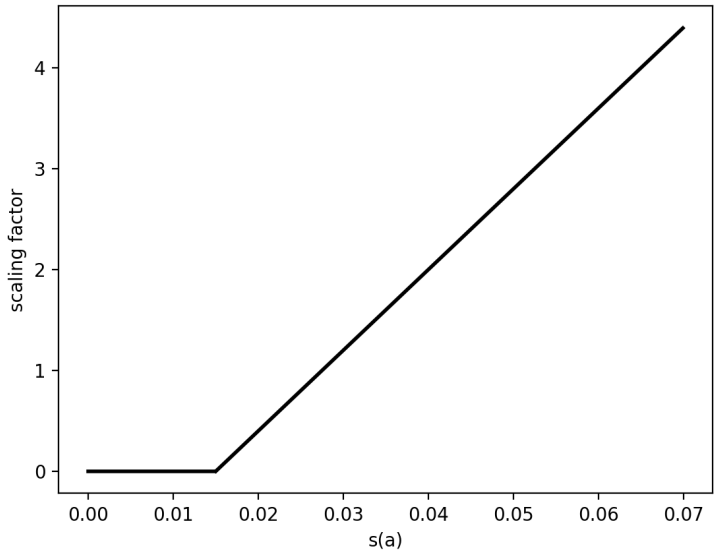

FIGURE 1. The scaling factor used for the elevation cost $e(a)$.

where μ can be chosen by the analyst, l_a is the length of link a and $n_a^{(k)}$ equals the number of routes in the choice set (computed up to this point and consisting of k alternatives) that use link a. We refer to the parameter μ as link penalty. The case $\mu = 0$ is the original BFS-LE algorithm.

2. Code and test data

2.1. **Code.** Most code should contain some documentation on what it is supposed to do. Since I am new to this, feedback is very welcome - just ask if you need more information. For most of the files you will find a few more remarks below. The documentation below is only for the files in the folder **org.ivt.breadthfirstsearchonlinkelimination**, the equivalent versions work analogously for the other use cases and should be self-explanatory.

SimpleAnalysisBFSLE.java. When called in the *RunSimpleAnalysisBFSLE.java file*, it reads the input file, calls the BFS-LE algorithm and analyses the chosen route as well as the generated alternatives, and writes out the result to the output file. It reports the computation time, the length, the number of links for the chosen route and each alternative in the choice set.

AnalysisBFSLE.java. When called in the *RunAnalysisBFSLE.java file*, it reads the input file, calls the BFS-LE algorithm and analyses the chosen route as well as the generated alternatives, and writes out the result to the output file. It reports the length, the number of links and the path size factor of the chosen route and each alternative in the choice set. Additionally, it returns the overlap and buffer overlap with the chosen route of each alternative. Notice that the alternative with the greatest overlap with the observed route is remove from the choice set (prior to the analysis) if its overlap exceeds 70% - otherwise, the alternative that is added last is removed.

BFSLE.java. The code implementing the BFS-LE algorithm. This is (up to small modifications) due to Michael Balmer, a former PhD student and postdoc at IVT.

ChoiceSetWriterBFSLE.java. When called in the *RunChoiceSetWriterBFSLE.java* file, this class reads the input file, calls the BFS-LE algorithm and writes out link-by-link each route in the choice set (including the chosen route) to the output file.

ChoiceSetWriterBFSLENoObservedRoute.java. This file is used when no observed route is available (i.e. the only input is an OD pair). When called in the *RunChoiceSetWriterBFSLENoObservedRoute.java* file, this class reads the input file, calls the BFS-LE algorithm and writes out link-by-link each route in the choice set to the output file.

DistanceCostFunction.java. A distance based link cost function. In the other use cases the cost function is multi-attributed.

RunAnalysisBFSLE.java. Calls the simple analysis function, loads the network, produces the output file. To run it, follow the instructions below.

RunSimpleAnalysisBFSLE.java. Calls the simple analysis function, loads the network, produces the output file. To run it, follow the instructions below.

RunChoiceSetWriterBFSLE.java. Calls the choice set writer function, loads the network, produces the output file. To run it, follow the instructions below.

RunChoiceSetWriterBFSLENoObservedRoute.java. Calls the choice set writer function in the case where there is no observed route available (i.e. the only input is an OD pair), loads the network, produces the output file. To run it, follow the instructions below.

The following additional tools are used throughout the project.

StreetSegment.java. This class is used in the BFS-LE algorithm. In particular, the algorithm works with a simplified network by merging links that pass through nodes which do not model junctions or intersections into one street segment (see [ARSB13, Section 2.1.2]).

AnalysisTools.java. Contains functions to compute the length of a path (i.e. route), the path size factor and the commonality factor. Notice that there exist several versions for the path size factor. The one we use can be found in [HRSANP14, Section 3.3].

ParseInputFile.java. Converts the input file to a Java dictionary mapping the ID of each OD pair to the a list containing the origin, the destination and the chosen route (if available, it is recorded as a path in MATSim). This dictionary is then used as an input in all *(Simple)Analysis... .java* and *ChoiceSetWriter... .java* files. If you need to input more information, or input information in a different format, you can adapt this file accordingly.

2.2. **Test data.** The folder **data** contains a network typically used for performing very simple tests in MATSim, a tsv-file `test_od_pair.tsv` in the correct format containing an OD pair (with ID equal to 1, origin at Node 1, destination at Node 13), and a possible chosen route (Link 1, Link 6, Link 13, Link 20). You can also experiment with the case where only an OD pair is given, but no observed route is available by using the second tsv file (`test_od_pair_no_obs.tsv`).

3. Pipeline

Depending on the network, the pipeline consists of either one or two main steps. First, the network needs to be converted to a MATSim compatible format. Secondly, the various analysis files need to be run.

3.1. **Network conversion.** If you already have a clean MATSim network, this step is not needed.

The Python scripts (in the `network_conversion` folder) used for converting OSM networks to MATSim networks are very specific to our use case, and only those examples are included in the repository. You may adapt them to your application, or simply write your own. The main steps and challenges are:

(1) the links in Meister's input network are undirected [MGA21]. In case traffic runs only in one direction, this is specified by a one-way-attribute. However, within the MATSim framework links are required to be uni-directional. The first basic network conversion step consists of converting each link into one link per admissible direction while also redistributing the link attributes over the two newly created links.
(2) The map matching algorithm does not return the observed route as a sequence of link IDs, but rather in terms of a list of pairs of node IDs. Thus, one issue that arises is that some pairs of node IDs might give rise to links missing in the network. This might occur when a cyclist drives in the direction opposite to the the direction of traffic, or simply because of some imprecision in the map matching procedure. A second network conversion step thus entails making sure that all links in the observed routes are contained in the network, in particular by adding any missing ones.

(3) In a third and last network preparation step, we apply a MATSim-specific network cleaning tool (*NetworkCleaner.java*, [HNA16]) which retains only the largest strongly connected component within the network, i.e. it ensures that in the final network every node can be reached by every other node.
(4) In particular, this last step might remove edges which are part of an observed route. Since this event does not occur often, we (for now) simply need to remove the corresponding OD pair by hand.

3.2. **Importing the project in Eclipse.** The Github repository can be found at `https://github.com/MatteoFel/MATSimBFSLE.git`. At this point, I am unsure whether it will be publicly available - email me if you are interested and its access is restricted.

(1) Download and install Eclipse.
(2) Fork the repository **MATSimBFSLE** from Github
(3) In Eclipse, under **file -> import**, go to **Maven -> Existing Maven Projects**.
(4) Go to the directory where you saved the folder **MATSimBFSLE** and select the respective Maven project. Press **Finish**.
(5) Make sure you are working with Java 11. To check whether this is the case, open the package explorer (**Window -> Show View -> Package Explorer**). Check that the **JRE System Library** is **JavaSE-11**. Otherwise, right-click on **JRE System Library** and go to **Properties**. Under **Environments**, choose **JavaSE-11** if you can (you might need to download this somewhere).

3.3. **Running BFS-LE and LP-BFS-LE.** There are four folders that can be used to produce an output:

- org.ivt.breadthfirstsearchonlinkelimination
- org.ivt.linkpenaltybfsle
- org.ivt.linkpenaltybfslezurich
- org.ivt.linkpenaltybfslegreaterzuricharea

The first two can be tested using the **data** folder whereas the third and fourth require additional data specific to the city of Zurich. These are for internal use only.

Each folder contains two to four Java files that can be run as Java applications. They are either called

- **Run(Simple)Analysis... .java**
- **RunChoiceSetWriter... .java**

where the **...** part describes the use case at hand.

For instance, to run the **RunAnalysisBFSLE.java** file (or any of the other run files), go to **Run -> Run Configurations**.

On the left, go to **Java Application**. For **Project**, choose **MATSimBFSLE** and for **Main class**, choose **org.ivt.breadthfirstsearchonlinkelimination.RunAnalysisBFSLE** (you might need to search for RunAnalysisBFSLE).

Under **(x) = Arguments**, enter the required data into the **Program arguments** slot. This consists of

- the number of alternatives to be generated (a non-negative integer)
- the variation factor (an integer greater or equal to one, default is 1)
- the maximal computation time (in seconds, max is 604800000 = 1 week)
- a MATSim input XML network file (for example the one in your **data** folder)
- the path of your input file
  (a tsv-file, containing for each of your OD pairs, the **ID** of your OD pair, the **origin node**, the **destination node**, a **list of links** in the observed/chosen route – for example as given in your **data** folder)
- the path of your output file (this can be a tsv- or a csv-file).

For example, using the **data** folder the input might look as follows.

```
5
1
604800000
```

```
"PATH_TO_DATA/data/network.xml"
"PATH_TO_DATAdata/test_od_pairs.tsv"
"PATH_TO_DATA/data/output/test_output.csv"
```

Click on **Run**. Check your output folder to view the results.

References

[ARSB13] Kay W. Axhausen, Nadine Rieser-Schüssler, and Michael Balmer. "Route choice sets for very high-resolution data". en. *Transportmetrica A: Transport Science* 9.9 (2013), pp. 825 –845.

[HRSANP14] Katrín Halldórsdóttir, Nadine Rieser-Schüssler, Kay W. Axhausen, Otto A. Nielsen, and Carlo G. Prato. "Efficiency of choice set generation methods for bicycle routes". en. *European Journal of Transport and Infrastructure Research* 14.4 (2014-12), pp. 332 –348.

[HNA16] Andreas Horni, Kai Nagel, and Kay W. Axhausen, eds. *The Multi-Agent Transport Simulation MATSim*. en. London, 2016-08.

[MGA21] Adrian Meister, Joyeeta Gupta, and Kay W. Axhausen. "Descriptive route choice analysis of cyclists in Zurich". en. In: 21st Swiss Transport Research Conference (STRC 2021); Conference Location: Ascona, Switzerland; Conference Date: September 12–14, 2021. Ascona: STRC, 2021-09.

Matteo Felder: IVT, ETH Zurich, Zurich, Switzerland
Email address: matteo.felder@ivt.baug.ethz.ch