Code Documentation for 'Settling Payments Fast and Private: Efficient Decentralized Routing for Path-Based Transactions'

Stefanie Roos, sroos@uwaterloo.ca

November 25, 2017

In this document, we provide some guidance on how to reproduce the results in the paper 'Settling Payments Fast and Private: Efficient Decentralized Routing for Path-Based Transactions', by Stefanie Roos, Pedro Moreno-Sanchez, Aniket Kate, and Ian Goldberg, to appear in NDSS 2018.

Our documentation is divided in four parts. In the first two parts, we document how to reproduce our results from the processed data sets. Afterwards, we give the steps for obtaining these data sets from the original Ripple data. Last, we provide advice for troubleshooting. The code is written in Java and makes use of the graph analysis tool GTNA [1].

1 Static Simulation Setup

To reproduce our results for the static scenario, we provide the class `Static.java` (located in package: `treeembeddings.tests`). You have to configure the path to the location of the required data sets and provide the parameters governing your run before running the experiment.

You have to configure the variable `path` in Line 35 to point to your local folder containing the data sets. The program will then expect to find the data files used to generate the credit network at `path+finalSets/static`. In addition, you might want to configure the location that the results are written to by modifying Line 34, by default the program creates a folder `data` in the same directory as the source code. Note that the current configuration will overwrite previous results when run with the same parameters, you can change this behaviour by replacing Line 33 (`Config.overwrite("SKIP_EXISTING_DATA_FOLDERS", "false");`) with
`Config.overwrite("SKIP_EXISTING_DATA_FOLDERS", "true");`.

The program takes 4 parameters. First, you have to specify which of the 20 lists of transactions you want to use for the run by providing an integer in the range **0,...,19**. Second, you specify the routing algorithm you aim to use. We implemented 11 possible algorithms, encoded as integers in the range **0,...,10** as follows (information on the different algorithms can be found in the paper):

- 0: LM-MUL-PER(SilentWhispers)
- 1: LM-RAND-PER
- 2: LM-MUL-OND
- 3: LM-RAND-OND
- 4: GE-MUL-PER
- 5: GE-RAND-PER
- 6: GE-MUL-OND
- 7: GE-RAND-OND (SpeedyMurmurs)
- 8: ONLY-MUL-PER
- 9: ONLY-RAND-OND
- 10: max flow (Ford-Fulkerson)

[1] `https://github.com/BenjaminSchiller/GTNA`

The third parameter specifies how often initially unsuccessful transaction should be tried again. Commonly, the value is an integer less than 5. The fourth and last parameter is only required for the routing algorithms 0 to 9, as it gives the number of landmarks, or equivalently trees or embeddings, that are used. For instance, if you want to run SpeedyMurmurs (i.e., algorithm 7) on transaction list 0 with 1 retry and 3 trees, you would use the parameters `0 7 1 3`.

In its current version, the program chooses the nodes with the highest degrees as landmarks. If you want to use random landmarks, you can adapt the program by changing Line 69 to
`int[] roots = CreditTests.selectRoots(degFile, true, trees, i);`.

After the experiments are done, you can check your result folder, which will detail a number of metrics associated with the experiments. For each set of parameters, the program creates a folder of the form `READABLE_FILE-*` with * containing the parameters of your run among other information (see last line of the output produced by the program to see the exact folder name). In this folder, there are sub-directories for each run (0–19 if you used all 20 transaction lists), each of which contains the statistics of the individual run. The metrics used in the paper such as success ratio, delay, and overheads are summarized in the file `_singles.txt`.

2 Dynamic Simulation Setup

To reproduce our results for the dynamic scenario with an evolving, we provide the class `Dynamic.java` (located in package: `treeembeddings.tests`). You have to configure the path to the location of the required data sets and provide the parameters governing your run before running the experiment.

Similar to the static case, you have to configure the variable `path` in Line 33 to point to your local folder for the data. The program will then expect to find the data files used to generate the network at `path+finalSets/dynamic`. In addition, you might want to configure the location that the results are written to (Line 30), by default the program creates a folder `data` in the same directory as the source code. Note that the current configuration will overwrite previous results when run with the same parameters, you can change this behaviour by replacing Line 30 with
`Config.overwrite("SKIP_EXISTING_DATA_FOLDERS", "true");`.

The program takes 3 parameters. First, you have to specify the number of the run, which varies between 0 and the number of desired simulation runs minus 1. We chose 20 for the paper. Your second parameter specifies the routing algorithm, we only considered the following algorithms for the dynamic scenario (encoding in agreement with the static scenario):

- 0: LM-MUL-PER(SilentWhispers)
- 7: GE-RAND-OND (SpeedyMurmurs)
- 10: max flow (Ford-Fulkerson)

The third parameter specifies the number of steps of the simulation that were executed previously, i.e., 0 if you starting the simulation for the first time. Due to the long duration of the experiment, we performed it in 9 steps, meaning you have to consecutively run the program with `step=0...8`. For instance, if you want to run SpeedyMurmurs and it is your first step and run, you would use the parameters `0 7 0`.

Note that we run all dynamic experiments with one retry and three landmark/trees. You can adapt these parameters in Lines 98 and 100, respectively.

Your results will be stored similar to the static scenario, with the exception that there is one folder for each step. In addition to the file `_singles.txt`, the files `cnet-succR.txt` and `cnet-stab.txt` are of relevance as they display the success ratio and the stabilization overhead for each simulation epoch (Figure 3 of the paper).

3 Data Set Construction

In this section, we state the steps involved in creating the data sets used in our experiments from a Ripple data set. We provide two of the original data sets on the webpage. You require three files, the initial graph topology (including nodes, links, and link weights), the list of transactions (including sender, receiver, value, and time), and the list of changes to the topology due to nodes adjusting link weights (including node pair, new link weight, and time). Our pre-possessing steps on the data has two purposes: i) removing inconsistencies and ii) transforming the data in the expected format for GTNA.

Before starting the main step of the file conversion, we sorted the original list of transactions (e.g., `transactions-in-USD-jan-2013-aug-2016.txt`) sets by time using bash:

```
sort -k5 -n TRANSACTION-LIST > SORTED-TRANSACTION-LIST
```

Afterwards, the main conversion is done by the program `ParseFilesToGTNA.java` (located in package: `treeembeddings.tests`). You first have to specify the following file names and locations (Line 35 to 42):

- name: name of the graph, not a file just sth to put in the header; e.g. RippleJan2013
- rawgraph: input file of the graph, can contain multiple edges between two nodes; e.g., `all-in-USD-trust-lines-2016-nov-7.txt`
- graph: first output: cleaned file without multiple edges, not yet in GTNA format
- resGraph: transformed graph file in GTNA, there will be the outputs resGraph.graph and resGraph-lcc.graph, the latter only contains the giant component
- add: input files of links added during the run; e.g, `ripple_links_history.txt`
- resAdd: transformed added links file, there will be two files add.txt and add-lcc.txt, the latter only containing nodes in giant component
- trans: input transaction file (should be sorted);
- resTrans: transformed transaction file, there will be three files resTrans.txt, resTrans-lcc.txt, and resTrans-lcc-noself.txt, the transformed file, the transaction between nodes in giant component files, and the transaction in giant component without self-transactions

The code then executes four steps:

1. Remove inconsistencies from graph file (e.g., multiple edges between same nodes)
2. Transfer into GTNA format
3. Reduce to giant component
4. Remove self-transactions

After having completing the main steps, you still have to sort the list of added links (resAdd in the above list of files) by time:

```
sort -k1 -n RESADD > SORTED-RESADD
```

Note that the sorting is easier when done during post-processing as the original list of link additions (in contrast to the transaction file) is not in unix time.

For the static scenario, we furthermore removed all transactions that could not terminate successfully (determined by running max-flow). We then choose 20 sets of 50,000 transactions uniformly at random.

4 Troubleshooting

- Ensure that the folders `config` and `lib` is stored in the same folder as the source code.
- Ensure that the external libraries in the folder `lib` are properly referenced in your Java project.