Identification of FA_1 pattern

Mittwoch, 1. Juni 2022 23:16

Preparation of common data set = Data Structure

To ease extraction of attributes and design of features we are going to need the design of optimized data structure. The format of nested *JSON Lines*, or so-called Newline delimited JSON file, is an appropriated format, which provides the following advantages:

1. The JSON object can be easily converted to Python Dictionaries and be stored either as TXT or *cpickled* binary file.
2. Dense expression saves storage, reduces file sise
3. It is hashable just by using pairs of nodes, even with direction
4. Nested object gives cleaner capsuling enabling direct handling of timestamps
5. Same data set can be used for multiple purposes, GNN feature extraction, FA_1 thresholds evaluation

The data structure could look like this:

```
{
 "node_id": "Wallet_X",
 "node": [
  ("Wallet_Y", "in"),
  ("Wallet_Z", "out")
  ("Wallet_M", "in")
 ],
 "labels": {
  "FA_1_case_1": True,
  "FA_1_case_2": False,
  ...
 },
 "transaction_val": [ # only amount, corresponding timestamps are in "transaction_time" of same position
  123,
  456,
  789
 ],
 "transaction_time": [  # epoch timestamps
  1654119683,
  1654119702,
  1654119711
 ],
 further_attributes, ...
} # End of line, Newline
{
 next object, separated only by NEWLINE, no comma!
}
```

Modules and functions associated with Data Structure- Algorithm

Associated and centered with the pre-defined Data Structure, we need to build the library to:

- Create the data from Neu4J dump
- Export for NX Graph load
- Export for feature matrix
- Evaluation for pattern recognition

We expect, at lease, to have the following functions:

- Class FileLoader:
  - def __init__(self, input_fn: str) -> None
  - def load_from_csv(self, input_fn: str) -> bool
  - def load_from_json(self, data: object) -> bool
  - def json(self) -> []
  - def write(self, output_fn: str) -> None
  - def convert_value(self, unit: str, factor: float) -> None
  - def convert_nx_graph(self) -> nx.Graph
- def test_fa_1_case_1(lst_time: []) -> bool
- def test_fa_1_case_2(lst_time: []) -> bool
- def label_fa_1_case_1(dataset: [dict]) -> [dict]
- def label_fa_1_case_2(dataset: [dict]) -> [dict]

Alysay

X_i

Idea about testing FA_1

To test FA_1 pattern, we need the following thresholds:

- Threshold for transaction value in Euro -> threshold_trans_val ~ 10 000
- Threshold for total transaction value in Euro -> threshold_tot_trans_val ~ 10 000
- Threshold for time frame (in hours), in which the numbered transactions take place -> threshold_timewindow ~ 48
- Threshold for min. outbound / inbound transactions -> threshold_trans_in / threshold_trans_out ~ 5
- For identify a diamond, paths between case_1 and case_2, the threshold of steps in the middle -> threshold_distance ~ 3 (no need to define minimum distance, as it is 0)

Basic idea:

We go through all of our nodes in the data set, which holds the adjacent nodes with directions.
At each node: Use the list "**node**" as the first mask (mask1). Then we test the list "*transaction_val*" again the threshold, *threshold_trans_val* resulting in the second mask (mask2). Then we filter the "*transaction_time*" through *mask1* & *mask2* resulting a list of candidate timestamps. Then we go through the remaining list of timestamp, for each item i we calculate T_i+threshold_trans_in/out - T_i, if the value is less than threshold_timewindow, stop, label it as positive.

The filtering, masking etc. operations can be taken from numpy.Array.

Repeat this process for all nodes in the data set.

In such way, we stop earlier have O(n) computing time and resolve the time window cutting issue.