USER MANUAL

GS-SRL
a graph sampling approach with applications to estimating the number of pattern embeddings and the parameters of a statistical relational model

Irma Ravkic, Martin Znidarsic, Jan Ramon and Jesse Davis
KU Leuven, Belgium
DTAI group at Department of Computer Science

Contents

INTRODUCTION

Counting the number of times a pattern occurs in a database is a fundamental data mining problem. It is a subroutine in a diverse set of tasks ranging from pattern mining to supervised learning and probabilistic model learning. While a pattern and a database can take many forms, we focus on the case where both the pattern and the database are graphs (networks). Unfortunately, in general, the problem of counting graph occurrences is #P-complete. In contrast to earlier work, which focused on exact counting for simple (i.e., very short) patterns, we present a sampling approach for estimating the statistics of larger graph pattern occurrences. In our paper [] we illustrates its practical behavior and provides insight into the trade-off between its accuracy of estimation and computational efficiency. In this document we provide a short manual and the demonstration of our code which can be found at

0.1 REQUIREMENTS

All our code is written in Python 2.7.14 and it heavily relies on the https://networkx.github.io/, which is the Python package for the creation, manipulation, and study of the structure, dynamics, and functions of complex networks.. More specifically we use an older version of NetworkX and we also suggest it to the user since we noticed that there are some issues with running our code with the newer versions, which we will inspect in more details and account for. The NetworkX version we use is **1.9.1**.

0.2 GRAPH AND DATA REPRESENTATION

We assume that the graphs specified by users are in the https://gephi.org/users/supported-graph-formats/gml-format/ format. For larger datasets we also assume that .gml formatted graphs might be stored as Python's pickle, more specifically, a pickled graph format called *.gpickle*. For more details please check the Networkx documentation on writing and reading .gpickles. We provide all our data in the .gpickle format. For example, if you want to read one of our data graphs and iterate through nodes you will use the following code:

```
import networkx as nx
...
d=nx.read_gpickle(path_do_data_graph)
for n in d.nodes():
        print d.node[n]
```

If you wish to turn your data in .csv format or similar to a networkx data graph, you will have to write a script by yourself. But the main Networkx concepts you are going to use is to add the nodes and edges between nodes. To do that we found the concept of Python's dictionary very useful, because it helps you knowing the mappings between the unique objects (e.g., proteins) to the associated nodes you created. In the code we shared we include some scripts showing the creation of some datasets we used (e.g., WEBKB, IMDB and YEAST). It might help you creating your own graph data from .csv and similar formats. Check the scripts in *demo/data_creation* of our shared code.

0.3 PATTERN REPRESENTATION

0.3.1 Node types

There are two types of nodes you can specify in the patterns we consider: *target nodes* and *attribute-value* nodes.

- Target nodes are those for which we count the combination of values in the data. For example, consider the pattern in Figure 1. For this pattern we will count the number of value combinations for *protein_class* and *location* nodes, given the generic values of other nodes (e.g., *interaction, 2 x protein*), and one constraint being true (e.g., *phenotype = Phenotype_id_12005*). All the non-target nodes we refer to as *the context*. In our experiments we use these obtained combinations to create a conditional probability distribution of *location* | *protein_class* <- *the context* or *location* | *protein_class* <- *context*.

- *Attribute-value* nodes have their value clamped to a specific value and they represent constraints in the patterns. Note, in your data graph all the nodes have a specific value, that is all the nodes are *attribute-value*. When you have an *attribute-value* node in your pattern you limit the template to match to only those specific value when performing the matching. Fro example, in Figure 1 the node *4: phenotype = Phenotype_id_12005* is a type of this node. This means we count the combinations of values for *location* and *protein_class* given that the value of *phenotype* of the protein with $id = 1$ is equal to *Phenotype_id_12005*.

Next, we explain how you present these two types of nodes in the .gml format.

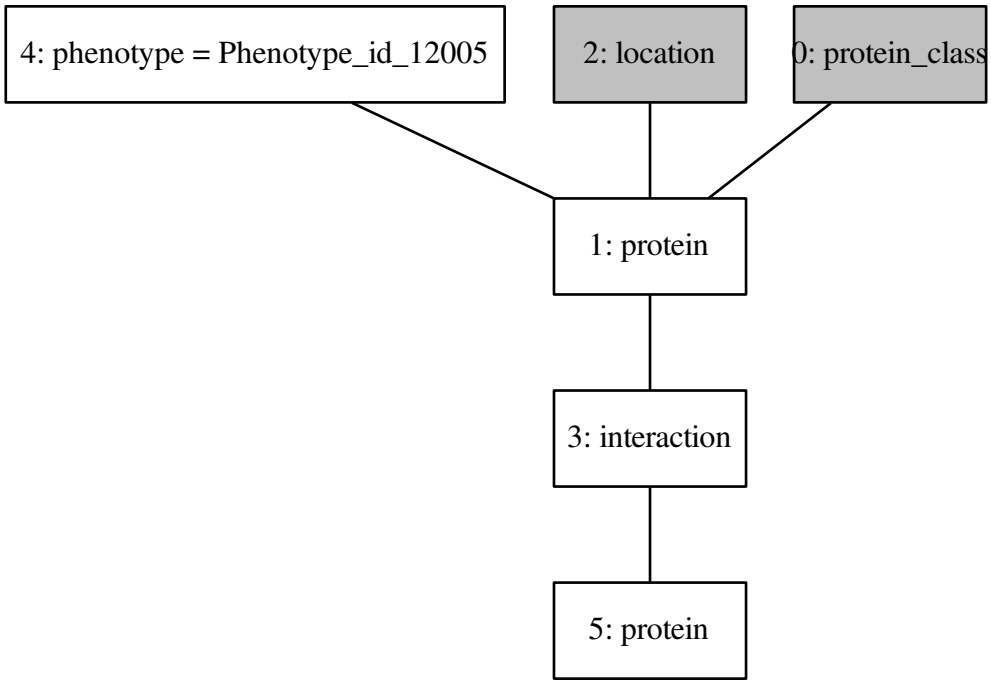

Figure 1: A pattern illustration from our demo in the project code for the YEAST dataset. The shaded nodes represent the target predicates, and "phenotype = Phenotype_id_12005" represents an attribute-value node that imposes binding constraints on the pattern.

0.3.2 Graph representation

As mentioned before, we represent graphs in the .gml format. When representing your pattern as graphs we require some information in the .gml fields.

Our nodes are having the following nodes:

```
node [
   id - an integer uniquely specifying the node
   label - a string representing the name of the node
   predicate - a predicate representing the node
   target - 0 or 1, specifying whether the node
   is the target predicate (explained above}
   valueinpattern - 0 or 1, specifying whether this node has
   a value in the pattern
   value (option) - if valueinpattern=1, you need to specify
   which is the value of the node in the pattern
]
```

For example, this is a node representing *protein_class* in the YEAST dataset and it has no specific value (*valueinpattern*=1). But it is a target predicate.

```
node [
   id 0
   label "protein_class"
   predicate "protein_class"
   target 1
   valueinpattern 0
]
```

And the following is an example of a node that has a predefined value in the pattern, that is it has *attribute=value* format in the pattern. This is a node representing the *phenotype* predicate that is bound to a specific value of *Phenotype_id_12005*. When creating your data and its associated patterns you need to make sure that the values of the nodes are valid string comparisons. That means, if your data has node values in lower case, and your pattern values are in upper case, the string matching would not succeed. So you need to make sure that this fields are consistent across the patterns and nodes.

```
node [
   id 5
```

```
    label "phenotype = Phenotype_id_12005"
    predicate "phenotype"
    target 0
    valueinpattern 1
    value "Phenotype_id_12005"
  ]
```

To see the full example pattern specification, see the *demo/test_pattern.gml* in our shared code.

0.4 RUNNING THE CODE

Once you have your data and patterns specified in the .gml format, NetworkX installed and the code in the *PYTHONPATH*, running the code is as easy as running the following command in the command line.

```
python path_to_code/approaches/X_approach.py
-d path_to_code/gssrl/demo/YEAST.gpickle
-p path_to_code/gssrl/demo/test_pattern.gml
-o path_to_code/gssrl/demo/Results_test/pattern1/
-t 2
-max_time 30
```

With this command you will count the number of embeddings with approach X. The possible approaches can be found in *approaches* package and they are:

- exhaustive: this is the exact counting approach. Results stored in *exhaustive_results*.
- fk_OBD: this is the FK-OBD approach from our paper. If the pattern has no OBD, it is run with AD, as specified in the paper. Results stored in *fk_obd_results*.
- fk_AD: this is the FK-AD approach from our paper. Results stored in *fk_ad_results*.
- random: corresponds to random sampling. Results stored in *random_results*.

Let us explain all the parameters:

- -d: path to the data file (.gml or .gpickle format)

- -p: path to the pattern file (.gml)
- -o: path to the output folder, the appropriate folder will be created: exhaustive_results, furer_results, ...
- -t: time interval. In our experiments we run each approach for a limited amount of time, and we record the results every t seconds. If you don't want to do this, do not specify this flag. A benefit of setting this flag is if your exact counting takes more time than expected, you will have temporary results saved which you can analyze.
- max_time: the maximum time allowed to each algorithm to run.

0.5 INTERPRETING THE RESULTS

If you used the *-t* flag, you will have in each approach's result folder a new folder called *monitoring*. This folder will contain all the intermediate results recorded at the multiples of *t* until the *max_time* is reached. Each file will have information on KLD and number of embeddings. We also output a *monitoring_reports.pickle* structure, which is a dictionary containing several fields that the user might be interested in for further analysis of the results. The data structure is a list with all the intermediate results. Each element has a dictionary of all the combinations of the target predicates' value encountered in the data and the (estimate of the) counts. You can get the within the code by first loading the pickle and then accessing each snapshot by doing: *monitoring_reports[0][1].current_fdict*.