Contents lists available at ScienceDirect

SoftwareX

journal homepage: www.elsevier.com/locate/softx

MGtoolkit: A python package for implementing metagraphs

D. Ranathunga [a,*], H. Nguyen [a], M. Roughan [b]

[a] *Teletraffic Research Centre, University of Adelaide, Australia*
[b] *ARC Centre of Excellence for Mathematical and Statistical Frontiers, University of Adelaide, Australia*

A R T I C L E I N F O

Article history:
Received 3 February 2017
Received in revised form 11 April 2017
Accepted 11 April 2017

Keywords:
Metagraph implementation
Computer-network policy
Policy analysis

A B S T R A C T

In this paper we present *MGtoolkit*: an open-source Python package for implementing metagraphs - a first of its kind. Metagraphs are commonly used to specify and analyse business and computer-network policies alike. *MGtoolkit* can help verify such policies and promotes learning and experimentation with metagraphs. The package currently provides purely textual output for visualising metagraphs and their analysis results.

Code metadata

Current code version	V1.0.1
Permanent link to code/repository used for this code version	https://github.com/ElsevierSoftwareX/SOFTX-D-17-00014
Legal Code License	MIT
Code versioning system used	git
Software code languages, tools, and services used	Python2.7
Compilation requirements, operating environments & dependencies	Mac OS X, Linux
If available Link to developer documentation/manual	https://readthedocs.org/projects/mgtoolkit/badge/?version=latest
Support email for questions	mgtkhelp@gmail.com

1. Motivation

A metagraph is a generalised graph theoretic structure that has several useful applications. They are commonly used to construct and analyse business policies in decision-support systems and workflow-management systems [1]. Metagraphs are also useful to analyse, optimise and troubleshoot communication-network policies [2].

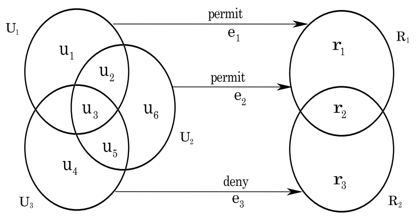
A metagraph is a directed graph between a collection of sets of 'atomic' elements. Each set is a node in the graph and each directed edge represents the relationship between the sets. A simple example is given in Fig. 1(a) where multiple sets of users (U_1, U_2, U_3) are related to sets of network resources (R_1, R_2) by the directed edges e_1, e_2 and e_3 which describes which user u_i is allowed to access resource r_j.

In this paper we describe an off-the-shelf tool for implementing metagraphs – *MGtoolkit* – implemented in Python. At the time of writing, we are aware of one other metagraph API- 'Haskell library for metagraph data structure' [3].

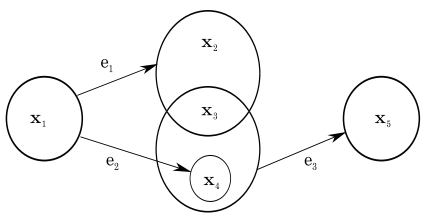
Developing a metagraph tool faces several key challenges. For instance, a metagraph does not use simple edge weights in its adjacency matrix. Also, metagraphs admit representations other than those used for simple graphs, but as in simple graphs, the representation is important for certain algorithms. In addition, there are many operations defined on a metagraph that must be supported by such a tool. These operations help analyse useful properties such as connectivity, redundancy and allow metagraph transformations, but go beyond standard graph operators.

Metagraphs have many uses in general. One in particular is in specifying and analysing communication-network policies. We will demonstrate the use of metagraphs here by taking access-control policies in a computer network as an example. But, metagraphs can be equally used in other policy contexts (*e.g.*, QoS, network-service chaining, traffic measurement *etc.*).

* Corresponding author.
E-mail address: dinesha.ranathunga@adelaide.edu.au (D. Ranathunga).



(a) Metagraph consisting of five sets and three edges.



(b) Metagraph that shows the advantage of a metapath over simple paths.

Fig. 1. Metagraph examples.

2. Background

The formal structure of a metagraph can be defined as follows:

Definition 1 (*Metagraph*)**.** A metagraph $S = \langle X, E\rangle$ is a graphical construct specified by a generating set X and a set of edges E defined on X. A generating set is a set of variables $X = \{x_1, x_2, \dots, x_n\}$ and an edge $e \in E$ is a pair $e = \langle V_e, W_e\rangle$ such that $V_e \subset X$ is the invertex and $W_e \subset X$ is the outvertex.

This definition is similar to that of a directed hypergraph, but in addition metagraphs have several useful operators and properties. One in particular is the notion of a *metapath* [1] which describes connectivity between sets of elements in a metagraph, but is somewhat different from a path in a graph.

Definition 2 (*Metapath*)**.** A metapath from source $B \subset X$ to target $C \subset X$ in a metagraph $S = \langle X, E\rangle$ is set of edges E' such that every $e' \in E'$ is on a path from an element in B to an element in C. In addition $[\bigcup_{e'} V_{e'} \setminus \bigcup_{e'} W_{e'}] \subseteq B$ and $C \subseteq \bigcup_{e'} W_{e'}$.

A metapath is more useful than a simple path (*i.e.,* a sequence of edges). Fig. 1(b) illustrates this using two simple paths from x_1 to x_5: (e_1, e_3) and (e_2, e_3). Element x_1 can reach x_5 without knowing anything about the intermediate nodes x_2, x_3, x_4 if all three edges e_1, e_2, e_3 are used but the simple paths do no capture this fact. But $\{e_1, e_2, e_3\}$ does not represent a simple path; there is no sequence of connected edges consisting of these edges. Rather, this metapath is the union of edges in two simple paths.

Reachability between a source node and a target node can be determined by finding valid metapaths between the two in a metagraph [1] (*e.g.,* the metapath from x_1 to x_5 in Fig. 1(b) is $\{e_1, e_2, e_3\}$).

Metagraphs have a property called *dominance* which allows to determine whether a metapath has any redundant components (edges or elements) [1]. A metapath is *input-dominant* if no proper subset of its source connects to the target; *edge-dominant* if no proper subset of its edges is also a metapath from the source to the target; and *dominant* if it is both input- and edge-dominant [1]. Non-dominant metapaths indicate redundancies in a metagraph and hence, redundancies in the policies depicted by the metagraph.

In metagraph theory, the notion of cutsets and bridges allow one to locate edges that are critical [1]. A *cutset* is a set of edges which if removed, eliminates all metapaths between a given source and a target. A singleton cutset is a *bridge*. In an access-control policy context for instance, bridges and cutsets indicate if there exists a critical policy or a policy set that enable access between certain users and resources.

It is also possible to derive a projection for a given metagraph. A projection is a simplified metagraph that provides a high-level view of the original metagraph by concealing certain details [1]. In a complex metagraph with many edges, a projection helps to visualise the important aspects with clarity and ease. For instance, in a complex access-control policy with many rules, projections help administrators visualise connectivity between a subset of users and resources.

Metagraphs can have attributes associated with their edges. One such attributed metagraph is a *conditional metagraph* [1]. A conditional metagraph includes propositions – statements that may be true or false – assigned to their edges as qualitative attributes [1]. The generating set of these metagraphs are partitioned into a variables set and a propositions set.

Conditional metagraphs are particularly useful in specifying access-control policies because they allow a policy (such as permit user u_1 to access resource r_1) to be activated conditionally (*e.g.,* during business hours only).

3. Overview of *MGtoolkit*

MGtoolkit is implemented in Python 2.7, which is an interpreted, object-oriented, open-source language. Python has a concise but natural syntax for many of its data types, which makes programs exceedingly clear and easy to read; as the saying goes, 'Python is executable pseudocode.' Dependencies of *MGtoolkit* include the packages NumPy 1.9 and NetworkX 1.7; both very popular and stable open source Python packages.

Fig. 2 depicts the entity model we have employed in the underlying framework. Some attributes have been omitted in the *Metagraph* entity for simplicity.

A `Metagraph` entity consists of a set of `Node` entities and a set of `Edge` entities. Each `Node` contains a subset of elements from the metagraph's generating set. An `Edge` has the members: `invertex` and `outvertex`, assigned a `Node` each, and an `attributes` member that returns any edge attributes.

A `Metagraph` entity also has the methods: `add_edges_from()` and `remove_edges_from()`, to add and delete edges as necessary. In addition, the entity includes methods to derive its adjacency matrix, find metapaths, check metapath properties (*e.g.,* `is_dominant_ metapath()`) and edge properties (*e.g.,* `is_cutset()`).

The `source` and `target` members of a `Metapath` return subsets of elements in a metagraph's generating set. The `edge_list` member returns an edge set between the `source` and `target` which satisfy Definition 2.

A `ConditionalMetagraph` entity extends a `Metagraph` and supports proposition attributes in addition to variables. A `ConditionalMetagraph` inherits the base properties and methods of a `Metagraph` and additionally supports methods to derive its context metagraphs (*i.e.,* `get_context()`), check connectivity properties (*e.g.,* `is_fully_connected()`) and redundancy properties (*e.g.,* `is_non_redundant()`).

Listing 1: *MGtoolkit* implementation of policy in Figure 1(a).

```
# define policy metagraph
variable_set = {'u1','u2','u3','u4','u5','u6','r1','r2','r3'}
propositions_set = {'action=permit', 'action=deny'}
cm = ConditionalMetagraph(variable_set, propositions_set)
cm.add_edges_from([
```

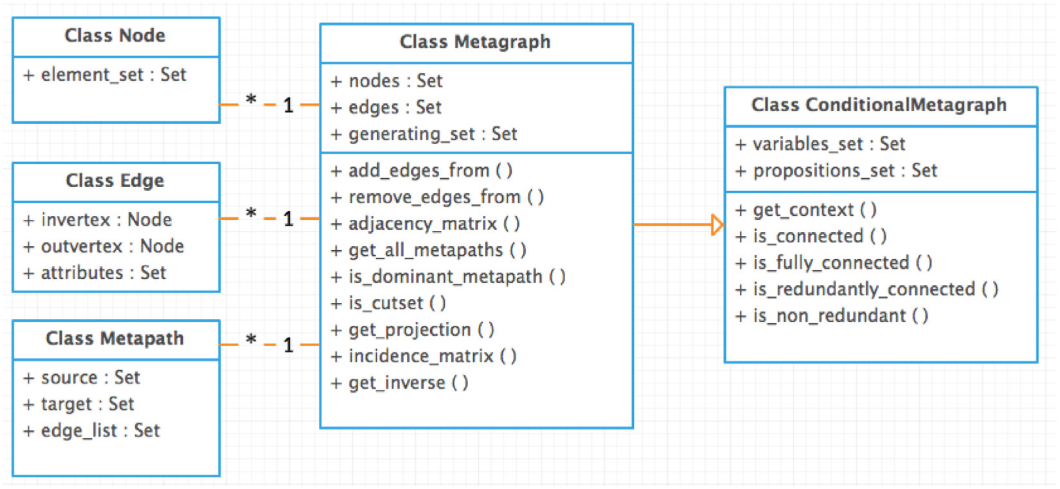


Fig. 2. *MGtoolkit* entity relationship model (*–1 denotes a many-to-one relationship and → denotes an extension).

```
  Edge({’u1’,’u2’,’u3’}, {’r1’,’r2’}, attributes=[’action=permit’]),
  Edge({’u3’,’u4’,’u5’}, {’r2’,’r3’}, attributes=[’action=deny’]),
  Edge({’u2’,’u3’,’u5’,’u6’}, {’r1’,’r2’}, attributes=[’action=permit’])])

# compute redundancies and conflicts
all_metapaths = cm.get_all_metapaths()
for metapath in all_metapaths:
  if cm.has_redundancies(metapath):
      print(’redundancy detected: \%s’\%repr(metapath))
      if cm.has_conflicts(metapath):
     print(’conflict detected: \%s’\%repr(metapath))
```

Listing 2: Partial output from running code in Listing 1.

```
conflict detected: Metapath({ Edge(set([’u1’,’u2’,’u3’,’action=permit’]),
        set([’r1’,’r2’])), Edge({’u3’,’u4’,’u5’}, {’r2’,’r3’}, attributes
       =[’action=deny’])})
```

The code snippet in Listing 1 instantiates the example access-control policy in Fig. 1(a) using *MGtoolkit* and then checks policy consistency. It returns a redundancy and two conflicts–one is shown in Listing 2. The redundancy is due to e_1 and e_3 both enabling access to R_1 from u_2 and u_3. The conflicts stem from e_3 denying access to R_2. More detailed examples based on business policies and workflows can be found on pages 81, 109 and 126 of the metagraph text [1].

4. Impact and challenges

There are many packages available for analysing graphs, *e.g.,* igraph, NetworkX, Gephi [4–6]. These are being increasingly utilised. Metagraphs provide a powerful generalisation of simple graphs and are particularly suitable for modelling business and computer-network policies [1,2].

MGtoolkit is the first publicly available Python API for implementing metagraphs. It serves two key purposes. Firstly, the API allows users to learn about metagraphs in an interactive manner by creating metagraph examples, applying metagraph operations and evaluating the results. The documentation and tutorials associated with the package simplify the learning curve. Secondly, the API is a building block for developing and analysing metagraph-based applications such as decision-support systems. Developers can harness the advantages and power of metagraphs in to their applications by simply importing *MGtoolkit*.

We believe our API is a first step to revisit old questions and tackle new challenges. For instance, in the specification and analysis of computer network policies: current approaches either lack high-level specification capability or formal semantics. *MGtoolkit* is a gateway to harness the best of both of these worlds.

We have used the GitHub open source code hosting and development platform to enable user collaboration.

A key drawback in developing *MGtoolkit* was the fact that the only metagraph text available for reference contained several discrepancies. For instance, the inverse metagraph generation algorithm given in the text failed to replicate the example output provided (Figure 4.9 on page 47 in [1]). Upon clarification with the author, we found that the example was in fact incorrect.

Also several metapath examples given contradicted the definition of a metapath (*e.g.,* metapath M_4 on page 28 in [1]). We strictly adhered to the definition because the formal metagraph properties derived were based on the definition.

5. Conclusions and future work

In this paper, we present *MGtoolkit*: an open-source Python package for implementing metagraphs. The software promotes learning and experimentation with metagraphs and can help analyse business- and computer-network policies alike.

In the future, we are planning several applications based on *MGtoolkit*, one in particular is a tool for the formal analysis of computer-network policies. Additionally, some of the algorithms suggested in [1] are not efficient and we plan to improve on them.

Acknowledgement

This project was supported by the Australian Government through the Australian Research Council Linkage Project LP140100489.

References

[1] Basu A, Blanning RW. Metagraphs and their applications. Springer Science & Business Media; 2007.
[2] Nguyen HX, Pham T, Hoang K, Nguyen DD, Parsonage E. A prototype of policy defined wireless access networks. In: International Telecommunication Networks and Applications Conference. 2016. p. 1–5.
[3] Gushcha A, Haskell library for metagraph data structure; 2017. [Online]. Available: https://github.com/Teaspot-Studio/metagraph.
[4] igraph Steering Committee. Get started with python-igraph; 2006. [Online]. Available: http://igraph.org/python/.
[5] NetworkX Developer Team. High-productivity software for complex networks; 2004. [Online]. Available: https://networkx.github.io/.
[6] Bastian M, Heymann S, Jacomy M, Gephi: An Open Source Software for Exploring and Manipulating Networks; (2009) https://gephi.org/.