The Journal of Open Source Software

BisPy: Bisimulation in Python

Francesco Andreuzzi[1,2]

1 Internation School of Advanced Studies, SISSA, Trieste, Italy **2** Università degli Studi di Trieste

DOI: 10.21105/joss.03519

Software

- Review
- Repository
- Archive

Editor: Matthew Sottile

Reviewers:

- @zoometh
- @mschordan
- @jonjoncardoso

Submitted: 13 July 2021
Published: 21 July 2021

Summary

A binary relation $\mathcal{B}$ on the set V of the nodes of a directed graph is a bisimulation if the following condition is satisfied (Gentilini et al., 2003):

$$(a,b) \in \mathcal{B} \implies \begin{cases} a \to a' & \implies \exists b' \in V \mid (a',b') \in \mathcal{B} \wedge b \to b' \\ b \to b' & \implies \exists a' \in V \mid (a',b') \in \mathcal{B} \wedge a \to a' \end{cases} \tag{1}$$

A *labeling function* $\ell : V \to L$ may be introduced, in which case the graph becomes a *Kripke structure* and the additional condition $(a,b) \in \mathcal{B} \implies \ell(a) = \ell(b)$ must be satisfied.

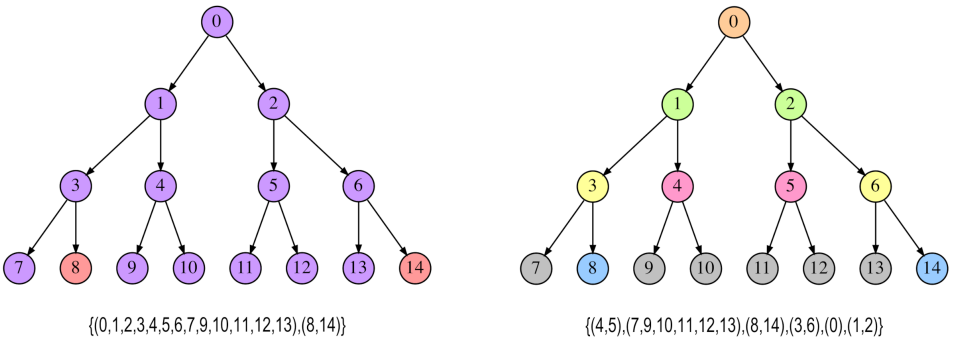

Figure 1: On the left, a balanced tree paired with a labeling function which induces a partition on V of cardinality 2. We represented visually the corresponding maximum bisimulation on the right, computed using `BisPy`.

The notion of *bisimulation* and in particular of *maximum bisimulation* — namely the bisimulation which contains all the other bisimulations on the graph — has applications in modal logic, formal verification and concurrency theory (Kanellakis & Smolka, 1990), and is used for graph reduction as well (Gentilini et al., 2003). The fact that *graphs* may be used to create digital models of a wide span of complex systems makes bisimulation a useful tool in many different cases. For this reason several algorithms for the computation of maximum bisimulation have been studied throughout the years, and it is now widely known that the problem has an $O(|E| \log |V|)$ algorithmic solution, where V is the set of nodes in the graph, and E is the set of edges of the graph.

The first procedure to match this time complexity was *Paige-Tarjan*'s algorithm (1987), whose authors obtained an efficient solution employing an insight from the famous algorithm for the

minimization of finite states automata (Hopcroft, 1971). Even though the problem was formally solved, several decades later a new algorithm was presented (Dovier et al., 2001). *Dovier-Piazza-Policriti*'s algorithm uses extensively the notion of *rank*, which can be defined in an informal way as the distance of a node from the *leafs* (or *sinks*) of the graph, and may be computed — prior the execution of the algorithm — using an $O(|V| + |E|)$ procedure (Sharir, 1981; Tarjan, 1972). The authors claimed that their new method outperformed *Paige-Tarjan*'s algorithm in several cases, even though the time complexity remains the same. Finally we considered an *incremental* algorithm, namely a method which updates the maximum bisimulation after a modification on the graph (in our case, the addition of a new edge). Its time complexity is substantially different than that of from-scratch algorithms, since it depends on how much the maximum bisimulation changes as a consequence of the modification (Saha, 2007). We found also other algorithms to compute the maximum bisimulation, but for what we could see they were slightly variations of the classical ones, or tailored on particular cases.

`BisPy` is a Python package for the computation of maximum bisimulation. It contains the implementation of the algorithms mentioned in the previous paragraph, each of which have been tested and documented. It is interesting to note that the three methods included in the package use a substantially different strategy to reach the result, therefore the performance of different methods may be compared easily on practical applications.

The algorithms have been splitted in several functions rather than being implemented in a monolithic block of code, in order to improve readability and testability. Moreover, this kind of modularity allowed us to reuse functions in multiple algorithms, since several procedures are shared (e.g., `split`, or the computation of rank), and for the same reason we think that the addition of new functionalities would be straightforward since we already have a solid set of common functions implemented.

Example

We present the code which we used to generate the example shown in Figure 1. First of all we import the modules needed to generate the graph (`BisPy` takes `NetworkX` directed graphs in input) and to compute the maximum bisimulation.

```
>>> import networkx as nx
>>> from bispy import compute_maximum_bisimulation
```

After that we generate the graph, which as we mentioned before is a balanced tree with *branching-factor*=2 and *depth*=3. We also create a list of tuples which represents the labeling function which we employed in the example.

```
>>> graph = nx.balanced_tree(2,3, create_using=nx.DiGraph)
>>> labels = [(0,1,2,3,4,5,6,7,9,10,11,12,13),(8,14)]
```

We can now compute the maximum bisimulation of the Kripke structure taken into account as follows:

```
>>> compute_maximum_bisimulation(graph, labels)
[(4,5),(7,9,10,11,12,13),(8,14),(3,6),(0,),(1,2)]
```

The visualization shown above has been drawn using the library `PyGraphviz`. `BisPy` provides the requested output in the form of a list of tuples, each of which contains the labels of all the nodes which are members of an equivalence class of the maximum bisimulation.

Statement of need

To the best of our knowledge `BisPy` is the first Python project to address the problem of bisimulation and to fulfill the needs of an healthy open source project. We found some sparse implementations of *Paige-Tarjan*'s algorithm, however the source code lacked documentation and test cases, and did not seem to be intended for distribution.

We think that our project may be a useful tool to study practical cases for people who are approaching the field — since the notion of bisimulation may be somewhat counterintuitive at first glance — as well as to established researchers, who may use `BisPy` to study improvements on particular types of graphs and to compare new algorithms with the state of the art.

Acknowledgements

We acknowledge the support received from Alberto Casagrande during the preliminar theoretical study of the topic, as well as SISSA mathLab for providing the hardware to perform experiments on large graphs.

References

Dovier, A., Piazza, C., & Policriti, A. (2001). A fast bisimulation algorithm. *International Conference on Computer Aided Verification*, 79–90. https://doi.org/10.1007/3-540-44585-4_8

Gentilini, R., Piazza, C., & Policriti, A. (2003). From bisimulation to simulation: Coarsest partition problems. *Journal of Automated Reasoning*, *31*(1), 73–103. https://doi.org/10.1023/A:1027328830731

Hopcroft, J. (1971). An n log n algorithm for minimizing states in a finite automaton. In *Theory of machines and computations* (pp. 189–196). Elsevier. https://doi.org/10.1016/B978-0-12-417750-5.50022-1

Kanellakis, P. C., & Smolka, S. A. (1990). CCS expressions, finite state processes, and three problems of equivalence. *Information and Computation*, *86*(1), 43–68. https://doi.org/10.1016/0890-5401(90)90025-D

Paige, R., & Tarjan, R. E. (1987). Three partition refinement algorithms. *SIAM Journal on Computing*, *16*(6), 973–989. https://doi.org/10.1137/0216062

Saha, D. (2007). An incremental bisimulation algorithm. *International Conference on Foundations of Software Technology and Theoretical Computer Science*, 204–215. https://doi.org/10.1007/978-3-540-77050-3_17

Sharir, M. (1981). A strong-connectivity algorithm and its applications in data flow analysis. *Computers & Mathematics with Applications*, *7*(1), 67–72. https://doi.org/10.1016/0898-1221(81)90008-0

Tarjan, R. (1972). Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, *1*(2), 146–160. https://doi.org/10.1137/0201010