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:

- @jonjoncardoso
- @mschordan
- @zoometh

Submitted: 13 July 2021
Published: 28 September 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.

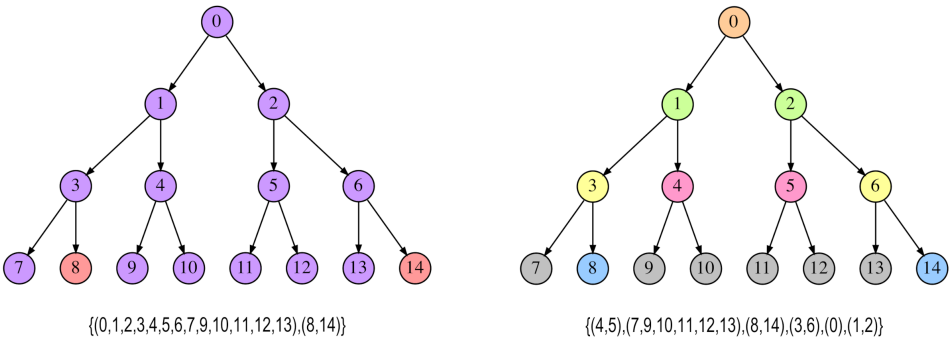

Figure 1: On the left, a balanced tree paired with a labeling function, which induces a partition on V of cardinality 2. We visually represent the corresponding maximum bisimulation on the right, computed using `BisPy`.

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 known that the problem has an $O(|E| \log |V|)$ algorithmic solution (Paige & Tarjan, 1987), where V is the set of nodes in the graph, and E is the set of edges of the graph.

`BisPy` is a Python package for the computation of maximum bisimulation.

Statement of need

To the best of our knowledge, `BisPy` is the first Python project to address the problem presented above, and to meet the objectives of healthy open source software, namely extensive testing, documentation, and intuitive code commenting.

We think that our project may be a useful tool to study practical cases for students approaching the field — since the notion of bisimulation may be somewhat counterintuitive at first glance — as well as for 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.

It is interesting to observe that the package `BisPy`, briefly presented below, contains the implementation of more than one algorithm for the computation of maximum bisimulation, and every algorithm uses a peculiar strategy to obtain the result. For this reason, we think that our package may be useful to assess the performance of different approaches on a particular problem.

`BisPy`

Our package contains the implementation of the following algorithms:

- Paige-Tarjan (1987), which employs an insight from the famous algorithm for the minimization of finite states automata (Hopcroft, 1971);
- Dovier-Piazza-Policriti (2001), which uses the notion of *rank* to optimize the overhead of splitting the initial partition, and can be computed — prior the execution of the algorithm — using an $O(|V| + |E|)$ procedure (Sharir, 1981; Tarjan, 1972);
- Saha (2007), which can be used to update the maximum bisimulation of a graph after the addition of a new edge, and is more efficient than the computation *from scratch* in some cases (the computational complexity depends on how much the maximum bisimulation changes due to the modification).

Our implementations have been tested and documented deeply; moreover we split the algorithms into smaller functions, which we prefer to having a monolithic block of code in order to improve readability and testability. This kind of modularity allows us to reuse functions across multiple algorithms, since several procedures are shared (e.g., `split` is used in all three of the algorithms that we mentioned above, while the computation of rank is carried out only in the last two), and for the same reason we think that the addition of new functionalities would be straightforward since we have already implemented a significant set of common functions.

Example

We present the code that 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 that 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 that are members of an equivalence class of the maximum bisimulation.

Performance

We briefly examine some performance results on two different kinds of graphs:

- *Balanced trees* (Cormen et al., 2009) with variable branching factor r and height h, for which we are going to use the notation $B_T(r, h)$;
- *Erdős-Rényi graphs* (2009), also called *binomial graphs*, whose set E of edges is generated randomly (the cardinality $|E|$ is roughly $p|V|$).

The first experiment involves balanced trees, and consists of the computation of the maximum bisimulation of trees with variable dimensions. The labeling set is the trivial partition of the set V. The results are shown in the left side of Figure 2. The quantity that varies along the x-axis is $|E| \log |V|$, since this allows the presentation of data in a more natural way.

The performance complies with the expected complexity $|E| \log |V|$: for instance our implementation of Dovier-Piazza-Policriti takes about 1.425 seconds to compute the maximum bisimulation on $B_T(3, 10)$, and 12.596 seconds on $B_T(3, 12)$. The value of the ratio $\frac{|E_{B_T(3,12)}| \log |V_{B_T(3,12)}|}{|E_{B_T(3,10)}| \log |V_{B_T(3,10)}|}$ is approximately 10.7, therefore the growth of the time function respects approximately the predicted behavior.

Concerning binomial graphs, we fixed $p = 0.0005$ in order to obtain a graph of some practical interest (as $p \to 1$ the graph becomes complete, as $p \to 0$ also $|E| \to 0$). This time we also consider Saha's incremental algorithm, and we conduct the experiment as follows:

1. Generate a binomial graph with the aforementioned features;
2. Compute the maximum bisimulation using Paige-Tarjan's algorithm;
3. Add a random edge to the graph;
4. Compute the updated maximum bisimulation three times, using the three algorithms taken into account, and verify the time taken by each one.

Since the experiment is not deterministic (the graph and the new edge are generated randomly) we evaluate and visualize the mean time taken by step 4 on a sample of 1000 iterations of steps 1-4.

The knowledge of the old maximum bisimulation is of no interest for non-incremental algorithms. However Saha's algorithm uses this input to reduce the number of steps: the goal of the second experiment is in fact to illustrate this improvement. The results are shown in the right side of Figure 2.

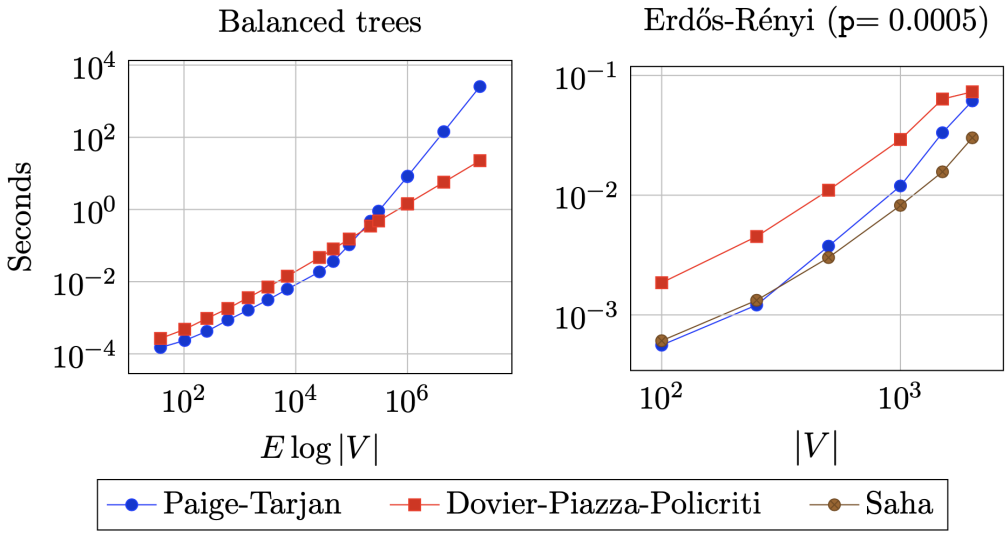

Figure 2: On the left side of the figure, the time taken by our implementations of Paige-Tarjan and Dovier-Piazza-Policriti to compute the maximum bisimulation of balanced trees with variable branching factor and height. On the right side, the time needed to update the maximum bisimulation of a binomial graph after the addition of a random edge (for this experiment we also consider Saha's incremental algorithm).

We ran the experiments on a workstation with operating system *CentOS Linux*, (x86*64), processor Intel(R) Core(TM) i7-4790 CPU (4 cores, 3.60GHz), and 16 GB RAM. Graphs have been generated using functions from the Python package _NetworkX* (Hagberg et al., 2008). We measured time using the Python module *timeit* (Van Rossum & Drake, 2009).

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 used to perform experiments on large graphs.

References

Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). *Introduction to algorithms, 3rd edition*. MIT Press. ISBN: 978-0-262-03384-8

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

Hagberg, A., Swart, P., & S Chult, D. (2008). *Exploring network structure, dynamics, and function using NetworkX*. Los Alamos National Laboratory (LANL), Los Alamos, NM (United States).

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

Van Rossum, G., & Drake, F. L. (2009). *Python 3 reference manual*. CreateSpace. ISBN: 1441412697