automata: A Python package for simulating and manipulating automata

Caleb Evans [1] **and Eliot W. Robson** [2¶]

1 Independent Developer, USA **2** Department of Computer Science, University of Illinois, Urbana, IL, USA ¶ Corresponding author

DOI: 10.21105/joss.05759

Software

- Review
- Repository
- Archive

Editor: Vincent Knight

Reviewers:

- @viech
- @mahfuz05062

Submitted: 19 July 2023
Published: 09 October 2023

Summary

Automata are abstract machines used to represent models of computation, and are a central object of study in theoretical computer science (J. E. Hopcroft et al., 2006). Given an input string of characters over a fixed alphabet, these machines either accept or reject the string. A language corresponding to an automaton is the set of all strings it accepts. Three important families of automata in increasing order of generality are the following:

1. Finite-state automata
2. Pushdown automata
3. Turing machines

The `automata` package facilitates working with these families by allowing simulation of reading input and higher-level manipulation of the corresponding languages using specialized algorithms.

Statement of need

Automata are a core component of both computer science education and research, seeing further theoretical work and applications in a wide variety of areas such as computational biology (Marschall, 2011) and networking (Xu et al., 2016). Consequently, the manipulation of automata with software packages has seen significant attention from researchers in the past. The similarly named Mathematica package `Automata` (Sutner, 2003) implements a number of algorithms for use with finite-state automata, including regular expression conversion and binary set operations. In Java, the Brics package (Møller, 2021) implements similar algorithms, while the JFLAP package (Rodger, 2006) places an emphasis on interactivity and simulation of more general families of automata.

`automata` serves the demand for such a package in the Python software ecosystem, implementing algorithms and allowing for simulation of automata in a manner comparable to the packages described previously. As a popular high-level language, Python enables significant flexibility and ease of use that directly benefits many users. The package includes a comprehensive test suite, support for modern language features (including type annotations), and has a large number of different automata, meeting the demands of users across a wide variety of use cases. In particular, the target audience is both researchers that wish to manipulate automata, and for those in educational contexts to reinforce understanding about how these models of computation function.

The `automata` package

The API of the package is designed to mimic the formal mathematical description of each automaton using built-in Python data structures (such as sets and dicts). This is for ease of use by those that are unfamiliar with these models of computation, while also providing performance suitable for tasks arising in research. In particular, algorithms in the package have been written for tackling performance on large inputs, incorporating optimizations such as only exploring the reachable set of states in the construction of a new finite-state automaton. The package also has native display integration with Jupyter notebooks, enabling easy visualization that allows students to interact with `automata` in an exploratory manner.

Of note are some commonly used and technical algorithms implemented in the package for finite-state automata:

- An optimized version of the Hopcroft-Karp algorithm to determine whether two deterministic finite automata (DFA) are equivalent (Almeida et al., 2010).
- The product construction algorithm for binary set operations (union, intersection, etc.) on the languages corresponding to two input DFAs (Sipser, 2012).
- Thompson's algorithm for converting regular expressions to equivalent nondeterministic finite automata (NFA) (Aho et al., 2006).
- Hopcroft's algorithm for DFA minimization (J. Hopcroft, 1971; Knuutila, 2001).
- A specialized algorithm for directly constructing a state-minimal DFA accepting a given finite language (Mihov & Schulz, 2019).
- A specialized algorithm for directly constructing a minimal DFA recognizing strings containing a given substring (Knuth et al., 1977).

To the authors' knowledge, this is the only Python package implementing all of the automata manipulation algorithms stated above.

`automata` has already been cited in publications (Jeff Erickson & Solomon, 2023), and has seen use in multiple large undergraduate courses in introductory theoretical computer science at the University of Illinois Urbana-Champaign (roughly 2000 students since Fall 2021). In this instance, the package is being used both as part of an autograder utility for finite-state automata created by students, and as an exploratory tool for use by students directly.

Example usage

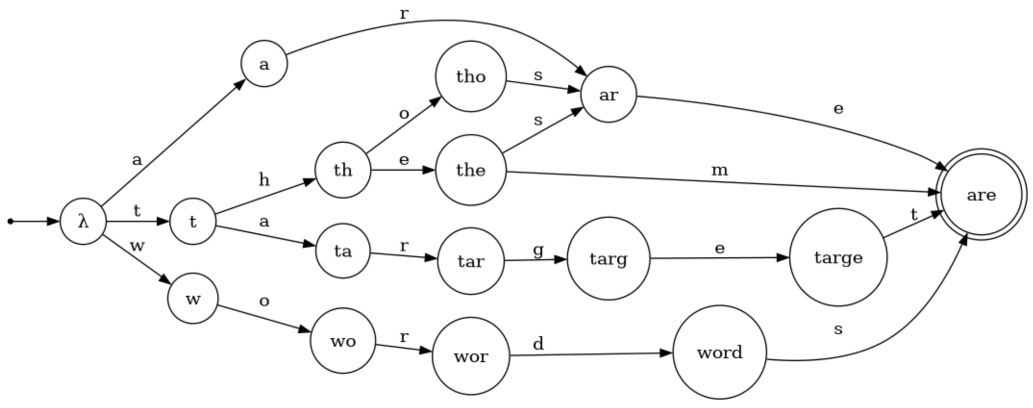

Figure 1: A visualization of `target_words_dfa`. Transitions on characters leading to immediate rejections are omitted.

The following example is inspired by the use case described in (Johnson, 2010). We wish to determine which strings in a given set are within the target edit distance to a reference string.

We will first initialize a DFA corresponding to a fixed set of target words over the alphabet of all lowercase ascii characters.

```python
from automata.fa.dfa import DFA
from automata.fa.nfa import NFA
import string

target_words_dfa = DFA.from_finite_language(
  input_symbols=set(string.ascii_lowercase),
  language={'these', 'are', 'target', 'words', 'them', 'those'},
)
```

A visualization of `target_words_dfa`, generated by the package in a Jupyter notebook, is depicted in Figure 1.

Next, we construct an NFA recognizing all strings within a target edit distance of a fixed reference string, and then immediately convert this to an equivalent DFA. The package provides builtin functions to make this construction easy, and we use the same alphabet as the DFA that was just created.

```python
words_within_edit_distance_dfa = DFA.from_nfa(
  NFA.edit_distance(
    input_symbols=set(string.ascii_lowercase),
    reference_str='they',
    max_edit_distance=2,
  )
)
```

Finally, we take the intersection of the two DFAs we have constructed and read all of the words in the output DFA into a list. The library makes this straightforward and idiomatic.

```python
found_words_dfa = target_words_dfa & words_within_edit_distance_dfa
found_words = list(found_words_dfa)
```

The DFA `found_words_dfa` accepts strings in the intersection of the languages of the DFAs given as input, and `found_words` is a list containing this language. Note the power of this technique is that the DFA `words_within_edit_distance_dfa` has an infinite language, meaning we could not do this same computation just using the builtin sets in Python directly (as they always represent a finite collection), although the syntax used by `automata` is very similar to promote intuition.

Acknowledgements

Thanks (in no particular order) to GitHub users YtvwlD, dengl11, Tagl, lewiuberg, Camilo-MartinezM, abhinavsinha-adrino, EduardoGoulart1, and khoda81 for their invaluable code contributions to this project.

References

Aho, A. V., Lam, M. S., Sethi, R., & Ullman, J. D. (2006). *Compilers: Principles, techniques, and tools (2nd edition)* (pp. 152–155). Addison-Wesley Longman Publishing Co., Inc. ISBN: 0321486811

Almeida, M., Moreira, N., & Reis, R. (2010). Testing the equivalence of regular languages. *Journal of Automata, Languages and Combinatorics*, *15*(1/2), 7–25. https://doi.org/10.25596/jalc-2010-007

Hopcroft, J. (1971). AN n log n ALGORITHM FOR MINIMIZING STATES IN a FINITE AUTOMATON. In Z. Kohavi & A. Paz (Eds.), *Theory of machines and computations* (pp. 189–196). Academic Press. https://doi.org/10.1016/B978-0-12-417750-5.50022-1

Hopcroft, J. E., Motwani, R., & Ullman, J. D. (2006). *Introduction to automata theory, languages, and computation (3rd edition)*. Addison-Wesley Longman Publishing Co., Inc. ISBN: 0321455363

Jeff Erickson, E. W. R., Jason Xia, & Solomon, B. (2023). Auto-graded scaffolding exercises for theoretical computer science. *2023 ASEE Annual Conference & Exposition*. https://peer.asee.org/42347

Johnson, N. (2010). Damn cool algorithms: Levenshtein automata. In *Nick's Blog*. http://blog.notdot.net/2010/07/Damn-Cool-Algorithms-Levenshtein-Automata

Knuth, D. E., Morris, J. H., Jr., & Pratt, V. R. (1977). Fast pattern matching in strings. *SIAM Journal on Computing*, *6*(2), 323–350. https://doi.org/10.1137/0206024

Knuutila, T. (2001). Re-describing an algorithm by Hopcroft. *Theoretical Computer Science*, *250*(1), 333–363. https://doi.org/10.1016/S0304-3975(99)00150-4

Marschall, T. (2011). Construction of minimal deterministic finite automata from biological motifs. *Theoretical Computer Science*, *412*(8), 922–930. https://doi.org/10.1016/j.tcs.2010.12.003

Mihov, S., & Schulz, K. U. (2019). The minimal deterministic finite-state automaton for a finite language. In *Finite-state techniques: Automata, transducers and bimachines* (pp. 253–278). Cambridge University Press. https://doi.org/10.1017/9781108756945.011

Møller, A. (2021). *Dk.brics.automaton – finite-state automata and regular expressions for Java*. https://www.brics.dk/automaton/

Rodger, S. H. (2006). *JFLAP: An interactive formal languages and automata package*. Jones; Bartlett Publishers, Inc. ISBN: 0763738344

Sipser, M. (2012). *Introduction to the Theory of Computation* (pp. 45–47). Cengage Learning. ISBN: 978-1-133-18781-3

Sutner, K. (2003). Automata, a hybrid system for computational automata theory. In J.-M. Champarnaud & D. Maurel (Eds.), *Implementation and application of automata* (pp. 221–227). Springer Berlin Heidelberg. https://doi.org/10.1007/3-540-44977-9_21

Xu, C., Chen, S., Su, J., Yiu, S. M., & Hui, L. C. K. (2016). A survey on regular expression matching for deep packet inspection: Applications, algorithms, and hardware platforms. *IEEE Communications Surveys & Tutorials*, *18*(4), 2991–3029. https://doi.org/10.1109/COMST.2016.2566669