Finite Automata

Release 1.0.0

Jaskirat Singh Maskeen

Jul 13, 2025

CONTENTS:

1 Features **3**

2 Installation **5**

3 Usage **7**

4 References **9**

5 Table of Contents **11**

- 5.1 src.algo package 11
  - 5.1.1 Submodules 11
  - 5.1.2 src.algo.atomicity module 11
  - 5.1.3 src.algo.kamedaweiner module 12
  - 5.1.4 Module contents 15
- 5.2 src.custom_typing package 15
  - 5.2.1 Submodules 15
  - 5.2.2 src.custom_typing.custom_types module 15
  - 5.2.3 Module contents 16
- 5.3 src.fa package 16
  - 5.3.1 Submodules 16
  - 5.3.2 src.fa.dfa module 16
  - 5.3.3 src.fa.nfa module 18
  - 5.3.4 Module contents 19
- 5.4 src.utils package 19
  - 5.4.1 Submodules 19
  - 5.4.2 src.utils.helper module 19
  - 5.4.3 src.utils.visualizer module 20
  - 5.4.4 Module contents 20

6 Indices and tables **21**

Python Module Index **23**

Index **25**

A Python library for working with finite automata.

This library provides tools for creating, manipulating, and minimizing both **Deterministic Finite Automata (DFA)** and **Nondeterministic Finite Automata (NFA)**. It includes an implementation of the **Kameda-Weiner algorithm** for NFA minimization, along with functionalities for conversion, reversal, and equivalence checking of automata. You can also convert any given NFA to a non-atomic or atomic NFA. Additionally, it integrates with *graphviz* to visualize the state diagrams of these automata.

CHAPTER

ONE

FEATURES

- **DFA and NFA Implementation** Create and modify DFAs and NFAs with an intuitive and flexible API.
- **Automata Operations**
  - Convert NFAs to DFAs using subset construction.
  - Minimize DFAs using the table-filling algorithm.
  - Reverse the language of an automaton.
  - Check for isomorphism between two DFAs.
  - Determine if two NFAs are equivalent.
- **Kameda-Weiner NFA Minimization** As far as we know, this is the first clear and Pythonic implementation, with full verbose output of steps.
- **Visualization** Generate clear and readable diagrams of your automata using *graphviz*. For Kameda-Weiner, you can render the steps for minimization as a website or LaTeX document.

CHAPTER TWO

INSTALLATION

To use this library, you'll need Python installed, as well as the *graphviz* and *requests* libraries:

```
pip install graphviz requests
```

You will also need to install the Graphviz software on your system. Refer to the official Graphviz download page for installation instructions based on your operating system.

Once dependencies are in place, you can clone the repository:

```
git clone https://github.com/jsmaskeen/finite-automata.git
```

CHAPTER THREE

USAGE

See the `Examples` folder for detailed usage.

You can also run examples directly in Google Colab:

1. **Atomicity Function Checks**
2. **NFA - DFA Construction Examples**
3. **Kameda-Weiner Minimisation and Enumeration of NFAs**

CHAPTER FOUR

REFERENCES

- Brzozowski, J. A., & Tamm, H. (2013). *Minimal Nondeterministic Finite Automata and Atoms of Regular Languages*. arXiv preprint arXiv:1301.5585. https://arxiv.org/abs/1301.5585
- Brzozowski, J. A., & Tamm, H. (2011). *Theory of Atomata*. arXiv preprint arXiv:1102.3901. https://arxiv.org/abs/1102.3901
- Tsyganov, A. V. (2012). *Local Search Heuristics for NFA State Minimization Problem*. International Journal of Communications, Network and System Sciences, 5(9), 638-643. https://doi.org/10.4236/ijcns.2012.529074
- Kameda, T., & Weiner, P. (1970). *On the State Minimization of Nondeterministic Finite Automata*. IEEE Transactions on Computers, C-19(7), 617-627. https://doi.org/10.1109/T-C.1970.222994

CHAPTER

FIVE

TABLE OF CONTENTS

5.1 src.algo package

5.1.1 Submodules

5.1.2 src.algo.atomicity module

`src.algo.atomicity.`**`check_atomicity`**`(`*nfa*`)`

Check whether the given NFA is atomic.

An NFA is atomic if its reversal, determinized, and minimized form has the same number of states before and after minimization. This property indicates that the NFA's reverse- deterministic structure is already minimal.

Parameters
`nfa` (`NFA`) – The nondeterministic finite automaton to check for atomicity.

Returns
True if the NFA is atomic, False otherwise.

Return type
bool

`src.algo.atomicity.`**`make_atomic_by_reverse_min`**`(`*nfa*`)`

Create an atomic NFA by reversing and minimizing.

This function determinizes and minimizes the input NFA to a DFA, then reverses that DFA to obtain an NFA that is guaranteed to be atomic.

Parameters
`nfa` (`NFA`) – The original nondeterministic finite automaton.

Returns
An atomic NFA obtained by reverse-minimization of the original NFA.

Return type
NFA

`src.algo.atomicity.`**`make_nonminimal`**`(`*nfa*`)`

Construct a new NFA which recognizes the same language as input NFA but this new NFA's subset construction yields a DFA which is ensured to be non-minimal by cloning each state.

This function takes the input NFA and creates a new one with two copies of each original state: one with its original name and one with a "'" suffix. Transitions are updated so that every transition from a state p to q in the original NFA is mirrored to both q and q' in the new NFA.

Parameters
`nfa` (`NFA`) – The source NFA to clone into a non-minimal NFA.

Returns

A new nondeterministic finite automaton with duplicated states and transitions,
ensuring it's DFA (from subset construction) is non-minimal.

Return type
NFA

5.1.3 src.algo.kamedaweiner module

`class src.algo.kamedaweiner.KamedaWeinerMinimize`(*nfa*, *store_progress=False*, *verbose=False*)

Bases: `NFA`

State minimization of an NFA using the Kameda-Weiner algorithm.

Extends NFA to perform the Kameda-Weiner synthesis procedure, which finds a minimum-state NFA equivalent to the given NFA by analyzing the reduced automaton matrix (RAM), identifying prime grids, and applying the intersection rule over minimal covers.

Inherits all standard NFA functionality and adds methods specific to the Kameda-Weiner minimization steps.

`apply_intersection_rule`(*dfa*, *rsm*, *cover*)

Synthesize an NFA from a chosen cover via the intersection rule.

Constructs a new NFA whose states correspond to the grids in the cover, using the DFA transition structure to determine transitions by intersecting preimages.

Parameters

- **dfa** (`DFA`) – The forward-subset DFA of the original NFA.
- **rsm** (`StatesMap`) – The reduced states map.
- **cover** (`List[Tuple[FrozenSet[int], FrozenSet[int]]]`) – The selected minimal cover of prime grids.

Returns
A synthesized NFA corresponding to this cover.

Return type
NFA

`construct_states_map`(*dfa_map*, *dfa_dual_map*)

Build the initial states map (SM) matrix from DFA and dual DFA.

Given the forward subset-construction mapping of the NFA to a DFA and the dual DFA mapping, construct a StatesMap that partitions NFA states into rows and columns and records their intersections.

Parameters

- **dfa_map** (`Dict[int, FrozenSet[STATE]]`) – Mapping from DFA state ID to its corresponding NFA-state subset.
- **dfa_dual_map** (`Dict[int, FrozenSet[STATE]]`) – Mapping from dual-DFA state ID to its corresponding NFA-state subset.

Returns

A dict containing:

- *rows*: list of singleton frozensets for each DFA state
- *cols*: list of singleton frozensets for each dual DFA state

- *matrix*: 2D list where each cell is the intersection (FrozenSet) of the corresponding row and column, or None if empty

Return type
StatesMap

`find_all_minimal_covers`(*sm*, *prime_grids*)

Generate all covers of minimal size from the prime grids.

Yields each distinct minimal cover (list of grids) in increasing order of grid count.

Parameters

- **sm** (`StatesMap`) – The reduced states map.
- **prime_grids** (*List[Tuple[FrozenSet[int], FrozenSet[int]]]*) – Candidate prime grids.

Yields
Iterator[List[Tuple[FrozenSet[int], FrozenSet[int]]]] – Each minimal cover as a list of grid coordinate pairs.

`find_maximal_prime_grids`(*sm*)

Enumerate all maximal prime grids in the reduced states map.

Explores subgrids by breadth-first reduction of rows or columns containing None entries, and collects those grids that are prime and not strictly contained in any other prime grid.

Parameters
sm (`StatesMap`) – The reduced states map.

Returns
List of pairs (rows, cols) each defining a maximal prime grid.

Return type
List[Tuple[FrozenSet[int], FrozenSet[int]]]

`is_cover`(*sm*, *grids*)

Determine if a set of grids covers all 1-entries in the RAM.

A cover is valid if every nonempty cell in the states map matrix lies within at least one of the supplied grids.

Parameters

- **sm** (`StatesMap`) – The reduced states map.
- **grids** (*Tuple[Tuple[FrozenSet[int], FrozenSet[int]], ...]*) – Sequence of grids to test as a cover.

Returns
True if the grids form a complete cover, False otherwise.

Return type
bool

`is_grid_prime`(*sm*, *grid_rows*, *grid_cols*)

Check whether a given grid is prime (cannot be extended).

A grid is prime if all intersections between the specified row set and column set in the states map are non-None, and it is not strictly contained in any larger grid.

Parameters

- **sm** (`StatesMap`) – The reduced states map.

- **grid_rows** (*FrozenSet[int]*) – The set of row indices defining the grid.
- **grid_cols** (*FrozenSet[int]*) – The set of column indices defining the grid.

Returns
True if the grid is prime, False otherwise.

Return type
bool

`merge_cols`(*sm*, *indices_to_merge*)

Merge a set of columns in the states map into a single column.

Combines the specified column indices by unioning their column-blocks and merging their matrix entries component-wise.

Parameters

- **sm** (`StatesMap`) – The current states map.
- **indices_to_merge** (*List[int]*) – List of column indices that should be merged.

Returns
A new states map with those columns merged.

Return type
StatesMap

`merge_rows`(*sm*, *indices_to_merge*)

Merge a set of rows in the states map into a single row.

Combines the specified row indices by unioning their row-blocks and merging their matrix entries component-wise.

Parameters

- **sm** (`StatesMap`) – The current states map.
- **indices_to_merge** (*List[int]*) – List of row indices that should be merged.

Returns
A new states map with those rows merged.

Return type
StatesMap

`reduce_states_map`(*states_map*)

Compute the reduced states map (RSM) by merging equivalent rows/columns.

Iteratively merges rows with identical patterns of nonempty entries, then columns likewise, until no further merges are possible.

Parameters
states_map (`StatesMap`) – The initial or current states map to reduce.

Returns
The reduced map with merged rows/columns.

Return type
StatesMap

`run`(*get_all_nfas=False*)

Execute the full Kameda-Weiner minimization procedure.

1. Convert the NFA to a DFA and its dual.

2. Build and reduce the states map.
3. Find prime grids and enumerate minimal covers.
4. Apply the intersection rule to each cover until a legitimate (language-preserving) NFA is found.

Parameters
`get_all_nfas` (*bool, optional*) – If True, return all candidate NFAs in non decreasing order of number of states with a flag indicating equivalence to the original. If False, return the first valid minimized NFA found.

Returns
The minimized NFA, or a list of (NFA, is_equivalent) pairs.

Return type
NFA or List[Tuple[*NFA*, bool]]

5.1.4 Module contents

5.2 src.custom_typing package

5.2.1 Submodules

5.2.2 src.custom_typing.custom_types module

`class src.custom_typing.custom_types.DeltaIn`

Bases: `DefaultDict[str, Set[str | int]]`

Mapping from input symbols to sets of destination states for an NFA.

This dictionary subclass maps each input symbol (string) to a set of NFA states (STATE) reachable on that symbol from a given source state.

Example

delta_in = DeltaIn() delta_in['a'].add(state1) delta_in['b'].update({state2, state3})

`class src.custom_typing.custom_types.DeltaOut`

Bases: `DefaultDict[str | int, DeltaIn]`

Mapping from source states to their outgoing transitions in an NFA.

This dictionary subclass maps each source state (STATE) to a DeltaIn instance, which in turn maps input symbols to sets of destination states. Together, DeltaOut[src][char] gives the set of states reachable from src on symbol char.

Example

delta_out = DeltaOut() delta_out[src_state]['a'].add(dest_state) # Now dest_state is in delta_out[src_state]['a']

`src.custom_typing.custom_types.STATE`

A type alias for representing either a string or integer state identifier.

alias of `str | int`

`class src.custom_typing.custom_types.StatesMap`

Bases: `TypedDict`

Mapping of DFA state partitions used in the Kameda-Weiner minimization algorithm.

In the context of the Kameda-Weiner algorithm, the states map matrix records how groups of DFA states (rows and columns) intersect during minimization. This structure supports tracking which combined state sets remain distinguishable or mergeable.

cols: List[FrozenSet[int]]

Sequence of frozen sets, each representing a block of DFA state indices corresponding to a column partition in the states map.

matrix: List[List[Optional[FrozenSet[Union[str, int]]]]]

A two-dimensional list with dimensions len(rows) x len(cols). Each cell holds either: - A frozen set of STATE elements indicating the intersection of the respective row's and column's state blocks during minimization. - None if no intersection (i.e., the blocks do not share any states).

rows: List[FrozenSet[int]]

Sequence of frozen sets, each representing a block of DFA state indices corresponding to a row partition in the states map.

5.2.3 Module contents

5.3 src.fa package

5.3.1 Submodules

5.3.2 src.fa.dfa module

class `src.fa.dfa.`**DFA**

Bases: `object`

Deterministic Finite Automaton (DFA) representation and operations.

This class models a DFA with a set of states, an input alphabet (Sigma), a single initial state, a set of final (accepting) states, and a transition function (delta). It provides methods for constructing the DFA, converting to an NFA, computing the language reversal, minimizing the DFA, and checking isomorphism with another DFA.

addFinal(*state*)

Mark a state as a final (accepting) state.

Adds the state to the DFA if it does not already exist.

Parameters
state (*STATE*) – The state to designate as accepting.

addState(*state*)

Add a new state to the DFA.

Parameters
state (*STATE*) – The identifier for the new state, either a string or integer.

addTransition(*src*, *char*, *dest*)

Add a transition for the DFA.

Parameters

- **src** (*STATE*) – The source state for the transition.
- **char** (*str*) – The input symbol that triggers the transition.
- **dest** (*STATE*) – The destination state for the transition.

`is_isomorphic_to`(*other_dfa*)

Check whether this DFA is isomorphic to another DFA.

Two DFAs are isomorphic if there exists a renaming of states that makes their transition structures and accepting behavior identical.

Parameters
`other_dfa` (`DFA`) – The DFA to compare against.

Returns
True if the DFAs are isomorphic, False otherwise.

Return type
bool

`minimal`()

Compute and return the minimal equivalent DFA.

Uses completion (adding a trap state), the table-filling algorithm to distinguish states, and union-find to merge indistinguishable states.

Returns
A minimized DFA equivalent to the original.

Return type
DFA

`reversal`()

Construct the NFA which accepts the reverse of the language accepted by this DFA

Returns
An NFA accepting the reversal of the language of this DFA.

Return type
NFA

`setInitial`(*state*)

Set the initial (start) state of the DFA.

Adds the state to the DFA if it does not already exist.

Parameters
`state` (*`STATE`*) – The state to designate as the initial state.

`setSigma`(*sigma_set*)

Define the input alphabet of the DFA.

Parameters
`sigma_set` (*`Set[str]`*) – A set of characters representing the DFA's input alphabet.

`toNFA`()

Convert this DFA into an equivalent NFA.

Returns
A nondeterministic finite automaton representing the same language as this DFA.

Return type
NFA

5.3.3 src.fa.nfa module

class src.fa.nfa.**NFA**(** (Keyword-only parameters separator (PEP 3102)), States=None, Sigma=None, Initial=None, Final=None, delta=None*)

Bases: `object`

Nondeterministic Finite Automaton (NFA) representation and operations.

Models an NFA with a set of states, an input alphabet (Sigma), an initial subset of states, a set of final (accepting) states, and a transition relation (delta). Provides methods for state and transition management, reversal, conversion to an equivalent DFA (subset construction), and language equivalence checking via DFA minimization.

addFinal(*state*)

Mark a state as final (accepting).

Parameters
state (*STATE*) – The state to designate as accepting.

addState(*state*)

Add a state to the NFA.

Parameters
state (*STATE*) – Identifier of the state to add.

addTransition(*src*, *char*, *dest*)

Add a transition to the NFA.

Parameters

- **src** (*STATE*) – Source state for the transition.
- **char** (*str*) – Input symbol triggering the transition.
- **dest** (*STATE*) – Destination state reached on that symbol.

is_equivalent_to(*other_nfa*)

Check language equivalence with another NFA.

Converts both NFAs to minimized DFAs and tests for isomorphism.

Parameters
other_nfa (NFA) – The other NFA to compare against.

Returns
True if the two NFAs accept the same language, False otherwise.

Return type
bool

reversal()

Compute the reversal of the NFA.

Constructs a new NFA that accepts the reversal of the original language by swapping initial and final states and reversing all transitions.

Returns
The reversed NFA.

Return type
NFA

`setInitial`(*states*)

Set the NFA's initial states.

Converts a single state or an iterable of states into the internal set of initial states and ensures they are included in the overall state set.

Parameters
`states` (*Union[STATE, Iterable[STATE]]*) – A single state or collection of states to designate as initial.

Return type
`None`

`setSigma`(*sigma_set*)

Define the NFA's input alphabet.

Parameters
`sigma_set` (*Set[str]*) – Set of symbols for the input alphabet.

`toDFA`(*get_rev_dfa_map=False*)

Convert this NFA to an equivalent DFA using subset construction.

Parameters
`get_rev_dfa_map` (*bool, optional*) – If True, also return a mapping from DFA state IDs back to the corresponding NFA state subsets.

Returns
The constructed DFA, and optionally the reverse mapping if requested.

Return type
DFA or Tuple[*DFA*, Dict[int, FrozenSet[STATE]]]

5.3.4 Module contents

5.4 src.utils package

5.4.1 Submodules

5.4.2 src.utils.helper module

`class src.utils.helper.Helper`

Bases: `object`

Utility class providing helper algorithms and data structures.

`static dsu`(*parent*)

Construct a disjoint-set union (DSU) structure over the given parent mapping.

The DSU supports efficient union and find operations with path compression on the provided parent dictionary.

Parameters
`parent` (*dict[Any, Any]*) – A mapping where each key is an element and each value is its current parent in the disjoint-set forest. Initially, each element should map to itself.

Returns
An object exposing *find* and *union* methods for managing the disjoint-set structure.

Return type
DSU

`static print_states_map(`*sm*`)`

Helper function to pretty print a StatesMap object.

Parameters

`sm` (`StatesMap`) – The StatesMap to print.

5.4.3 src.utils.visualizer module

`class src.utils.visualizer.Visualizer`

Bases: `object`

`static to_graphviz(`*automaton*`)`

Convert an NFA or DFA into a Graphviz Digraph for visualization.

Parameters

`automaton` (`Union[NFA, DFA]`) – The finite automaton to visualize.

Returns

A Graphviz object representing the automaton.

Return type

Digraph

`static to_latex(`*steps*, *dirname*`)`

Generate a LaTeX file 'story.tex' representing the Kameda-Weiner minimization process steps.

Parameters

- `steps` (`Dict[str, Any]`) – The steps dictionary from KamedaWeinerMinimize.
- `dirname` (`str`) – The directory name where to store the tex file and dot objects.

`static to_website(`*steps*`)`

Convert the steps of Kameda-Weiner minimization process, and display it on the website, with graphviz renders.

5.4.4 Module contents

CHAPTER

SIX

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

S

src.algo, 15
src.algo.atomicity, 11
src.algo.kamedaweiner, 12
src.custom_typing, 16
src.custom_typing.custom_types, 15
src.fa, 19
src.fa.dfa, 16
src.fa.nfa, 18
src.utils, 20
src.utils.helper, 19
src.utils.visualizer, 20

INDEX

A

addFinal() (*src.fa.dfa.DFA method*), 16
addFinal() (*src.fa.nfa.NFA method*), 18
addState() (*src.fa.dfa.DFA method*), 16
addState() (*src.fa.nfa.NFA method*), 18
addTransition() (*src.fa.dfa.DFA method*), 16
addTransition() (*src.fa.nfa.NFA method*), 18
apply_intersection_rule() (*src.algo.kamedaweiner.KamedaWeinerMinimize method*), 12

C

check_atomicity() (*in module src.algo.atomicity*), 11
cols (*src.custom_typing.custom_types.StatesMap attribute*), 16
construct_states_map() (*src.algo.kamedaweiner.KamedaWeinerMinimize method*), 12

D

DeltaIn (*class in src.custom_typing.custom_types*), 15
DeltaOut (*class in src.custom_typing.custom_types*), 15
DFA (*class in src.fa.dfa*), 16
dsu() (*src.utils.helper.Helper static method*), 19

F

find_all_minimal_covers() (*src.algo.kamedaweiner.KamedaWeinerMinimize method*), 13
find_maximal_prime_grids() (*src.algo.kamedaweiner.KamedaWeinerMinimize method*), 13

H

Helper (*class in src.utils.helper*), 19

I

is_cover() (*src.algo.kamedaweiner.KamedaWeinerMinimize method*), 13
is_equivalent_to() (*src.fa.nfa.NFA method*), 18
is_grid_prime() (*src.algo.kamedaweiner.KamedaWeinerMinimize method*), 13
is_isomorphic_to() (*src.fa.dfa.DFA method*), 16

K

KamedaWeinerMinimize (*class in src.algo.kamedaweiner*), 12

M

make_atomic_by_reverse_min() (*in module src.algo.atomicity*), 11
make_nonminimal() (*in module src.algo.atomicity*), 11
matrix (*src.custom_typing.custom_types.StatesMap attribute*), 16
merge_cols() (*src.algo.kamedaweiner.KamedaWeinerMinimize method*), 14
merge_rows() (*src.algo.kamedaweiner.KamedaWeinerMinimize method*), 14
minimal() (*src.fa.dfa.DFA method*), 17
module
    src.algo, 15
    src.algo.atomicity, 11
    src.algo.kamedaweiner, 12
    src.custom_typing, 16
    src.custom_typing.custom_types, 15
    src.fa, 19
    src.fa.dfa, 16
    src.fa.nfa, 18
    src.utils, 20
    src.utils.helper, 19
    src.utils.visualizer, 20

N

NFA (*class in src.fa.nfa*), 18

P

print_states_map() (*src.utils.helper.Helper static method*), 19

R

reduce_states_map() (*src.algo.kamedaweiner.KamedaWeinerMinimize method*), 14

`reversal()` (*src.fa.dfa.DFA method*), 17
`reversal()` (*src.fa.nfa.NFA method*), 18
`rows` (*src.custom_typing.custom_types.StatesMap attribute*), 16
`run()` (*src.algo.kamedaweiner.KamedaWeinerMinimize method*), 14

S

`setInitial()` (*src.fa.dfa.DFA method*), 17
`setInitial()` (*src.fa.nfa.NFA method*), 18
`setSigma()` (*src.fa.dfa.DFA method*), 17
`setSigma()` (*src.fa.nfa.NFA method*), 19
`src.algo`
    `module`, 15
`src.algo.atomicity`
    `module`, 11
`src.algo.kamedaweiner`
    `module`, 12
`src.custom_typing`
    `module`, 16
`src.custom_typing.custom_types`
    `module`, 15
`src.fa`
    `module`, 19
`src.fa.dfa`
    `module`, 16
`src.fa.nfa`
    `module`, 18
`src.utils`
    `module`, 20
`src.utils.helper`
    `module`, 19
`src.utils.visualizer`
    `module`, 20
`STATE` (*in module src.custom_typing.custom_types*), 15
`StatesMap` (*class in src.custom_typing.custom_types*), 15

T

`to_graphviz()` (*src.utils.visualizer.Visualizer static method*), 20
`to_latex()` (*src.utils.visualizer.Visualizer static method*), 20
`to_website()` (*src.utils.visualizer.Visualizer static method*), 20
`toDFA()` (*src.fa.nfa.NFA method*), 19
`toNFA()` (*src.fa.dfa.DFA method*), 17

V

`Visualizer` (*class in src.utils.visualizer*), 20