FAdo Documentation

Release 2.2.0

Rogério Reis & Nelma Moreira

Apr 09, 2024

CONTENTS

CHAPTER

ONE

MODULES

1.1 FAdo.fa

Finite automata manipulation.

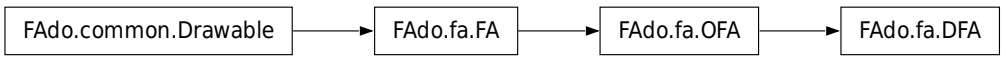
Deterministic and non-deterministic automata manipulation, conversion and evaluation.

class DFA

Class for Deterministic Finite Automata.

Variables

- **States** (*list*) – set of states.
- **sigma** (*set*) – alphabet set.
- **Initial** (*int*) – the initial state index.
- **Final** (*set*) – set of final states indexes.
- **delta** (*dict*) – the transition function.
- **delta_inv** (*dict*) – possible inverse transition map
- **i** (*bool*) – is inverse map computed?



Delta(*state*, *symbol*)

Evaluates the action of a symbol over a state

Parameters

- **state** (*int*) – state index
- **symbol** (*Any*) – symbol

Returns

the action of symbol over state

Return type

int

HKeqP(*other*, *strict=True*)

Tests the DFA's equivalence using Hopcroft and Karp's state equivalence algorithm

Parameters

- **other** –
- **strict** –

Returns
bool

See also:

J. E. Hopcroft and r. M. Karp.A Linear Algorithm for Testing Equivalence of Finite Automata.TR 71–114. U. California. 1971

Attention: The automaton must be complete.

MyhillNerodePartition()

Myhill-Nerode partition, Moore's way

New in version 1.3.5.

Attention: No state should be named with DeadName. This states is removed from the obtained partition.

See also:

F.Bassino, J.David and C.Nicaud, On the Average Complexity of Moores's State Minimization Algorihm, Symposium on Theoretical Aspects of Computer Science

aEquiv()

Computes almost equivalence, used by hyperMinimial

Returns
partition of states

Return type
dict

Note: may be optimized to avoid dupped

addTransition(*sti1*, *sym*, *sti2*)

Adds a new transition from sti1 to sti2 consuming symbol sym.

Parameters

- **sti1** (*int*) – state index of departure
- **sti2** (*int*) – state index of arrival
- **sym** (*Any*) – symbol consumed

Raises
DFAnotNFA – if one tries to add a non-deterministic transition

addTransitionIfNeeded(*sti1: int*, *sym*, *sti2: int*) → int

Adds a new transition from sti1 to sti2 consuming symbol sym, creating states if needed

Parameters

- **sti1** (*int*) – state index of departure
- **sti2** (*int*) – state index of arrival
- **sym** (*Any*) – symbol consumed

Returns
the destination state

Return type
int

Raises
DFAnotNFA – if one tries to add a non-deterministic transition

compat(*s1*, *s2*, *data*)

Tests compatibility between two states.

Parameters

- **data** –
- **s1** (*int*) – state index
- **s2** (*int*) – state index

Return type
bool

complete(*dead='DeaD'*)

Transforms the automata into a complete one. If sigma is empty nothing is done.

Parameters
dead (*str*) – dead state name

Returns
the complete FA

Return type
DFA

Note: Adds a dead state (if necessary) so that any word can be processed with the automata. The new state is named `dead`, so this name should never be used for other purposes.

Attention: The object is modified in place.

Changed in version 1.0.

completeMinimal()

Completes a DFA assuming it is a minimal and avoiding de destruction of its minimality If the automaton is not complete, all the non-final states are checked to see if tey are not already a dead state. Only in the negative case a new (dead) state is added to the automaton.

Return type
DFA

> **Attention:** The object is modified in place. If the alphabet is empty nothing is done

completeP()

Checks if it is a complete FA (if delta is total)

Returns
bool

completeProduct(*other*)

Product structure

Parameters
other (DFA) – the other DFA

Return type
DFA

computeKernel()

The Kernel of a ICDFA is the set of states that accept a non-finite language.

Returns
triple (comp, center , mark) where comp are the strongly connected components, center the set of center states and mark the kernel states

Return type
tuple

concat(*fa2*, *strict=False*)

Concatenation of two DFAs. If DFAs are not complete, they are completed.

Parameters

- **strict** (*bool*) – should alphabets be checked?
- **fa2** (DFA) – the second DFA

Returns
the result of the concatenation

Return type
DFA

Raises
DFAdifferentSigma – if alphabet are not equal

concatI(*fa2*, *strict=False*)

Concatenation of two DFAs.

Parameters

- **fa2** (DFA) – the second DFA
- **strict** (*bool*) – should alphabets be checked?

Returns
the result of the concatenation

Return type
DFA

Raises
DFAdifferentSigma – if alphabet are not equal

New in version 0.9.5.

Note: this is to be used with non-complete DFAs

delTransition(*sti1*, *sym*, *sti2*, *_no_check=False*)

Remove a transition if existing and perform cleanup on the transition function's internal data structure.

Parameters

- **sti1** (*int*) – state index of departure
- **sym** (*Any*) – symbol consumed
- **sti2** (*int*) – state index of arrival
- **_no_check** (*bool*) – use unsecure code?

Note: Unused alphabet symbols will be discarded from sigma.

deleteStates(*del_states*)

Delete given iterable collection of states from the automaton.

Parameters
del_states – collection of state indexes

Note: in-place action

Note: delta function will always be rebuilt, regardless of whether the states list to remove is a suffix, or a sublist, of the automaton's states list.

static deterministicP()

Yes it is deterministic!

Return type
bool

dist()

Evaluate the distinguishability language for a DFA

Return type
DFA

See also:

Cezar Câmpeanu, Nelma Moreira, Rogério Reis: The distinguishability operation on regular languages. NCMA 2014: 85-100

New in version 0.9.8.

distMin()

Evaluates the list of minimal words that distinguish each pair of states

Return type
set of minimal distinguishing words (*FL*)

New in version 0.9.8.

Attention: If the DFA is not minimal, the method loops forever

`distR()`

Evaluate the right distinguishability language for a DFA

Return type
DFA

..seealso:: Cezar Câmpeanu, Nelma Moreira, Rogério Reis:
The distinguishability operation on regular languages. NCMA 2014: 85-100

`distRMin()`

Compute distRMin for DFA

Return type
FL

..seealso:: Cezar Câmpeanu, Nelma Moreira, Rogério Reis:
The distinguishability operation on regular languages. NCMA 2014: 85-100

`distTS()`

Evaluate the two-sided distinguishability language for a DFA

Return type
DFA

..seealso:: Cezar Câmpeanu, Nelma Moreira, Rogério Reis:
The distinguishability operation on regular languages. NCMA 2014: 85-100

`dup()`

Duplicate the basic structure into a new DFA. Basically a copy.deep.

Return type
DFA

`enumDFA(`*n=None*`)`

returns the set of words of words of length up to n accepted by self :param int n: highest length or all words if finite

Return type
list of strings or None

`equal(`*other*`)`

Verify if the two automata are equivalent. Both are verified to be minimum and complete, and then one is matched against the other… Doesn't destroy either dfa…

Parameters
other (DFA) – the other DFA

Return type
bool

evalSymbol(*init*, *sym*)

Returns the state reached from given state through a given symbol.

Parameters

- **init** (*int*) – set of current states indexes
- **sym** (*str*) – symbol to be consumed

Returns
reached state

Return type
int

Raises

- ***DFAsymbolUnknown*** – if symbol not in alphabet
- ***DFAstopped*** – if transition function is not defined for the given input

evalSymbolI(*init*, *sym*)

Returns the state reached from a given state.

Parameters

- **init** (*init*) – current state
- **sym** (*str*) – symbol to be consumed

Returns
reached state or -1

Return type
set of int

Raises
DFAsymbolUnknown – if symbol not in alphabet

New in version 0.9.5.

Note: this is to be used with non-complete DFAs

---

evalSymbolL(*ls*, *sym*)

Returns the set of states reached from a given set of states through a given symbol

Parameters

- **ls** (*set of int*) – set of states indexes
- **sym** (*str*) – symbol to be read

Returns
set of reached states

Return type
set of int

evalSymbolLI(*ls*, *sym*)

Returns the set of states reached from a given set of states through a given symbol

Parameters

- **ls** (*set of int*) – set of current states

- **sym** (*str*) – symbol to be consumed

Returns
set of reached states

Return type
set of int

New in version 0.9.5.

Note: this is to be used with non-complete DFAs

---

evalWord(*wrd*)

Evaluates a word

Parameters
wrd (Word) – word

Returns
final state or None

Return type
int | None

New in version 1.3.3.

evalWordP(*word*, *initial=None*)

Verifies if the DFA recognises a given word

Parameters

- **word** (*list of symbols.*) – word to be recognised
- **initial** (*int*) – starting state index

Return type
bool

finalCompP(*s*)

Verifies if there is a final state in strongly connected component containing `s`.

Parameters
s (*int*) – state

Returns
1 if yes, 0 if no

hasTrapStateP()

Tests if the automaton has a dead trap state

Return type
bool

New in version 1.1.

hxState(*st: int*) → str

A hash value for the transition of a state. The automaton needs to be complete.

Parameters
st (*int*) – the state

Return type
str

hyperMinimal(*strict=False*)

Hyperminization of a minimal DFA

Parameters
strict (*bool*) – if strict=True it first minimizes the DFA

Returns
an hyperminimal DFA

Return type
DFA

See also:

M. Holzer and A. Maletti, An nlogn Algorithm for Hyper-Minimizing a (Minimized) Deterministic Automata, TCS 411(38-39): 3404-3413 (2010)

Note: if strict=False minimality is assumed

---

inDegree(*st*)

Returns the in-degree of a given state in an FA

Parameters
st (*int*) – index of the state

Return type
int

infix()

Returns a dfa that recognizes infix(L(a))

Return type
DFA

initialComp()

Evaluates the connected component starting at the initial state.

Returns
list of state indexes in the component

Return type
list of int

initialP(*state*)

Tests if a state is initial

Parameters
state (*int*) – state index

Return type
bool

initialSet()

The set of initial states

Returns
the set of the initial states

Return type
set

`joinStates`(*lst*)

Merge a list of states.

Parameters
lst (*iterable of state indexes.*) – set of equivalent states

`makeReversible`()

Make a DFA reversible (if possible)

See also:

M.Holzer, s. Jakobi, M. Kutrib 'Minimal Reversible Deterministic Finite Automata'

Return type
DFA

`make_prefix_free`()

Turns a DFA in a prefix-free automaton deleting all outgoing transitions from final states

Return type
DFA

New in version 2.0.3.

`markNonEquivalent`(*s1*, *s2*, *data*)

Mark states with indexes s1 and s2 in given map as non-equivalent states. If any back-effects exist, apply them.

Parameters

- **s1** (*int*) – one state's index
- **s2** (*int*) – the other state's index
- **data** – the matrix relating s1 and s2

`mergeStates`(*f*, *t*)

Merge the first given state into the second. If the first state is an initial state the second becomes the initial state.

Parameters

- **f** (*int*) – index of state to be absorbed
- **t** (*int*) – index of remaining state

Attention: It is up to the caller to remove the disconnected state. This can be achieved with `trim().

`minimal`(*method='minimalMooreSq'*, *complete=True*)

Evaluates the equivalent minimal complete DFA

Parameters

- **method** – method to use in the minimization
- **complete** (*bool*) – should the result be completed?

Returns
equivalent minimal DFA

Return type
DFA

`minimalHopcroft()`

Evaluates the equivalent minimal complete DFA using Hopcroft algorithm

Returns
equivalent minimal DFA

Return type
DFA

See also:

John Hopcroft,An n log{n} algorithm for minimizing states in a finite automaton.The Theory of Machines and Computations.AP. 1971

`minimalHopcroftP()`

Tests if a DFA is minimal

Return type
bool

`minimalIncremental(minimal_test=False, one_cicle=False)`

Minimizes the DFA with an incremental method using the Union-Find algorithm and Memoized non-equivalence intermediate results

Parameters
minimal_test (*bool*) – starts by verifying that the automaton is not minimal?

Returns
equivalent minimal DFA

Return type
DFA

See also:

M. Almeida and N. Moreira and and r. Reis.Incremental DFA minimisation. CIAA 2010. LNCS 6482. pp 39-48. 2010

`minimalIncrementalP()`

Tests if a DFA is minimal

Return type
bool

`minimalMoore()`

Evaluates the equivalent minimal automata with Moore's algorithm

See also:

John E. Hopcroft and Jeffrey D. Ullman, Introduction to Automata Theory, Languages, and Computation, AW, 1979

Returns
minimal complete DFA

Return type
DFA

minimalMooreSq()

Evaluates the equivalent minimal complete DFA using Moore's (quadratic) algorithm

See also:

John E. Hopcroft and Jeffrey D. Ullman, Introduction to Automata Theory, Languages, and Computation, AW, 1979

Returns
equivalent minimal DFA

Return type
DFA

minimalMooreSqP()

Tests if a DFA is minimal using the quadratic version of Moore's algorithm

Return type
bool

minimalNCompleteP()

Tests if a non necessarely complete DFA is minimal, i.e., if the DFA is non-complete, if the minimal complete has only one more state.

Returns
True if not minimal

Return type
bool

Attention: obsolete: use minimalP

minimalNotEquivP()

Tests if the DFA is minimal by computing the set of distinguishable (not equivalent) pairs of states

Return type
bool

minimalP(*method='minimalMooreSq'*)

Tests if the DFA is minimal

Parameters
method – the minimization algorithm to be used

Return type
bool

..note: if DFA non-complete test if complete minimal has one more state

minimalWatson(*test_only=False*)

Evaluates the equivalent minimal complete DFA using Waton's incremental algorithm

Parameters
test_only (*bool*) – is it only to test minimality

Returns
equivalent minimal DFA

Return type
DFA

Raises
DFAnotComplete – if automaton is not complete

..attention::
automaton must be complete

`minimalWatsonP()`

Tests if a DFA is minimal using Watson's incremental algorithm

Return type
bool

`notequal(other)`

Test non equivalence of two DFAs

Parameters
other (DFA) – the other DFA

Return type
bool

`orderedStrConnComponents()`

Topological ordered list of strong components

New in version 1.3.3.

Return type
list

`pairGraph()`

Returns pair graph

Return type
DiGraphVM

See also:

A graph theoretic apeoach to automata minimality. Antonio Restivo and Roberto Vaglica. Theoretical Computer Science, 429 (2012) 282-291. doi:10.1016/j.tcs.2011.12.049 Theoretical Computer Science, 2012 vol. 429 (C) pp. 282-291. http://dx.doi.org/10.1016/j.tcs.2011.12.049

`possibleToReverse()`

Tests if language is reversible

New in version 1.3.3.

`pref()`

Returns a dfa that recognizes pref(L(self))

Return type
DFA

New in version 1.1.

`prefix_free_p()`

Checks is a DFA is prefix-free :rtype: bool

New in version 2.0.3.

print_data(*data*)

Prints table of compatibility (in the context of the minimalization algorithm).

Parameters
data – data to print

product(*other*)

Returns a DFA resulting of the simultaneous execution of two DFA. No final states set.

Note: this is a fast version of the method. The resulting state names are not meaningfull.

Parameters
other – the other DFA

Return type
DFA

productSlow(*other*, *complete=True*)

Returns a DFA resulting of the simultaneous execution of two DFA. No final states set.

Note: this is a slow implementation for those that need meaningfull state names

New in version 1.3.3.

Parameters

- **other** – the other DFA
- **complete** (*bool*) – evaluate product as a complete DFA

Return type
DFA

reorder(*dicti*)

Reorders states according to given dictionary. Given a dictionary (not necessarily complete)... reorders states accordingly.

Parameters
dicti (*dict*) – reorder dictionary

reverseTransitions(*rev*)

Evaluate reverse transition function.

Parameters
rev (DFA) – DFA in which the reverse function will be stored

reversibleP()

Test if an automaton is reversible

Return type
bool

sMonoid()

Evaluation of the syntactic monoid of a DFA

Returns
the semigroup

Return type
SSemiGroup

sSemigroup()

Evaluation of the syntactic semigroup of a DFA

Returns
the semigroup

Return type
SSemiGroup

shuffle(*other*, *strict=False*)

CShuffle of two languages: L1 W L2

Parameters

- **other** (DFA) – second automaton
- **strict** (*bool*) – should the alphabets be necessary equal?

Return type
DFA

See also:

C. Câmpeanu, K. Salomaa and s. Yu, *Tight lower bound for the state complexity of CShuffle of regular languages.* J. Autom. Lang. Comb. 7 (2002) 303–310.

simDiff(*other*)

Symetrical difference

Parameters
other –

Returns

sop(*other*)

Strange operation

Parameters
other (DFA) – the other automaton

Return type
DFA

See also:

Nelma Moreira, Giovanni Pighizzini, and Rogério Reis. Universal disjunctive concatenation and star. In Jeffrey Shallit and Alexander Okhotin, editors, Proceedings of the 17th Int. Workshop on Descriptional Complexity of Formal Systems (DCFS15), number 9118 in LNCS, pages 197–208. Springer, 2015.

New in version 1.2b2.

star(*flag=False*)

Star of a DFA. If the DFA is not complete, it is completed.

..versionchanged: 0.9.6

Parameters
flag (*bool*) – plus instead of star

Returns
the result of the star

Return type
DFA

`starI`()

Star of an incomplete DFA.

Returns
the Kleene closure DFA

Return type
DFA

`stateChildren`(*state*, *strict=False*)

Set of children of a state

Parameters

- **strict** (*bool*) – if not strict a state is never its own child even if a self loop is in place
- **state** (*int*) – state id queried

Returns
map children -> multiplicity

Return type
dictionary

`stronglyConnectedComponents`()

Dummy method that uses the NFA conterpart

New in version 1.3.3.

Return type
list

`subword`()

A dfa that recognizes subword(L(self))

Return type
DFA

New in version 1.1.

`succintTransitions`()

Collects the transition information in a compact way suitable for graphical representation.

Returns
list of tupples

Return type
list

New in version 0.9.8.

`suff`()

Returns a dfa that recognizes suff(L(self))

Return type
DFA

New in version 0.9.8.

syncPower()

Evaluates the Power automata for the action of each symbol

Returns
The Power automata being the set of all states the initial state and all singleton states final

Return type
DFA

toADFA()

Try to convert DFA to ADFA

Returns
the same automaton as a ADFA

Return type
ADFA

Raises
notAcyclic – if this is not an acyclic DFA

New in version 1.2.

Changed in version 1.2.1.

toDFA()

Dummy function. It is already a DFA

Returns
a self deep copy

Return type
DFA

toNFA()

Migrates a DFA to a NFA as dup()

Returns
DFA seen as new NFA

Return type
NFA

transitions()

Iterator over transitions :rtype: symbol, int

transitionsA()

Iterator over transitions :rtype: symbol, int

uniqueRepr()

Normalise unique string for the string icdfa's representation. .. seealso:: TCS 387(2):93-102, 2007 https://www.dcc.fc.up.pt/~nam/publica/tcsamr06.pdf

Returns
normalised representation

Return type
list

Raises
DFAnotComplete – if DFA is not complete

universalP(*minimal=False*)

Checks if the automaton is universal through minimisation

Parameters
minimal (*bool*) – is the automaton already minimal?

Return type
bool

unmark()

Unmarked NFA that corresponds to a marked DFA: in which each alfabetic symbol is a tuple (symbol, index)

Returns
a NFA

Return type
NFA

usefulStates(*initial_states=None*)

Set of states reacheable from the given initial state(s) that have a path to a final state.

Parameters
initial_states (*iterable of int*) – starting states

Returns
set of state indexes

Return type
set of int

witness()

Witness of non emptyness

Returns
word

Return type
str

witnessDiff(*other*)

Returns a witness for the difference of two DFAs and:

0	if the witness belongs to the **other** language
1	if the witness belongs to the **self** language

Parameters
other (DFA) – the other DFA

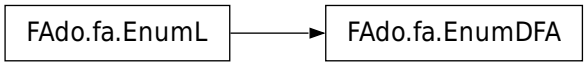
Returns
a witness word

Return type
list of symbols

Raises
DFAequivalent – if automata are equivalent

class EnumDFA(*aut*, *store=False*)

Class for enumerating languages defined by DFAs



fillStack(*w*)

Computes S_1,...,S_n-1 where S_i is the set of (n-i)-complete states reachable from S_i-1

Parameters
w – word

initStack()

Initializes the stack with initial states

minWordT(*n*)

Computes for each state the minimal word of length i<n accepted by the automaton. Stores the values in tmin

Parameters
n (*int*) – length of the word

Note: Makinen algorithm for DFAs

---

nextWord(*w*)

Given an word, returns next word on the nth cross-section of L(aut) according to the radix order

Parameters
w (*str*) – word

Return type
str

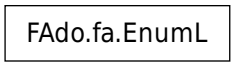
class EnumL(*aut*, *store=False*)

Class for enumerate FA languages

See: Efficient enumeration of words in regular languages, M. Ackerman and J. Shallit, Theor. Comput. Sci. 410, 37, pp 3461-3470. 2009. http://dx.doi.org/10.1016/j.tcs.2009.03.018

Variables

- **aut** (FA) – Automaton of the language
- **tmin** (*dict*) – table for minimal words for each s in aut.States
- **Words** (*list*) – list of words (if stored)
- **sigma** (*list*) – alphabet
- **stack** (*deque*) –



New in version 0.9.8.

enum(*m*)

Enumerates the first m words of L(A) according to the lexicographic order if there are at least m words. Otherwise, enumerates all words accepted by A.

Parameters
m (*int*) – max number of words

enumCrossSection(*n*)

Enumerates the nth cross-section of L(A)

Parameters
n (*int*) – nonnegative integer

abstract fillStack(*w*)

Abstract method :param str w: :type w: str

iCompleteP(*i*, *q*)

Tests if state q is i-complete

Parameters

- **i** (*int*) – int
- **q** (*int*) – state index

abstract initStack()

Abstract method

minWord(*m*)

Computes the minimal word of length m accepted by the automaton :param m: :type m: int

abstract minWordT(*n*)

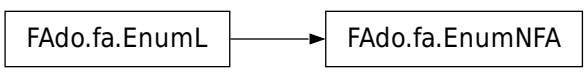
Abstract method :param int n: :type n: int

abstract nextWord(*w*)

Abstract method :param w: :type w: str

class EnumNFA(*aut*, *store=False*)

Class for enumerating languages defined by NFAs



fillStack(*w*)

Computes S_1,...,S_n-1 where S_i is the set of (n-i)-complete states reachable from S_i-1

Parameters
w – word

initStack()

Initializes the stack with initial states

minWordT(*n*)

Computes for each state the minimal word of length i <= n accepted by the automaton. Stores the values in tmin.

Parameters
n (*int*) – length of the word

nextWord(*w: str*)

Given an word, returns next word in the the nth cross-section of L(aut) according to the radix order

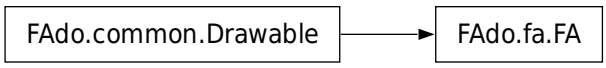
Parameters
w (*str*) – word

class FA

Base class for Finite Automata.
This is just an abstract class. **Not to be used directly!!**

Variables

- **States** (*list*) – set of states.
- **sigma** (*set*) – alphabet set.
- **Initial** (*int*) – the initial state index.
- **Final** (*set*) – set of final states indexes.
- **delta** (*dict*) – the transition function.



addFinal(*stateindex*)

A new state is added to the already defined set of final states.

Parameters
stateindex (*int*) – index of the new final state.

addSigma(*sym*)

Adds a new symbol to the alphabet.

Parameters
sym (*str*) – symbol to be added

Raises
DFAepsilonRedefinition – if sym is Epsilon

Note:

- There is no problem with duplicate symbols because sigma is a Set.
- No symbol Epsilon can be added.

---

addState(*name=None*) → int

Adds a new state to an FA. If no name is given a new name is created.

Parameters
name (Object, optional) – Name of the state to be added.

Returns
Current number of states (the new state index).

Return type
int

Raises
DuplicateName – if a state with that name already exists

changeSigma(*subst: dict*)

Change the alphabet of an automaton by means of a sunstitution subst

Parameters
subst (*dict*) – substitution

Raises
FASiseMismatch – if substitution has a size different from existing alphabet

conjunction(*other*)

A simple literate invocation of __and__

Parameters
other (FA) – right-hand operand.

Returns
Intersection of self and other.

Return type
FA

New in version 0.9.6.

countTransitions()

Evaluates the size of FA transitionwise

Returns
the number of transitions

Return type
int

Changed in version 1.0.

delFinal(*st*)

Deletes a state from the final states list

Parameters
st (*int*) – state to be marked as not final.

delFinals()

Deletes all the information about final states.

disj(*other*)

Another simple literate invocation of __or__

Parameters
other (FA) – the other FA.

Returns
Union of self and other.

Return type
FA

New in version 0.9.6.

disjunction(*other*)

A simple literate invocation of __or__

Parameters
other (FA) – the other FA

Returns
Union of self and other.

Return type
FA

dotDrawState(*sti: int*, *sep='\n'*, *_strict=False*, *_maxlblsz=6*)

Draw a state in dot format

Parameters

- **sti** (*int*) – index of the state.
- **sep** (str, optional) – separator.
- **_maxlblsz** (int, optional) – max size of labels before getting removed
- **_strict** (bool, optional) – use limitations of label size

Returns
string to be added to the dot file.

Return type
str

static dotDrawTransition(*st1*, *label*, *st2*, *sep='\n'*)

Draw a transition in dot format

Parameters

- **st1** (*str*) – departing state
- **label** (*str*) – label
- **st2** (*str*) – arriving state
- **sep** (*str*) – separator

Return type
str

dotFormat(*size='20,20'*, *filename=None*, *direction='LR'*, *strict=False*, *maxlblsz=6*, *sep='\n'*) → str

A dot representation

Parameters

- **direction** (*str*) – direction of drawing - "LR" or "RL"
- **size** (*str*) – size of image
- **filename** (*str*) – output file name
- **sep** (*str*) – line separator
- **maxlblsz** (*int*) – max size of labels before getting removed
- **strict** (*bool*) – use limitations of label sizes

Returns
the dot representation

Return type
str

New in version 0.9.6.

Changed in version 1.2.1.

eliminateDeadName()

Eliminates dead state name (common.DeadName) renaming the state

Returns
self

Return type
DFA

Attention: works inplace

New in version 1.2.

eliminateStout(*st*)

Eliminate all transitions outgoing from a given state

Parameters
st (*int*) – the state index to lose all outgoing transitions

Attention: performs in place alteration of the automata

New in version 0.9.6.

equivalentP(*other*)

Test equivalence between automata

Parameters
other (FA) – the other automata

Return type
bool

New in version 0.9.6.

abstract evalSymbol(*stil*, *sym*)

Evaluation of a single symbol

finalP(*state: int*) → bool

Tests if a state is final

Parameters
state (*int*) – state index.

Returns
is the state final?

Return type
bool

finalsP(*states: set*) → bool

Tests if al the states in a set are final

Parameters
states (*set*) – set of state indexes.

Returns
are all the states final?

Return type
bool

New in version 1.0.

hasStateIndexP(*st: int*) → bool

Checks if a state index pertains to an FA

Parameters
st (*int*) – index of the state.

Return type
bool

images(*sti: int*, *c*)

The set of images of a state by a symbol

Parameters

- **sti** (*int*) – state
- **c** (*object*) – symbol

Return type
iterable

indexList(*lstn*)

Converts a list of stateNames into a set of stateIndexes.

Parameters
lstn (*list*) – list of names

Returns
the list of state indexes

Return type
set

Raises
DFAstateUnknown – if a state name is unknown

initialP(*state: int*) → bool

Tests if a state is initial

Parameters
state – state index

Returns
is the state initial?

Return type
bool

initialSet()

The set of initial states

Returns
set of States.

Return type
set

inputS(*i*) → set[str]

Input labels coming out of state i

Parameters
i (*int*) – state

Returns
set of input labels

Return type
set of str

New in version 1.0.

noBlankNames()

Eliminates blank names

Returns
self

Return type
FA

Attention: in place transformation

plus()

Plus of a FA (star without the adding of epsilon)

New in version 0.9.6.

renameState(*st*, *name*)

Rename a given state.

Parameters

- **st** (*int*) – state index.
- **name** (*object*) – name.

Returns
self.

Return type
FA

Note: Deals gracefully both with int and str names in the case of name collision.

Attention: the object is modified in place

renameStates(*name_list=None*)

Renames all states using a new list of names.

Parameters
name_list (*list*) – list of new names.

Returns
self.

Return type
FA

Raises
DFAerror – if provided list is too short.

Note: If no list of names is given, state indexes are used.

Attention: the object is modified in place

reversal()

Returns a NFA that recognizes the reversal of the language

Returns
NFA recognizing reversal language

Return type
NFA

same_nullability(*s1: int*, *s2: int*) → bool

Tests if this two states have the same nullability

Parameters
- **s1** (*int*) – state index.
- **s2** (*int*) – state index.

Returns
have the states the same nullability?

Return type
bool

setFinal(*statelist*)

Sets the final states of the FA

Parameters
statelist (*int* | *list* | *set*) – a list (or set) of final states indexes.

Caution: Erases any previous definition of the final state set.

setInitial(*stateindex*)

Sets the initial state of a FA

Parameters
stateindex (*int*) – index of the initial state.

setSigma(*symbol_set*)

Defines the alphabet for the FA.

Parameters
symbol_set (*list* | *set*) – alphabet symbols

stateAlphabet(*sti: int*) → list

Active alphabet for this state

Parameters
sti (*int*) – state

Return type
list

stateIndex(*name*, *auto_create=False*)

Index of given state name.

Parameters

- **name** (*object*) – name of the state.
- **auto_create** (bool, optional) – flag to create state if not already done.

Returns
state index

Return type
int

Raises
DFAstateUnknown – if the state name is unknown and autoCreate==False

Note: Replaces stateName

Note: If the state name is not known and flag is set creates it on the fly

New in version 1.0.

stateName(*name*, *auto_create=False*)

Index of given state name.

Parameters

- **name** (*object*) – name of the state
- **auto_create** (bool, optional) – flag to create state if not already done

Returns
state index

Return type
int

Raises
DFAstateUnknown – if the state name is unknown and autoCreate==False

Deprecated since version 1.0: Use: *stateIndex()* instead

Deprecated since version 1.0: Use the stateIndex() method instead

abstract succintTransitions()

Collapsed transitions

union(*other*)

A simple literate invocation of __or__

Parameters
other (FA) – right-hand operand.

Returns
Union of self and other.

Return type
FA

words(*stringo=True*)

Lexicographical word generator

Parameters
stringo (bool, optional) – are words strings? Default is True.

Yields
Word – the next word generated.

Attention: Does not generate the empty word

New in version 0.9.8.

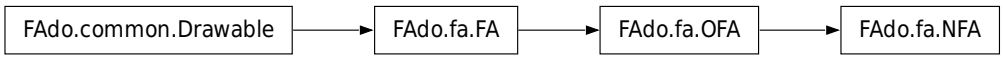
class NFA

Class for Non-deterministic Finite Automata (epsilon-transitions allowed).

Variables

- **States** (*list*) – set of states.
- **sigma** (*set*) – alphabet set.
- **Initial** (*set*) – initial state indexes.
- **Final** (*set*) – set of final states indexes.

- **delta** (*dict*) – the transition function.



HKeqP(*other*, *strict=True*)

Test NFA equivalence with extended Hopcroft-Karp method

Parameters

- **other** (NFA) –
- **strict** (*bool*) – if True checks for same alphabets

Return type
bool

See also:

J. E. Hopcroft and r. M. Karp. A Linear Algorithm for Testing Equivalence of Finite Automata.TR 71–114. U. California. 1971

addEpsilonLoops()

Add epsilon loops to every state

Attention: in-place modification

New in version 1.0.

addInitial(*stateindex*)

Add a new state to the set of initial states.

Parameters
stateindex (*int*) – index of new initial state

addTransition(*sti1*, *sym*, *sti2*)

Adds a new transition. Transition is from `sti1` to `sti2` consuming symbol `sym`. `sti2` is a unique state, not a set of them.

Parameters

- **sti1** (*int*) – state index of departure
- **sti2** (*int*) – state index of arrival
- **sym** (*str*) – symbol consumed

addTransitionQ(*srci*, *dest*, *symb*, *qfuture*, *qpast*)

Add transition to the new transducer instance.

Parameters

- **qpast** (*set*) – past queue
- **qfuture** (*set*) – future queue

- **symb** – symbol
- **dest** (*int*) – destination state
- **srci** (*int*) – source state

New in version 1.0.

addTransitionStar(*sti1*, *sti2*, *exception=()*)

Adds a new transition from sti1 to sti2 consuming any symbol

Parameters

- **sti1** (*int*) – state index of departure
- **sti2** (*int*) – state index of arrival
- **exception** (*list*) – letters to excluded from the pattern

New in version 2.1.

autobisimulation()

Largest right invariant equivalence between states of the NFA

Returns
Incomplete equivalence relation (transitivity, and reflexivity not calculated) as a set of unordered pairs of states

Return type
set

See also:

L. Ilie and S. Yu, Follow automata Inf. Comput. 186 - 1, pp 140-162, 2003

autobisimulation2() → list

Alternative space-efficient definition of NFA.autobisimulation.

Returns
Incomplete equivalence relation (reflexivity, symmetry, and transitivity not calculated) as a set of pairs of states

Return type
list

closeEpsilon(*st*)

Add all non epsilon transitions from the states in the epsilon closure of given state to given state.

Parameters
st (*int*) – state index

Attention: in-place modification

computeFollowNames() → list

Computes the follow set to use in names

Return type
list

concat(*other*, *middle='middle'*)

Concatenation of NFA

Parameters

- **middle** (*str*) – glue state name
- **other** (FA) – the other NFA

Returns
the result of the concatenation

Return type
NFA

countTransitions() → int

Count the number of transitions of a NFA

Return type
int

delTransition(*sti1*, *sym*, *sti2*, *_no_check=False*)

Remove a transition if existing and perform cleanup on the transition function's internal data structure.

Parameters

- **sti1** (*int*) – state index of departure
- **sti2** (*int*) – state index of arrival
- **sym** – symbol consumed
- **_no_check** (*bool*) – dismiss secure code

Note: unused alphabet symbols will be discarded from sigma.

deleteStates(*del_states*)

Delete given iterable collection of states from the automaton.

Parameters
del_states (*set* | *list*) – collection of int representing states

Note: delta function will always be rebuilt, regardless of whether the states list to remove is a suffix, or a sublist, of the automaton's states list.

detSet(*generic=False*)

Computes the determination uppon a followFromPosition result

Return type
NFA

deterministicP()

Verify whether this NFA is actually deterministic

Return type
bool

dotFormat(*size='20,20'*, *filename=None*, *direction='LR'*, *strict=False*, *maxlblsz=6*, *sep='\n'*) → str

A dot representation

Parameters

- **direction** (*str*) – direction of drawing - "LR" or "RL"
- **size** (*str*) – size of image
- **filename** (*str*) – output file name
- **sep** (*str*) – line separator
- **maxlblsz** (*int*) – max size of labels before getting removed
- **strict** (*bool*) – use limitations of label sizes

Returns
the dot representation

Return type
str

New in version 0.9.6.

Changed in version 1.2.1.

dup()

Duplicate the basic structure into a new NFA. Basically a copy.deep.

Return type
NFA

elimEpsilon()

Eliminate epsilon-transitions from this automaton.

:rtype : NFA

Attention: performs in place modification of automaton

Changed in version 1.1.1.

eliminateEpsilonTransitions()

Eliminates all epslilon-transitions with no state addition

Attention: in-place modification

eliminateTSymbol(*symbol*)

Delete all trasitions through a given symbol

Parameters
symbol (*str*) – the symbol to be excluded from delta

Attention: in-place modification

New in version 0.9.6.

enumNFA(*n=None*)

The set of words of length up to n accepted by self

Parameters
n (*int*) – highest lenght or all words if finite

Returns
list of strings or None

Return type
list

epsilonClosure(*st*)

Returns the set of states epsilon-connected to from given state or set of states.

Parameters
st (*int* | *set*) – state index or set of state indexes

Returns
the list of state indexes epsilon connected to `st`

Return type
set

Attention: `st` must exist beforehand.

epsilonP()

Whether this NFA has epsilon-transitions

Return type
bool

epsilonPaths(*start: int*, *end: int*) → set[int]

All states in all paths (DFS) through empty words from a given starting state to a given ending state.

Parameters

- **start** (*int*) – start state
- **end** (*int*) – end state

Returns
states in epsilon paths from start to end

Return type
set

equivReduced(*equiv_classes: UnionFind*)

Equivalent NFA reduced according to given equivalence classes.

Parameters
equiv_classes (*UnionFind*) – Equivalence classes

Returns
Equivalent NFA

Return type
NFA

evalSymbol(*stil*, *sym*)

Set of states reacheable from given states through given symbol and epsilon closure.

Parameters

- **stil** (*set* / *list*) – set of current states
- **sym** (*str*) – symbol to be consumed

Returns
set of reached state indexes

Return type
set

Raises
DFAsymbolUnknown – if symbol is not in alphabet

evalWordP(*word*)

Verify if the NFA recognises given word.

Parameters
word (*str*) – word to be recognised

Return type
bool

finalCompP(*s*)

Verify whether there is a final state in strongly connected component containing given state.

Parameters
s (*int*) – state index

Return type
bool

followFromPosition()

computes follow automaton from a Position automaton

Return type
NFA

half()

Half operation

Return type
NFA

New in version 0.9.6.

hasTransitionP(*state*, *symbol=None*, *target=None*)

Whether there's a transition from given state, optionally through given symbol, and optionally to a specific target.

Parameters

- **state** (*int*) – source state
- **symbol** (*str*) – (optional) transition symbol
- **target** (*int*) – (optional) target state

Returns
if there is a transition

Return type
bool

`homogeneousFinalityP()` → bool

Tests if states have incoming transitions froms states with different finalities

Return type
bool

`homogenousP(x)`

Whether this NFA is homogenous; that is, for all states, whether all incoming transitions to that state are through the same symbol.

Parameters
x – dummy parameter to agree with the method in DFAr

Return type
bool

`initialComp()`

Evaluate the connected component starting at the initial state.

Returns
list of state indexes in the component

Return type
list of int

`lEquivNFA()`

Equivalent NFA obtained from merging equivalent states from autobisimulation of this NFA's reversal.

Return type
NFA

Note: returns copy of self if autobisimulation renders no equivalent states.

`lrEquivNFA()`

Equivalent NFA obtained from merging equivalent states from autobisimulation of this NFA, and from autobisimulation of its reversal; i.e., merges all states that are equivalent w.r.t. the largest right invariant and largest left invariant equivalence relations.

Return type
NFA

Note: returns copy of self if autobisimulations render no equivalent states.

`minimal()`

Evaluates the equivalent minimal DFA

Returns
equivalent minimal DFA

Return type
DFA

`minimalDFA()`

Evaluates the equivalent minimal complete DFA

Returns
equivalent minimal DFA

Return type
DFA

product(*other*)

Returns a NFA (skeletom) resulting of the simultaneous execution of two DFA.

Parameters
other (NFA) – the other automata

Return type
NFA

Raises
NFAerror – if any argument has epsilon-transitions

Note: No final states are set.

Attention:

- operands cannot have epsilon-transitions
- the name EmptySet is used in a unique special state name
- the method uses 3 internal functions for simplicity of code (really!)

rEquivNFA()

Equivalent NFA obtained from merging equivalent states from autobisimulation of this NFA.

Return type
NFA

Note: returns copy of self if autobisimulation renders no equivalent states.

renameStatesFromPosition()

Rename states of a Glushkov automaton using the positions of the marked RE

Return type
NFA

reorder(*dicti*)

Reorder states indexes according to given dictionary.

Parameters
dicti (*dict*) – state name reorder

Attention: in-place modification

Note: dictionary does not have to be complete

reversal()

Returns a NFA that recognizes the reversal of the language

Returns
NFA recognizing reversal language

Return type
NFA

reverseTransitions(*rev*)

Evaluate reverse transition function.

Parameters
rev (NFA) – NFA in which the reverse function will be stored

setInitial(*statelist*)

Sets the initial states of an NFA

Parameters
statelist (*set* | *list* | *int*) – an iterable of initial state indexes

shuffle(*other*)

Shuffle of a NFA

Parameters
other (FA) – an FA

Returns
the resulting NFA

Return type
NFA

star(*flag=False*)

Kleene star of a NFA

Parameters
flag (*bool*) – plus instead of star?

Returns
the resulting NFA

Return type
NFA

stateChildren(*state: int*, *strict=False*) → set[int]

Set of children of a state

Parameters

- **state** (*int*) – state id queried
- **strict** (*bool*) – if not strict a state is never its own child even if a self loop is in place

Returns
children states

Return type
set

stronglyConnectedComponents()

Strong components

Return type
list

New in version 1.0.

subword()

NFA that recognizes subword(L(self))

Return type
NFA

succintTransitions()

Collects the transition information in a compact way suitable for graphical representation. :rtype: list

toDFA()

Construct a DFA equivalent to this NFA, by the subset construction method.

Return type
DFA

Note: valid to epsilon-NFA

---

toNFA()

Dummy identity function

Return type
NFA

toNFAr()

NFA with the reverse mapping of the delta function.

Returns
shallow copy with reverse delta function added

Return type
NFAr

uniqueRepr() → tuple

Dummy representation. Used DFA.uniqueRepr()

Return type
tuple

universalP()

Whether this NFA is universal (accetps all words)

usefulStates(*initial_states=None*)

Set of states reacheable from the given initial state(s) that have a path to a final state.

Parameters
initial_states (set) – set of initial states

Returns
set of state indexes

Return type
set

`witness()`

Witness of non emptiness

Returns
word

Return type
str

`wordImage(word, ist=None)`

Evaluates the set of states reached consuming given word

Parameters

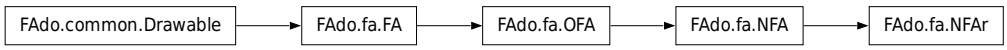
- **word** – the word
- **ist** (*int*) – starting state index (or set of)

Returns
the set of ending states

Return type
set of int

class NFAr

Class for Non-deterministic Finite Automata with reverse delta function added by construction. Includes efficient methods for merging states.



`addTransition(sti1, sym, sti2)`

Adds a new transition. Transition is from `sti1` to `sti2` consuming symbol `sym`. `sti2` is a unique state, not a set of them. Reversed transition function is also computed

Parameters

- **sti1** (*int*) – state index of departure
- **sti2** (*int*) – state index of arrival
- **sym** (*str*) – symbol consumed

`delTransition(sti1, sym, sti2, _no_check=False)`

Remove a transition if existing and perform cleanup on the transition function's internal data structure and in the reversal transition function

Parameters

- **sti1** (*int*) – state index of departure
- **sti2** (*int*) – state index of arrival
- **sym** (*str*) – symbol consumed
- **_no_check** (*bool*) – (optional) dismiss secure code

deleteStates(*del_states*)

Delete given iterable collection of states from the automaton. Performe deletion in the transition function and its reversal.

Parameters
del_states (*set* | *list*) – collection of states indexes

elimEpsilon0()

Eliminate epsilon-transitions from this automaton, with reduction of states through elimination of Epsilon-cycles, and single epsilon-transition cases.

Return type
NFAr

Attention: performs inplace modification of automaton

homogenousP(*inplace=False*)

Checks is the automaton is homogenous, i.e.the transitions that reaches a state have all the same label.

Parameters
inplace (*bool*) – if True performs Epsilon transitions elimination

Returns
True if homogenous

Return type
bool

mergeStates(*f*, *t*)

Merge the first given state into the second. If first state is an initial or final state, the second becomes respectively an initial or final state.

Parameters

- **f** (*int*) – index of state to be absorbed
- **t** (*int*) – index of remaining state

Attention: It is up to the caller to remove the disconnected state. This can be achieved with `trim().

mergeStatesSet(*tomerge*, *target=None*)

Merge a set of states with a target merge state. If the states in the set have transitions among them, those transitions will be directly merged into the target state.

Parameters

- **tomerge** (*set*) – set of states to merge with target
- **target** (*int*) – optional target state

Note: if target state is not given, the minimal index with be considered.

Attention: The states of the list will become unreacheable, but won't be removed. It is up to the caller to remove them. That can be achieved with `trim()`.

`toNFA()`

Turn into an instance of NFA, and remove the reverse mapping of the delta function.

Returns
shallow copy without reverse delta function

Return type
NFA

`unlinkSoleIncoming(state)`

If given state has only one incoming transition (indegree is one), and it's through epsilon, then remove such transition and return the source state.

Parameters
state (*int*) – state to check

Returns
source state

Return type
int | None

Note: if conditions aren't met, returned source state is None, and automaton remains unmodified.

`unlinkSoleOutgoing(state)`

If given state has only one outgoing transition (outdegree is one), and it's through epsilon, then remove such transition and return the target state.

Parameters
state (*int*) – state to check

Returns
target state

Return type
int | None

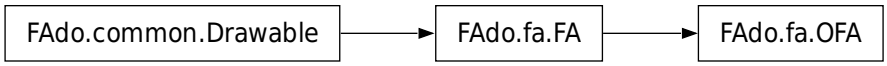
Note: if conditions aren't met, returned target state is None, and automaton remains unmodified.

`class OFA`

Base class for one-way automata

Variables

- **States** (*list*) – set of states.
- **sigma** (*set*) – alphabet set.
- **Initial** (*int*) – the initial state index.
- **Final** (*set*) – set of final states indexes.
- **delta** (*dict*) – the transition function.



acyclicP(*strict=True*)

Checks if the FA is acyclic

Parameters
strict (*bool*) – if not True loops are allowed

Returns: True if the FA is acyclic
bool: True if the FA is acyclic

abstract addTransition(*st1*, *sym*, *st2*)

Add transition

Parameters

- **st1** (*int*) – departing state
- **sym** (*str*) – label
- **st2** (*int*) – arriving state

dotDrawTransition(*st1*, *label*, *st2*, *sep='\n'*)

Draw a transition in dot format

Parameters

- **st1** (*str*) – departing state
- **label** (*str*) – symbol
- **st2** (*str*) – arriving state
- **sep** (*str*) – separator

Return type
str

dump()

Returns a python representation of the object

Returns
the python representation (Tags,States,sigma,delta,Initial,Final)

Return type
tuple

emptyP()

Tests if the automaton accepts an empty language

Return type
bool

New in version 1.0.

abstract evalSymbol(*stil*, *sym*)

Eval symbol

leftQuotient(*other*)

Returns the quotient (NFA) of a language by another language, both given by FA.

Parameters
other (OFA) – the language to be quotient by

Returns
the quotient

Return type
NFA

minimalBrzozowski()

Constructs the equivalent minimal DFA using Brzozowski's algorithm

Returns
equivalent minimal DFA

Return type
DFA

minimalBrzozowskiP()

Tests if the FA is minimal using Brzozowski's algorithm

Return type
bool

quotient(*other*)

Synonymous of leftQotient

rightQuotient(*other*)

Returns the quotient (NFA) of a language by another language, both given by FA.

Parameters
other (OFA) – the language to be quotient by

Returns
the quotient

Return type
NFA

abstract stateChildren(*_state*, *_strict=None*)

To be implemented below

Parameters

- **_state** (*state*) –
- **_strict** (*int*) – state id queried

Return type
list

abstract succintTransitions()

Collapsed transitions

`topoSort()`

Topological order for the FA

Returns
List of state indexes

Return type
list

Note: self loops are taken in consideration

`trim()`

Removes the states that do not lead to a final state, or, inclusively,
that can't be reached from the initial state. Only useful states remain.

Return type
FA

Attention: in place transformation

`trimP()`

Tests if the FA is trim: initially connected and co-accessible

Return type
bool

`class SemiDFA`

Class of automata without initial or final states

Variables

- **States** (*list*) – set of states.
- **sigma** (*set*) – alphabet set.
- **delta** (*dict*) – the transition function.

`dotDrawState`(*sti: int*, *sep='\n'*) → str

Dot representation of a state

Parameters

- **sti** (*int*) – state index.
- **sep** (*str*, optional) – separator.

Returns
line to add to the dot file.

Return type
str

`static dotDrawTransition`(*st1: str*, *lbl1: str*, *st2*, *sep='\n'*) → str

Draw a transition in dot format

Parameters

- **st1** (*str*) – departing state.

- **lbl1** (*str*) – label.
- **st2** (*str*) – arriving state.
- **sep** (str, optional) – separator.

Returns
line to add to the dot file.

Return type
str

dotFormat(*size='20,20'*, *filename=None*, *direction='LR'*, *strict=False*, *maxlblsz=6*, *sep='\n'*) → str

A dot representation

Parameters

- **direction** (*str*) – direction of drawdrawing - "LR" or "RL"
- **size** (*str*) – size of image
- **filename** (*str*) – Name of the output file
- **sep** (*str*) – line separator
- **maxlblsz** (*int*) – max size of labels before getting removed
- **strict** (*bool*) – use limitations of label sizes

Returns
the dot representation

Return type
str

New in version 0.9.6.

Changed in version 1.2.1.

emptyDFA(*sigma=None*)

Given an alphabet returns the minimal DFA for the empty language

Parameters
sigma (*set*) – set of symbols

Return type
DFA

New in version 1.3.4.2.

reduce_size(*aut:* DFA, *maxIter=None*) → *DFA*

A smaller (if possible) DFA. To use with huge automata.

Parameters

- **aut** (DFA) – the aoutomata to reduce
- **maxIter** (*int*) – the maxiimum number of iterations before return

Return type
DFA

saveToString(*aut:* FA, *sep='&'*) → str

Finite automata definition as a string using the input format.

New in version 0.9.5.

Changed in version 0.9.6: Names are now used instead of indexes.

Changed in version 0.9.7: New format with quotes and alphabet

Parameters

- **aut** (FA) – the FA
- **sep** (*str*) – separation between *lines*

Returns
the representation

Return type
str

sigmaStarDFA(*sigma=None*) → *DFA*

Given a alphabet s returns the minimal DFA for s*

Parameters
sigma (*set*) – set of simbols

Return type
DFA

New in version 1.2.

statePP(*state*)

Pretty print state

Parameters
state –

Returns

stringToDFA(*s: list*, *f: list*, *n: int*, *k: int*) → *DFA*

Converts a string icdfa's representation to dfa.

Parameters

- **s** (*list*) – canonical string representation
- **f** (*list*) – bit map of final states
- **n** (*int*) – number of states
- **k** (*int*) – number of symbols

Returns
a complete dfa with sigma [k], States [n]

Return type
DFA

Changed in version 0.9.8: symbols are converted to str

symbolDFA(*sym*, *sigma=None*) → *DFA*

Given symbol and an alphabet returns the minimal DFA that aceepts that symbol

Parameters

- **sym** – symbol
- **sigma** (*set*) – set of symbols

Return type
DFA

1.2 FAdo.common

Common definitions for FAdo files

class AllWords(*alphabet*)

Iterator thar generates all words of an alphabet in militar order

exception CFGerror

exception CFGgrammarError(*rule*)

exception CFGterminalError(*size*)

exception CodesError(*msg*)

exception CodingTheoryError(*msg*)

exception DFAEmptyDFA

exception DFAEmptySigma

exception DFAFileError

exception DFAFound(*word*)

exception DFASyntaticError(*line*)

exception DFAdifferentSigma

exception DFAepsilonRedefinition

exception DFAequivalent

exception DFAerror

exception DFAinputError(*word*)

exception DFAmarkedError(*sym*)

exception DFAnoInitial

exception DFAnotComplete

exception DFAnotMinimal

exception DFAnotNFA(*msg*)

exception DFAstateUnknown(*stidx*)

exception DFAstopped

exception DFAsymbolUnknown(*sym*)

class Drawable

Any FAdo object that is drawable

display(*filename=None*, *size='30,20'*, *strict=False*, *maxlblsz=6*)

Display automata using dot

Parameters

- **size** – size of representation
- **filename** – filename to use for the graphic representation (default a os tmpfile)
- **maxlblsz** (*int*) – max size of labels before getting removed
- **strict** (*bool*) – use limitations of label sizes

Changed in version 1.2.1.

abstract dotFormat(*size='20,20'*, *filename=None*, *direction='LR'*, *strict=False*, *maxlblsz=6*, *sep='\n'*)

Some dot representation

Parameters

- **size** (*str*) – size parameter for dotviz
- **filename** (*str*) – filename
- **direction** (*str*) –
- **strict** (*bool*) –
- **maxlblsz** (*int*) –
- **sep** (*str*) –

Returns: str:

dotLabel(*lbl0*)

Label string

makePNG(*filename=None*, *size='30,20'*)

Produce png file to display

Parameters

- **filename** (*str*) – file name, if None will be a tmpfile
- **size** – size for graphviz

Returns

name of the file created

New in version 1.0.4.

exception DuplicateName(*number*)

exception FAException

exception FASiseMismatch

exception FAdoError

exception FAdoGeneralError(*msg*)

exception FAdoNotImplemented

exception FAdoSyntacticError

exception GraphError(*message*)

exception IllegalBias(*msg*)

class Memoized(*func*)

Decorator that caches a function's return value each time it is called.

If called later with the same arguments, the cached value is returned, and not re-evaluated.

exception NFAEmpty(*msg=''*)

exception NFAerror(*msg=''*)

exception NImplemented

exception NonPlanar

exception NotSP

exception PDAerror

exception PDAsymbolUnknown(*symb*)

exception PEGError(*msg*)

exception ParRangError

exception PropertyNotSatisfied(*msg*)

class SPLabel(*val=None*)

Label class for Serial-Paralel test algorithm

See also:

Moreira & Reis, Fundamenta Informatica, 'Series-Paralel automata and short regular expressions', n.91 3-4, pag 611-629

exception SSBadTransition

exception SSError

exception SSMissAlphabet

exception TFAAccept

exception TFAReject

exception TFARejectBlocked

exception TFARejectLoop

exception TFARejectNonFinal

exception TFASignal

exception TRError

exception TstError(*message*)

class TwDict(*fw=None*)

A class for dictionaries 'both ways'

exception TypeError

exception VersoError(*msg*)

exception VertexNotInGraph

class Word(*data=None*)

Class to implement generic words as iterables with pretty-print

Basically a unified way to deal with words with caracters of sizes different of one with no much fuss

binomial(*n*, *k*)

Exactly what it seems

Parameters

- **n** (*int*) – n
- **k** (*int*) – k

Return type
int

delFromList(*l*, *l1*)

Delete every element of l1 from l

Parameters

- **l** (*list*) –
- **l1** (*list*) –

dememoize(*cls*, *method_name*)

Restore method of given class from Memoized state. Stored attributes will be removed.

exception fnhException

forceIterable(*x*)

Forces a non-iterable object into a list, otherwise returns itself

Parameters
x (*list*) – the object

Returns
object as a list

Return type
list

fromBase(*n: list*, *b: int*) → int

Converts a number n in base b into an integer

Parameters

- **n** (*list*) – number to convert
- **b** (*int*) – base used

Returns: int .. versionadded: 2.1.3

graphvizTranslate(*s*, *strict=False*, *maxlblsz=6*)

Translate epsilons for graphviz

Parameters

- **s** (*str*) – symbol
- **maxlblsz** – max size of labels before getting removed
- **strict** (*bool*) – use limitations of label sizes

Return type
str

homogeneousP(*l*)

Is the list homogeneous?

Parameters
l (*list*) – list to be inspected

Return type
bool

inBase(*n: int*, *base: int*, *tail=None*) → list

Writes the representation of a non-null natural in a base.

Parameters

- **n** – number to conver
- **base** – base to use

Returns: list of integers

New in version 2.1.3.

lSet(*s: set*)

returns the last element of a set

Parameters
s (*set*) – the set

Returns
the last element of the set

New in version 1.3.3.

memoize(*cls*, *method_name*)

Memoizes a given method result on instances of given class.

Given method should have no side effects. Results are stored as instance attributes — given parameters are disregarded.

Parameters

- **cls** –
- **method_name** –

exception notAcyclic

overlapFreeP(*word*)

Returns True if word is overlap free, i.e, no proper and nonempty
prefix is a suffix

Parameters
word – the word

Return type
Boolean

pad(*n: int*, *nu: list*) → list

Pads the given list nu to have the appropriate number of leading 0 up to size n

Parameters

- **n** – number of algarisms
- **nu** – list

padList(*l: list*, *size: int*) → list

Pads the list l, with zeros, up to the size size

Parameters

- **l** (*list*) – the list to pad
- **size** (*int*) – the desired size

Returns
the resulting list

Return type
list

New in version 2.1.3.

exception regexpInvalid(*word*)

exception regexpInvalidMethod

exception regexpInvalidSymbols

sConcat(*x*, *y*)

CConcat words

Parameters

- **x** – first word
- **y** – second word

Returns
concatenation word

suffixes(*word*)

Returns the list of proper suffixes of a word

Parameters
word (*str*) – the word

Return type
list

uSet(*s: set*)

returns the first element of a set

Parameters
s (*set*) – the set

Returns
the first element of s

unifSzSubset(*max: int*) → int

Returns a size uniformly distributed for a variable that behaves like a subset of a max element set.

Parameters
max (*int*) – max size to accept

Returns
the size

Return type
int

New in version 2.1.3.

unique(*l*)

Eliminate duplicates

Parameters
l (*list*) – source list

Returns
list wthout repetitions

Return type
lst

zeta(*s: int | float, t=100*) → float | complex

Implementation of Riemman's zeta function

1.3 FAdo.conversions

Conversions between objects.

Deterministic and non-deterministic automata manipulation, conversion and evaluation. .. *Authors:* Rogério Reis & Nelma Moreira .. *This is part of FAdo project* https://fado.dcc.fc.up.pt.

DFA2regexpDijkstra(*aut*) → *RegExp*

Returns a regexp for the current DFA considering the recursive method. Very inefficent.

Parameters
aut (DFA) – the automaton

Returns
a regexp equivalent to the current DFA

Return type
reex.RegExp

DFAsyncWords(*aut*)

Evaluates the regular expression corresponding to the synchronizing pwords of the automata.

Parameters
aut (DFA) – the automata

Returns
a regular expression of the sync words of the automata

Return type
reex.RegExp

FA2GFA(*aut*)

Creates a GFA equivalent to NFA

Parameters
aut (OFA) – the automaton

Returns
deep copy

Return type
GFA

FA2regexpCG(*aut*)

Regular expression from state elimination whose language is recognised by the FA. Uses a heuristic to choose the order of elimination.

Parameters
aut (OFA) – the automaton

Returns
the equivalent regular expression

Return type
reex.RegExp

FA2regexpCG_nn(*aut:* OFA)

Regular expression from state elimination whose language is recognised by the FA. Uses a heuristic to choose the order of elimination. The FA is not normalized before the state elimination.

Parameters
aut (OFA) – the automaton

Returns
the equivalent regular expression

Return type
reex.RegExp

FA2regexpDynamicCycleHeuristic(*aut*)

State elimination Heuristic based on the number of cycles that passes through each state. Here those numbers are evaluated dynamically after each elimination step

Parameters
aut (OFA) – the automaton

Returns
an equivalent regular expression

Return type
reex.RegExp

See also:

Nelma Moreira, Davide Nabais, and Rogério Reis. State elimination ordering strategies: Some experimental results. Proc. of 11th Workshop on Descriptional Complexity of Formal Systems (DCFS10), pages 169-180.2010. DOI: 10.4204/EPTCS.31.16

FA2regexpSE(*aut*)

A regular expression obtained by state elimination algorithm whose language is recognised by the FA aut.

Parameters
aut (OFA) – the automaton

Returns
the equivalent regular expression

Return type
reex.RegExp

FA2regexpSEO(*aut*, *order=None*)

Regular expression from state elimination whose language is recognised by the FA. The FA is normalized before the state elimination.

Parameters

- **aut** (OFA) – the automaton
- **order** (*list*) – state elimination sequence

Returns
the equivalent regular expression

Return type
reex.RegExp

FA2regexpSE_nn(*aut*, *order=None*)

Regular expression from state elimination whose language is recognised by the FA. The FA is not normalized before the state elimination.

Parameters

- **aut** (OFA) – the automaton
- **order** (*list*) – state elimination sequence

Returns
the equivalent regular expression

Return type
reex.RegExp

FA2regexpStaticCycleHeuristic(*aut*)

State elimination Heuristic based on the number of cycles that passes through each state. Here those numbers are evaluated statically in the beginning of the process

Parameters
aut (OFA) – the automaton

Returns
a equivalent regular expression

Return type
reex.RegExp

See also:

Nelma Moreira, Davide Nabais, and Rogério Reis. State elimination ordering strategies: Some experimental results. Proc. of 11th Workshop on Descriptional Complexity of Formal Systems (DCFS10), pages 169-180.2010. DOI: 10.4204/EPTCS.31.16

FAallRegExps(*aut*)

Evaluates the alphabetic length of the equivalent regular expression using every possible order of state elimination.

Parameters
aut (OFA) – the automaton

Returns
list of tuples (int, list of states)

Return type
listo of tuples

`FAeliminateSingles`(*aut*)

Eliminates every state that only have one successor and one predecessor.

Parameters
aut (OFA) – the automaton

Returns
GFA after eliminating states

Return type
GFA

`class GFA`

Class for Generalized Finite Automata: NFA with a unique initial state and transitions are labeled with RegExp.

`DFS`(*io*)

Depth first search

Parameters
io –

`addTransition`(*sti1*, *sym*, *sti2*)

Adds a new transition from sti1 to sti2 consuming symbol sym. Label of the transition function is a RegExp.

Parameters

- **sti1** (*int*) – state index of departure
- **sti2** (*int*) – state index of arrival
- **sym** (*str*) – symbol consumed

Raises
DFAepsilonRedefenition – if sym is Epsilon

`assignLow`(*st*)

Parameters
st –

`assignNum`(*st*)

Parameters
st –

completeDelta()

Adds empty set transitions between the automatons final and initial states in order to make it complete. It's only meant to be used in the final stage of SEA...

deleteState(*sti*)

deletes a state from the GFA :param sti:

dfs_visit(*s*, *visited*, *io*)

Parameters

- **s** – state
- **visited** – list od states visited
- **io** –

dup()

Returns a copy of a GFA

Return type
GFA

eliminate(*st*)

Eliminate a state.

Parameters
st (*int*) – state to be eliminated

eliminateAll(*lr*)

Eliminate a list of states.

Parameters
lr (*list*) – list of states indexes

eliminateState(*st*)

Deletes a state and updates the automaton

Parameters
st (*int*) – the state to be deleted

evalNumberOfStateCycles()

Evaluates the number of cycles each state participates

Returns
state->list of cycle lengths

Return type
dict

evalSymbol(*stil*, *sym*)

Eval symbol

normalize()

Create a single initial and final state with Epsilon transitions.

Attention: works in place

reorder(*dictio*)

Reorder states indexes according to given dictionary.

Parameters
dictio (*dict*) – order

Note: dictionary does not have to be complete

stateChildren(*state*, *strict=False*)

Set of children of a state

Parameters

- **strict** (*bool*) – a state is never its own children even if a self loop is in place
- **state** (*int*) – state id queried

Returns
map: children -> alphabetic length

Return type
dictionary

succintTransitions()

Collapsed transitions

weight(*state*)

Calculates the weight of a state based on a heuristic

Parameters
state (*int*) – state

Returns
the weight of the state

Return type
int

weightWithCycles(*state*, *cycles*)

Parameters

- **state** –
- **cycles** –

Returns

SP2regexp(*aut*)

Checks if FA is SP (Serial-PArallel), and if so returns the regular expression whose language is recognised by the FA

Parameters
aut (*OFA*) – the automaton

Returns
equivalent regular expression

Return type
reex.RegExp

Raises
NotSP – if the automaton is not Serial-Parallel

See also:

Moreira & Reis, Fundamenta Informatica, Series-Parallel automata and short regular expressions, n.91 3-4, pag 611-629. https://www.dcc.fc.up.pt/~nam/publica/spa07.pdf

---

Note: Automata must be Serial-Parallel

`cutPoints`(*aut*)

Set of FA's cut points

Parameters
aut (OFA) – the automaton

Return type
set of states

1.4 FAdo.fio

In/Out.

FAdo I/O methods. The parsing grammars for most of the objects reside here.

`class BuildFadoObject`(*visit_tokens: bool = True*)

Semantics of the FAdo grammars' objects

`readFromFile`(*FileName*)

Reads list of finite automata definition from a file.

Parameters
FileName (*str*) – file name

Return type
list

The format of these files must be the as simple as possible:

- # begins a comment
- @DFA or @NFA begin a new automata (and determines its type) and must be followed by the list of the final states separated by blanks
- fields are separated by a blank and transitions by a CR: `state symbol new state`
- in case of a NFA declaration, the "symbol" @epsilon is interpreted as an epsilon-transition
- the source state of the first transition is the initial state
- in the case of a NFA, its declaration @NFA can, after the declaration of the final states, have a * followed by the list of initial states
- both, NFA and DFA, may have a declaration of alphabet starting with a $ followed by the symbols of the alphabet
- a line with a sigle name, decrares a state

```
FAdo        ::=   FA | FA CR FAdo
FA          ::=   DFA | NFA | Transducer
DFA         ::=   "@DFA" LsStates Alphabet CR dTrans
NFA         ::=   "@NFA" LsStates Initials Alphabet CR nTrans
```

```
Transducer  ::=   "@Transducer" LsStates Initials Alphabet Output CR tTrans
Initials    ::=   "*" LsStates | /Epsilon
Alphabet    ::=   "$" LsSymbols | /Epsilon
Output      ::=   "$" LsSymbols | /Epsilon
nSymbol     ::=   symbol | "@epsilon"
LsStates    ::=   stateid | stateid , LsStates
LsSymbols   ::=   symbol | symbol , LsSymbols
dTrans      ::=   stateid symbol stateid |
                | stateid symbol stateid CR dTrans
nTrans      ::=   stateid nSymbol stateid |
                | stateid nSymbol stateid CR nTrans
tTrans      ::=   stateid nSymbol nSymbol stateid |
                | stateid nSymbol nSymbol stateid CR nTrans
```

Note: If an error occur, either syntactic or because of a violation of the declared automata type, an exception is raised

Changed in version 0.9.6.

Changed in version 1.0.

readOneFromFile(*fileName*)

Read the first of the FAdo objects from File

Parameters
fileName (*str*) – name of the file

Return type
DFA|*FA*|STF|*SST*

readOneFromString(*s*)

Reads one finite automata definition from a file.

See also:

readFromFile for description of format

Parameters
s (*str*) – the string

Return type
DFA|*NFA*|*SFT*

saveToFile(*FileName*, *fa*, *mode='a'*)

Saves a list finite automata definition to a file using the input format

Changed in version 0.9.5.

Changed in version 0.9.6.

Changed in version 0.9.7: New format with quotes and alphabet

Parameters

- **FileName** (*str*) – file name
- **fa** (*list of FA*) – the FA
- **mode** (*str*) – writing mode

saveToJson(*FileName*, *aut*, *mode='w'*)

Saves a finite automata definition to a file using the JSON format

saveToString(*fa*)

Saves a finite automaton definition to a string :param fa: automaton :return: the string containing the automaton definition :rtype: str

..versionadded:: 1.2.1

show(*obj*)

General, context sensitive, display method :param obj: the object to show

New in version 1.2.1.

toJson(*aut*)

Json for a FA

Parameters
aut (FA) – the automaton

Return type
str

1.5 FAdo.reex

Regular expressions manipulation

Regular expression classes and manipulation

class BuildRPNRegexp(*context=None*)

class BuildRPNSRE(*context=None*)

class BuildRegexp(*context=None*)

Semantics of the FAdo grammars' regexps Priorities of operators: disj > conj > shuffle > concat > not > star >= option

class BuildSRE(*context=None*)

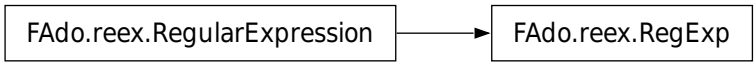
Parser for sre

class CAtom(*val*, *sigma=None*)

Simple Atom (symbol)

Variables

- **Sigma** – alphabet set of strings
- **val** – the actual symbol



Constructor of a regular expression symbol.

Parameters
val – the actual symbol

`PD()`

Closure of partial derivatives of the regular expression in relation to all words.

Returns
set of regular expressions

Return type
set

See also:

Antimirov, 95

`static alphabeticLength()`

Number of occurrences of alphabet symbols in the regular expression.

Return type
int

Attention: Doesn't include the empty word.

`derivative(sigma)`

Derivative of the regular expression in relation to the given symbol.

Parameters
sigma (*str*) – an arbitrary symbol.

Returns
regular expression

Return type
RegExp

Note: whether the symbols belong to the expression's alphabet goes unchecked. The given symbol will be matched against the string representation of the regular expression's symbol.

See also:

J. A. Brzozowski, Derivatives of Regular Expressions. J. ACM 11(4): 481-494 (1964)

`static epsilonLength()`

Number of occurrences of the empty word in the regular expression.

Return type
int

`first(parent_first=None)`

List of possible symbols matching the first symbol of a string in the language of the regular expression.

Parameters
parent_first (*list*) –

Returns
list of symbols

Return type
list

`first_l()`

First set for locations

`followLists(lists=None)`

Map of each symbol's follow list in the regular expression.

Parameters
lists (*dict*) –

Returns
map of symbols' follow lists {symbol: list of symbols}

Return type
dict

> **Attention:** For first() and last() return lists, the follow list for certain symbols might have repetitions in the case of follow maps calculated from Star operators. The union of last(), first() and follow() sets are always disjoint when the regular expression is in Star normal form (Brüggemann-Klein, 92), therefore FAdo implements them as lists. You should order exclusively, or take a set from a list in order to resolve repetitions.

`followListsD(lists=None)`

Map of each symbol's follow list in the regular expression.

Parameters
lists (*dict*) –

Returns
map of symbols' follow list {symbol: list of symbols}

Return type
dict

> **Attention:** For first() and last() return lists, the follow list for certain symbols might have repetitions in the case of follow maps calculated from star operators. The union of last(), first() and follow() sets are always disjoint

See also:

Sabine Broda, António Machiavelo, Nelma Moreira, and Rogério Reis. On the average size of glushkov and partial derivative automata. International Journal of Foundations of Computer Science, 23(5):969-984, 2012.

`followListsStar(lists=None)`

Map of each symbol's follow list in the regular expression under a star.

Parameters
lists (*dict*) –

Returns
map of symbols' follow lists {symbol: list of symbols}

Return type

dict

See also:

Sabine Broda, António Machiavelo, Nelma Moreira, and Rogério Reis. On the average size of glushkov and partial derivative automata. International Journal of Foundations of Computer Science, 23(5):969-984, 2012.

`follow_l()`

Follow set for locations

`last(parent_last=None)`

List of possible symbols matching the last symbol of a string in the language of the regular expression.

Parameters
parent_last (*list*) –

Returns
list of symbols

Return type
list

`last_l()`

Last set for locations

`linearForm()`

Linear form of the regular expression , as a mapping from heads to sets of tails, so that each pair (head, tail) is a monomial in the set of linear forms.

Returns:
dict: dictionary mapping heads to sets of tails

See also:

Antimirov, 95

`linearFormC()`

Returns
linear form

Return type
dict

`linearP()`

Whether the regular expression is linear; i.e., the occurrence of a symbol in the expression is unique.

Return type
bool

`mark()`

Return type
MAtom

`static measure(from_parent=None)`

A list with four measures for regular expressions.

Parameters
from_parent –

Returns
the measures

Return type
[int,int,int,int]

[alphabeticLength, treeLength, epsilonLength, starHeight]

1. alphabeticLength: number of occurences of symbols of the alphabet;
2. treeLength: number of functors in the regular expression, including constants.
3. epsilonLength: number of occurrences of the empty word.
4. starHeight: highest level of nested Kleene stars, starting at one for one star occurrence.
5. disjLength: number of occurrences of the disj operator
6. concatLength: number of occurrences of the concat operator
7. starLength: number of occurrences of the star operator
8. conjLength: number of occurrences of the conj operator
9. starLength: number of occurrences of the shuffle operator

Attention: Methods for each of the measures are implemented independently. This is the most effective for obtaining more than one measure.

`nfaThompson()`

Epsilon-NFA constructed with Thompson's method that accepts the regular expression's language.

Returns
NFA Thompson

Return type
NFA

See also:

K. Thompson. Regular Expression Search Algorithm. CACM 11(6), 419-422 (1968)

`partialDerivatives(sigma)`

Set of partial derivatives of the regular expression in relation to given symbol.

Parameters
sigma (*str*) – symbol in relation to which the derivative will be calculated.

Returns
set of regular expressions

Return type
set

See also:

Antimirov, 95

`partialDerivativesC(sigma)`

Parameters
sigma (*str*) – symbol

Returns
set of partial derivatives

Return type
set

`reduced`(*has_epsilon=False*)

Equivalent regular expression with the following cases simplified:

1. Epsilon.RE = RE.Epsilon = RE
2. EmptySet.RE = RE.EmptySet = EmptySet
3. EmptySet + RE = RE + EmptySet = RE
4. Epsilon + RE = RE + Epsilon = RE, where Epsilon is in L(RE)
5. RE** = RE*
6. EmptySet* = Epsilon* = Epsilon

7.Epsilon:RE = RE:Epsilon= RE

Parameters

- **has_epsilon** (*bool*) – used internally to indicate that the language of which this term is a subterm has the empty
- **word.** –

Returns
regular expression

Return type
RegExp

Attention: Returned structure isn't strictly a duplicate. Use __copy__() for that purpose.

`reversal`()

Reversal of RegExp

Return type
Regexp

`rpn`()

RPN representation

Returns
printable RPN representation

Return type
str

`setOfSymbols`()

Set of symbols that occur in a regular expression..

Returns
set of symbols

Return type
set

snf(*hollowdot=False*)

Star Normal Form (SNF) of the regular expression.

Parameters
hollowdot (*bool*) – if True computes hollow dot function else black dot

Returns
regular expression in star normal form

Return type
RegExp

static starHeight()

Maximum level of nested regular expressions with a star operation applied.
For instance, starHeight(((a*b)*+b*)*) is 3.

Return type
int

stringLength()

Length of the string representation of the regular expression.

Returns
string length

Return type
int

support(*side=True*)

Support of a regular expression.

Parameters
side (*bool*) – if True concatenation of a set on the left if False on the right (prefix support)

Returns
set of regular expressions

Return type
set

See also:

Champarnaud, J.M., Ziadi, D.: From Mirkin's prebases to Antimirov's word partial derivative. Fundam. Inform. 45(3), 195-205 (2001)

See also:

Maia et al, Prefix and Right-partial derivative automata, 11th CIE 2015, 258-267 LNCS 9136, 2015

supportlast(*side=True*)

Subset of support such that elements have ewp

Parameters
side (*bool*) – if True left-derivatives else right-derivatives

Returns
set of partial derivatives

Return type
set

static syntacticLength()

Number of nodes of the regular expression's syntactical tree (sets).

Return type
int

tailForm()

Returns
tail form

Return type
dict

static treeLength()

Number of nodes of the regular expression's syntactical tree.

Return type
int

unmarked()

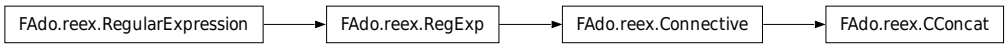
The unmarked form of the regular expression. Each leaf in its syntactical tree becomes a RegExp(), the CEpsilon() or the CEmptySet().

Returns
(general) regular expression

Return type
RegExp

class CConcat(*arg1*, *arg2*, *sigma=None*)

Class for catenation operation on regular expressions.



ewp()

Whether the empty word property holds for this regular expression's language.

Return type
bool

first(*parent_first=None*)

First set

Returns
first position set

Return type
set

first_l()

First sets for locations

followLists(*lists=None*)

Follow set

Returns
for each key position a set of follow positions

Return type
dict

followListsD(*lists=None*)

Follow set

Returns
for each key position a set of follow positions

Return type
dict

follow_l()

Follow sets for locations

last(*parent_last=None*)

Last set

Returns
last position set

Return type
set

last_l()

Last sets for locations

linearForm()

Returns
linear form

Return type
dict

mark()

Make all atoms maked (tag False)

Return type
RegExp

reversal()

Reversal of RegExp

Return type
reex.RegExp

rpn()

Return type
str

snf(*_hollowdot=False*)

Star Normal Form

support(*side=True*)

Set of partial derivatives

tailForm()

Returns
tail form

Return type
dict

unmark()

Conversion back to unmarked atoms :rtype: CConcat

class CConj(*arg1*, *arg2*, *sigma=None*)

Intersection operation of regexps

ewp()

Whether the empty word property holds for this regular expression's language.

Return type
bool

first_l()

First sets for locations

follow_l()

Follow sets for locations

last_l()

Last sets for locations

linearForm()

Returns
linear form

Return type
dict

mark()

Make all atoms maked (tag False)

Return type
RegExp

rpn()

RPN representation

Returns
printable RPN representation

Return type
str

snf()

Star Normal Form

`tailForm()`

Returns
tail form

Return type
dict

`class CDisj(arg1, arg2, sigma=None)`

Class for disjunction/union operation on regular expressions.

`ewp()`

Whether the empty word property holds for this regular expression's language.

Return type
bool

`first(parent_first=None)`

First set

Returns
first position set

Return type
set

`first_l()`

First sets for locations

`followLists(lists=None)`

Follow set

Returns
for each key position a set of follow positions

Return type
dict

`followListsD(lists=None)`

Follow set

Returns
for each key position a set of follow positions

Return type
dict

`follow_l()`

Follow sets for locations

`last`(*parent_last=None*)

Last set

Returns
last position set

Return type
set

`last_l`()

Last sets for locations

`linearForm`()

Returns
linear form

Return type
dict

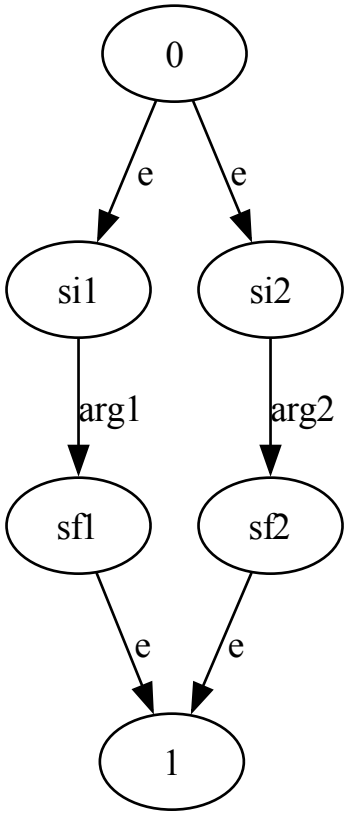
`mark`()

Convertion to marked atoms :rtype: CDisj

`nfaThompson`()

Returns an NFA (Thompson) that accepts the RE.

Return type
NFA



reversal()

Reversal of RegExp

Return type
reex.RegExp

rpn()

RPN representation

Returns
printable RPN representation

Return type
str

snf(*hollowdot=False*)

Star Normal Form

support(*side=True*)

Set of partial derivatives

tailForm()

Returns
tail form

Return type
dict

`unmark()`

Conversion back to unmarked atoms :rtype: CDisj

`class CEmptySet(sigma=None)`

Class that represents the empty set.

Constructor of a regular expression symbol.

Parameters
`val` – the actual symbol

`static emptysetP()`

Returns

`static epsilonLength()`

Returns

`static epsilonP()`

Returns

`ewp()`

Returns

`static measure(from_parent=None)`

Parameters
`from_parent` –

Returns

`nfaPD(pdmethod='nfaPDNaive')`

Computes the partial derivative automaton

`partialDerivatives(_)`

Partial derivatives

`partialDerivativesC(_)`

Returns

`rpn()`

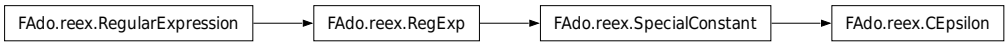
Returns

snf(*_hollowdot=False*)

Star Normal Form

class CEpsilon(*sigma=None*)

Class that represents the empty word.



Constructor of a regular expression symbol.

Parameters
val – the actual symbol

static epsilonLength()

Number of occurrences of the empty word in the regular expression.

Returns
number of epsilons

Return type
int

static epsilonP()

Return type
bool

ewp()

Return type
bool

static measure(*from_parent=None*)

Parameters
from_parent –

Returns
measures

nfaThompson()

Return type
NFA

partialDerivatives(*_*)

Returns

partialDerivativesC(*_*)

Returns

rpn()

Returns
str

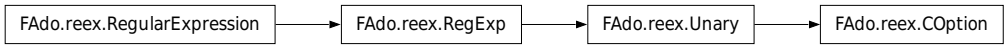
snf(*_hollowdot=False*)

Parameters
_hollowdot –

Returns

class **COption**(*arg*, *sigma=None*)

Class for option operation (reg + @epsilon) on regular expressions.



epsilonLength()

Number of occurrences of the empty word in the regular expression.

Returns
number of epsilons

Return type
int

ewp()

Whether the empty word property holds for this regular expression's language.

Return type
bool

first(*parent_first=None*)

First set

Returns
first position set

Return type
set

first_l()

First sets for locations

followLists(*lists=None*)

Follow set

Returns
for each key position a set of follow positions

Return type
dict

followListsD(*lists=None*)

Follow set

Returns
for each key position a set of follow positions

Return type
dict

followListsStar(*lists=None*)

to be fixed

follow_l()

Follow sets for locations

last(*parent_first=None*)

Last set

Returns
last position set

Return type
set

last_l()

Last sets for locations

linearForm()

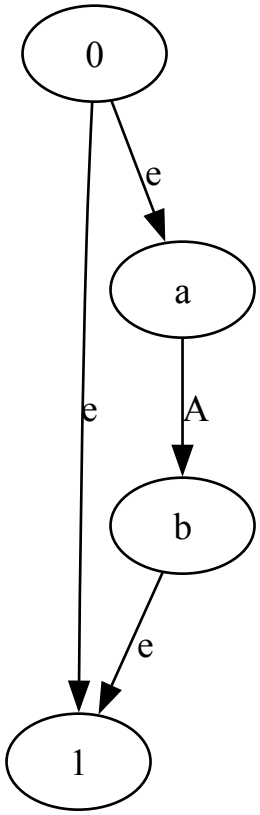
Returns
linear form

Return type
dict

nfaThompson()

Returns a NFA that accepts the RE.

Return type
NFA



rpn()

RPN representation

Returns
printable RPN representation

Return type
str

setOfSymbols()

Returns
set of symbols

Return type
set

snf(*_hollowdot=False*)

Star Normal Form

starHeight()

Maximum level of nested regular expressions with a star operation applied.

For instance, starHeight(((a*b)*+b*)*) is 3.

Returns
number of nested star

Return type
int

`support`(*side=True*)

Set of partial derivatives

`tailForm`()

Returns
tail form

Return type
dict

`class CShuffle`(*arg1*, *arg2*, *sigma=None*)

Shuffle operation of regexps

`ewp`()

Whether the empty word property holds for this regular expression's language.

Return type
bool

`first`(*parent_list=None*)

Parameters
parent_list –

Returns

`first_l`()

First sets for locations

`followListsD`(*lists=None*)

in progress

`follow_l`()

Follow sets for locations

`last_l`()

Last sets for locations

`linearForm`()

Returns
linear form

Return type
dict

`mark`()

Make all atoms maked (tag False)

Return type
RegExp

rpn()

RPN representation

Returns
printable RPN representation

Return type
str

snf()

Star Normal Form

tailForm()

Returns
tail form

Return type
dict

class CShuffleU(*arg*, *sigma=None*)

Unary Shuffle operation of regexps

epsilonLength()

Number of occurrences of the empty word in the regular expression.

Returns
number of epsilons

Return type
int

ewp()

Whether the empty word property holds for this regular expression's language.

Return type
bool

first(*parent_list=None*)

Parameters
parent_list –

Returns

followListsD(*lists=None*)

in progress

last(*parent_last=None*)

Last set

Returns
last position set

Return type
set

linearForm()

Returns
linear form

Return type
dict

`mark()`

Make all atoms maked (tag False)

Return type
RegExp

`rpn()`

RPN representation

Returns
printable RPN representation

Return type
str

`snf()`

Star Normal Form

`starHeight()`

Maximum level of nested regular expressions with a star operation applied.

For instance, starHeight(((a*b)*+b*)*) is 3.

Returns
number of nested star

Return type
int

`tailForm()`

Returns
tail form

Return type
dict

`unmark()`

Conversion back to RegExp

Return type
reex.__class__

`class CSigmaP(sigma=None)`

Special regular expressions modulo associativity, commutativity, idempotence of disjunction and intersection;
associativity of concatenation; identities sigma^* and sigma^+.

CSigmaP: Class that represents the complement of the EmptySet word (sigma^+)

Constructor of a regular expression symbol.

Parameters
val – the actual symbol

derivative(*sigma*)

Parameters
sigma –

Returns

ewp()

Returns

linearForm()

Returns

linearFormC()

Returns

nfaPD(*pdmethod='nfaPDNaive'*)
Computes the partial derivative automaton

partialDerivatives(*sigma*)

Parameters
sigma –

Returns

partialDerivativesC(*_*)

Parameters
_ –

Returns

rpn()
RPN representation

Returns
printable RPN representation

Return type
str

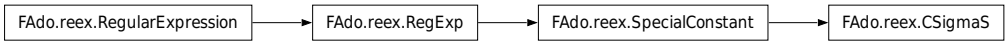
support(*side=True*)

Returns

class CSigmaS(*sigma=None*)

Special regular expressions modulo associativity, commutativity, idempotence of disjunction and intersection;
associativity of concatenation; identities sigma^* and sigma^+.

CSigmaS: Class that represents the complement of the EmptySet set (sigma^*)



Constructor of a regular expression symbol.

Parameters
val – the actual symbol

derivative(*sigma*)

Parameters
sigma –

Returns

ewp()

Returns

linearForm()

Returns

linearFormC()

Returns

nfaPD(*pdmethod='nfaPDNaive'*)
Computes the partial derivative automaton

partialDerivatives(*sigma*)

Parameters
sigma –

Returns

partialDerivativesC(*sigma*)

Parameters
sigma –

Returns

rpn()
RPN representation

Returns
printable RPN representation

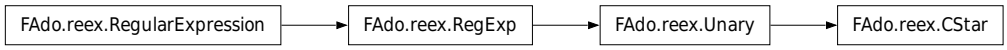
Return type
str

support(*side=True*)

Returns

class CStar(*arg*, *sigma=None*)

Class for iteration operation (aka Kleene star, or Kleene closure) on regular expressions.



epsilonLength()

Number of occurrences of the empty word in the regular expression.

Returns
number of epsilons

Return type
int

ewp()

Whether the empty word property holds for this regular expression's language.

Return type
bool

first(*parent_first=None*)

First set

Returns
first position set

Return type
set

first_l()

First sets for locations

followLists(*lists=None*)

Follow set

Returns
for each key position a set of follow positions

Return type
dict

followListsD(*lists=None*)

Follow set

Returns
for each key position a set of follow positions

Return type
dict

follow_l()

Follow sets for locations

`last`(*parent_first=None*)

Last set

Returns
last position set

Return type
set

`last_l`()

Last sets for locations

`linearForm`()

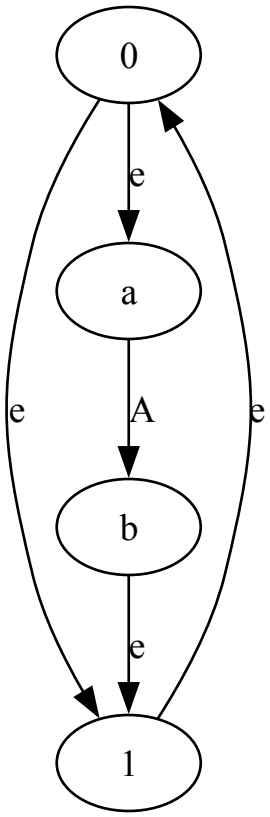
Returns
linear form

Return type
dict

`nfaThompson`()

Returns a NFA that accepts the RE.

Return type
NFA



rpn()

RPN representation

Returns
printable RPN representation

Return type
str

snf(*_hollowdot=False*)

Star Normal Form

starHeight()

Maximum level of nested regular expressions with a star operation applied.

For instance, starHeight(((a*b)*+b*)*) is 3.

Returns
number of nested star

Return type
int

support(*side=True*)

Set of partial derivatives

tailForm()

Returns
tail form

Return type
dict

class Compl(*arg*, *sigma=None*)

Class for not operation on regular expressions.

epsilonLength()

Number of occurrences of the empty word in the regular expression.

Returns
number of epsilons

Return type
int

ewp()

Whether the empty word property holds for this regular expression's language.

Return type
bool

first(_)

First set

Returns
first position set

Return type
set

followLists()

Follow set

Returns
for each key position a set of follow positions

Return type
dict

followListsD()

Follow set

Returns
for each key position a set of follow positions

Return type
dict

last()

Last set

Returns
last position set

Return type
set

linearForm()

Returns
linear form

Return type
dict

mark()

Make all atoms maked (tag False)

Return type
RegExp

rpn()

RPN representation

Returns
printable RPN representation

Return type
str

snf()

Star Normal Form

starHeight()

Maximum level of nested regular expressions with a star operation applied.

For instance, starHeight(((a*b)*+b*)*) is 3.

Returns
number of nested star

Return type
int

support(*side=True*)

Set of partial derivatives

tailForm()

Returns
tail form

Return type
dict

treeLength()

Number of nodes of the regular expression's syntactical tree.

Returns
tree lenght

Return type
int

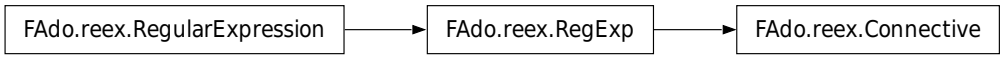
unmark()

Conversion back to RegExp

Return type
reex.__class__

class Connective(*arg1*, *arg2*, *sigma=None*)

Base class for (binary) operations: concatenation, disjunction, etc



alphabeticLength()

Number of occurrences of alphabet symbols in the regular expression. :returns: alphapetic length :rtype: int

Attention: Doesn't include the empty word.

epsilonLength()

Number of occurrences of the empty word in the regular expression.

Returns
number of epsilons

Return type
int

`first`(*parent_first=None*)

First set

Returns
first position set

Return type
set

`followLists`(*lists=None*)

Follow set

Returns
for each key position a set of follow positions

Return type
dict

`followListsD`(*lists=None*)

Follow set

Returns
for each key position a set of follow positions

Return type
dict

`last`(*parent_last=None*)

Last set

Returns
last position set

Return type
set

`abstract linearForm`()

Returns
linear form

Return type
dict

`abstract mark`()

Make all atoms maked (tag False)

Return type
RegExp

`abstract rpn`()

RPN representation

Returns
printable RPN representation

Return type
str

`setOfSymbols()`

Returns
set of symbols

Return type
set

`abstract snf()`

Star Normal Form

`starHeight()`

Maximum level of nested regular expressions with a star operation applied.

For instance, starHeight(((a*b)*+b*)*) is 3.

Returns
number of nested star

Return type
int

`treeLength()`

Number of nodes of the regular expression's syntactical tree.

Returns
tree lenght

Return type
int

`class DAG(reg)`

Class to support dags representing regexps

...seealso: P. Flajolet, P. Sipala, J.-M. Steyaert, Analytic variations on the common subexpression problem,
in: Automata, Languages and Programmin, LNCS, vol. 443, Springer, New York, 1990, pp. 220–234.

Args: reg (RegExp): regular expression

`NFA()`

Returns
the partial derivative automaton

Return type
NFA

`catLF(idl, idr, delay=False)`

Linear form for concatenation :param idl: node :type idl: int :param idr: node :type idr: int :param delay: if true partial derivatives are delayed :type delay: bool

Returns
partial derivatives

Return type
dict

..note:: both arguments are assumed to be already present in the DAG

getAtomIdx(*val*)

Node atom :param val: letter :type val: str

Returns
node id

Return type
int

getIdx(*reg*)

Builds dag nodes :param reg: regular expression :type reg: regexp

Returns
node id

Return type
int

interLF(*diff1*, *diff2*)

Intersection of partial derivatives

Parameters

- **diff1** (*dict*) – partial diff of the first argument
- **diff2** (*dict*) – partial diff of the second argument

Return type
dict

static plusLF(*diff1*, *diff2*)

Union of partial derivatives

Parameters

- **diff1** (*dict*) – partial diff of the first argument
- **diff2** (*dict*) – partial diff of the second argument

Return type
dict

shuffleLF(*id1*, *id2*)

Shuffle of partial derivatives :param id1: node :type id1: int :param id2: node :type id2: int

Returns
partial derivatives

Return type
dict

class DAG_I(*reg*)

Class to support dags representing regexps that inherit from DAG Partial derivatives are buid incrementally

Args: reg (RegExp): regular expression

cat_one(*idl*, *idr*, *c*)

Partial derivative by one symbol for concatenation :param idl: node :type idl: int :param idr: node :type idr: int :param c: symbol :type c: char

Returns
partial derivatives

Return type
set

`evalWordP`(*w*)

Parameters
w (*str*) – a word

Returns
True if w in L(reg)

Return type
bool

`one_derivative`(*id*, *c*)

Parameters

- **c** (*str*) – a symbol
- **id** (*int*) – a node (representing a regexp)

`shuffle_one`(*id1*, *id2*, *c*)

Shuffle of partial derivatives :param id1: node :type id1: int :param id2: node :type id2: int :param c: symbol :type c: char

Returns
of partial derivatives

Return type
set

`class MAtom`(*val*, *mark*, *sigma=None*)

Base class for pointed (marked) regular expressions

Used directly to represent atoms (characters). This class is used to obtain Yamada or Asperti automata. There is no evident use for it, outside this module.

Parameters

- **val** – symbol
- **sigma** – alphabet

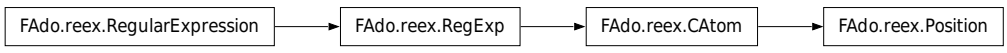
`unmark`()

Conversion back to RegExp

Return type
reex.RegExp

`class Position`(*val*, *sigma=None*)

Class for marked regular expression symbols.



Constructor of a regular expression symbol.

Parameters
val – the actual symbol

`setOfSymbols()`

Set of symbols that occur in a regular expression..

Returns
set of symbols

Return type
set

`unmarked()`

The unmarked form of the regular expression. Each leaf in its syntactical tree becomes a RegExp(), the CEpsilon() or the CEmptySet().

Returns
(general) regular expression

Return type
RegExp

`class Power(arg, n=1, sigma=None)`

Class for Power operation on regular expressions.

`alphabeticLength()`

Number of occurrences of alphabet symbols in the regular expression. :returns: alphapetic length :rtype: int

Attention: Doesn't include the empty word.

`epsilonLength()`

Number of occurrences of the empty word in the regular expression.

Returns
number of epsilons

Return type
int

`first()`

First set

Returns
first position set

Return type
set

followLists()

Follow set

Returns
for each key position a set of follow positions

Return type
dict

followListsD()

Follow set

Returns
for each key position a set of follow positions

Return type
dict

last()

Last set

Returns
last position set

Return type
set

linearForm()

Returns
linear form

Return type
dict

mark()

Make all atoms maked (tag False)

Return type
RegExp

reversal()

Reversal of RegExp

Return type
reex.RegExp

rpn()

RPN representation

Returns
printable RPN representation

Return type
str

setOfSymbols()

Returns
set of symbols

Return type
set

snf()

Star Normal Form

starHeight()

Maximum level of nested regular expressions with a star operation applied.

For instance, starHeight(((a*b)*+b*)*) is 3.

Returns
number of nested star

Return type
int

support(*side=True*)

Set of partial derivatives

tailForm()

Returns
tail form

Return type
dict

treeLength()

Number of nodes of the regular expression's syntactical tree.

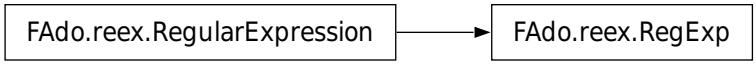
Returns
tree lenght

Return type
int

class RegExp(*sigma=None*)

Base class for regular expressions.

Variables
Sigma – alphabet set of strings



abstract static alphabeticLength()

Number of occurrences of alphabet symbols in the regular expression. :returns: alphapetic length :rtype: int

Attention: Doesn't include the empty word.

compare(*r*, *cmp_method='compareMinimalDFA'*, *nfa_method='nfaPD'*)

Compare with another regular expression for equivalence.

Parameters

- **r** (RegExp) –
- **cmp_method** (*str*) –
- **nfa_method** (*str*) – NFA construction

Returns
True if the expressions are equivalent

Return type
bool

compareMinimalDFA(*r*, *nfa_method='nfaPosition'*)

Compare with another regular expression for equivalence through minimal DFAs.

Parameters

- **r** (RegExp) –
- **nfa_method** (*str*) – NTFA construction

Returns
True if equivalent

Return type
bool

dfaAuPoint()

DFA "au-point" acconding to Nipkow

Returns
"au-point" DFA

Return type
DFA

See also:

Andrea Asperti, Claudio Sacerdoti Coen and Enrico Tassi, Regular Expressions, au point. arXiv 2010

See also:

Tobias Nipkow and Dmitriy Traytel, Unified Decision Procedures for Regular Expression Equivalence

dfaBrzozowski(*memo=None*)

Word derivatives automaton of the regular expression

Parameters
memo – if True memorizes the states already computed

Returns
word derivatives automaton

Return type
DFA

See also:

J. A. Brzozowski, Derivatives of Regular Expressions. J. ACM 11(4): 481-494 (1964)

dfaNaiveFollow()

DFA that accepts the regular expression's language, and is obtained from the follow automaton.

Returns
DFA follow

Return type
NFA

Note: Included for testing purposes.

See also:

Ilie & Yu (Follow Automata, 2003)

dfaYMG()

DFA Yamada-McNaugthon-Gluskov acconding to Nipkow

Returns
Y-M-G DFA

Return type
DFA

See also:

Tobias Nipkow and Dmitriy Traytel, Unified Decision Procedures for Regular Expression Equivalence

static emptysetP()

Whether the regular expression is the empty set.

Return type
bool

abstract static epsilonLength()

Number of occurrences of the empty word in the regular expression.

Returns
number of epsilons

Return type
int

static epsilonP()

Whether the regular expression is the empty word.

Return type
bool

equivP(*other*, *strict=True*)

Test RE equivalence with extended Hopcroft-Karp method

Parameters

- **other** (RegExp) – RE
- **strict** (*bool*) – if True checks for same alphabets

Return type
bool

equivalentP(*other*)

Tests equivalence

Parameters
other (RegExp) – other regexp

Returns
True if regexps are equivalent

Return type
bool

evalWordP(*word*)

Verifies if a word is a member of the language represented by the regular expression.

Parameters
word (*str*) – the word

Returns
True if word belongs to the language

Return type
bool

static ewp()

Whether the empty word property holds for this regular expression's language.

Return type
bool

abstract first()

First set

Returns
first position set

Return type
set

abstract followLists()

Follow set

Returns
for each key position a set of follow positions

Return type
dict

abstract followListsD()

Follow set

Returns
for each key position a set of follow positions

Return type
dict

abstract last()

Last set

Returns
last position set

Return type
set

`abstract linearForm()`

Returns
linear form

Return type
dict

`abstract mark()`

Make all atoms maked (tag False)

Return type
RegExp

`marked()`

Regular expression in which every alphabetic symbol is marked with its Position.

The kind of regular expression returned is known, depending on the literary source, as marked, linear or restricted regular expression.

Returns
linear regular expression

Return type
RegExp

See also:

r. McNaughton and H. Yamada, Regular Expressions and State Graphs for Automata, IEEE Transactions on Electronic Computers, V.9 pp:39-47, 1960

..attention: mark and unmark do not preserve the alphabet, neither set the new alphabet

`nfaFollow()`

NFA that accepts the regular expression's language, whose structure, equiand construction.

Returns
NFA follow

Return type
NFA

See also:

Ilie & Yu (Follow Automata, 03)

`nfaFollowEpsilon(trim=True)`

Epsilon-NFA constructed with Ilie and Yu's method () that accepts the regular expression's language.

Parameters
trim (*bool*) – if True automaton is trimmed at the end

Returns
possibly with epsilon transitions

Return type
NFAe

Note: The regular expression must be reduced

See also:

Ilie & Yu, Follow automta, Inf. Comp. ,v. 186 (1),140-162,2003

nfaGlushkov()

Position or Glushkov automaton of the regular expression. Recursive method.

Returns
NFA position

Return type
NFA

nfaLoc()

Location automaton of the regular expression.

Returns
location nfa

Return type
NFA

nfaNaiveFollow()

NFA that accepts the regular expression's language, and is equal in structure to the follow automaton.

Returns
NFA follow

Return type
NFA

Note: Included for testing purposes.

See also:

Ilie & Yu (Follow Automata, 2003)

nfaPD(*pdmethod='nfaPDDAG'*)

Computes the partial derivative automaton

Parameters
pdmethod (*str*) – an implementation of the PD automaton. Default value : nfaPDDAG

Returns
a PD nfa

Return type
NFA

Attention: for sre classes, CConj and CShuffle use nfaPDNaive directly

nfaPDDAG()

Partial derivative automaton using a DAG for the re and partial derivatives

Returns
a PD nfa build using a DAG

Return type
NFA

..seealso:: s.Konstantinidis, A. Machiavelo, N. Moreira, and r. Reis.
Partial derivative automaton by compressing regular expressions. DCFS 2021, volume 13037 of LNCS, pages 100–112. Springer, 2022

`nfaPDNaive()`

NFA that accepts the regular expression's language,
and which is constructed from the expression's partial derivatives.

Returns
partial derivatives [or equation] automaton

Return type
NFA

See also:

V. M. Antimirov, Partial Derivatives of Regular Expressions and Finite Automaton Constructions .Theor. Comput. Sci.155(2): 291-319 (1996)

`nfaPDO()`

NFA that accepts the regular expression's language, and which is constructed from the expression's partial
derivatives.

Returns
partial derivatives [or equation] automaton

Return type
NFA

Note: optimized version

`nfaPSNF()`

Position or Glushkov automaton of the regular expression constructed from the expression's star normal form.

Returns
Position automaton

Return type
NFA

`nfaPosition(lstar=True)`

Position automaton of the regular expression.

Parameters
lstar (*bool*) – if not None followlists are computed as disjunct

Returns
Position NFA

Return type
NFA

nfaPre()

Prefix NFA of a regular expression

Returns
prefix automaton

Return type
NFA

See also:

Maia et al, Prefix and Right-partial derivative automata, 11th CIE 2015, 258-267 LNCS 9136, 2015

notEmptyW()

Witness of non emptyness

Return type
word or None

abstract rpn()

RPN representation

Returns
printable RPN representation

Return type
str

abstract static setOfSymbols()

Returns
set of symbols

Return type
set

setSigma(*symbolset=None*, *strict=False*)

Set the alphabet for a regular expression and all its nodes

Parameters

- **symbolset** (*set or list*) – accepted symbols. If None, alphabet is unset.
- **strict** (*bool*) – if True checks if setOfSymbols is included in symbolSet

..attention: Normally this attribute is not defined in a RegExp()

abstract snf()

Star Normal Form

abstract static starHeight()

Maximum level of nested regular expressions with a star operation applied.

For instance, starHeight(((a*b)*+b*)*) is 3.

Returns
number of nested star

Return type
int

abstract support(*side=True*)

Set of partial derivatives

abstract tailForm()

Returns
tail form

Return type
dict

toDFA()
DFA that accepts the regular expression's language

toNFA(*nfa_method='nfaPD'*)
NFA that accepts the regular expression's language. :param nfa_method:

abstract static treeLength()
Number of nodes of the regular expression's syntactical tree.

Returns
tree lenght

Return type
int

unionSigma(*other*)
Returns the union of two alphabets

Return type
set

wordDerivative(*word*)

Derivative of the regular expression in relation to the given word,
which is represented by a list of symbols.

Parameters
word (*list*) – list of arbitrary symbols.

Returns
regular expression

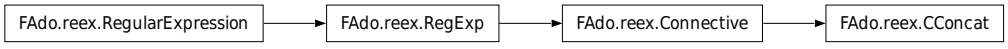
Return type
RegExp

See also:

J. A. Brzozowski, Derivatives of Regular Expressions. J. ACM 11(4): 481-494 (1964)

class RegularExpression
Abstract base class for all regular expression objects

class SConcat(*arg*, *sigma=None*)
Class that represents the concatenation operation.



derivative(*sigma*)

Parameters
sigma –

Returns

ewp()

Returns

head()

Returns

head_rev()

Returns

linearForm()

Returns

linearFormC()

Returns

partialDerivatives(*sigma*)

Parameters
sigma –

Returns

partialDerivativesC(*sigma*)

Parameters
sigma –

Returns

support(*side=True*)

Returns

tail()

Returns

tailForm()

Returns
tail form

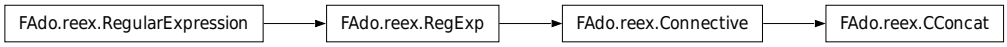
Return type
dict

tail_rev()

Returns

class SConj(*arg*, *sigma=None*)

Class that represents the conjunction operation.



derivative(*sigma*)

Parameters
sigma –

Returns

ewp()

Returns

linearForm()

Returns

partialDerivatives(*sigma*)

Parameters
sigma –

Returns

partialDerivativesC(*sigma*)

Parameters
sigma –

Returns

support(*side=True*)

Returns

tailForm()

Returns
tail form

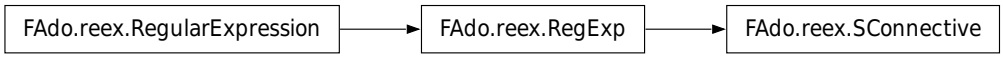
Return type
dict

class SConnective(*arg*, *sigma=None*)

Special regular expressions modulo associativity, commutativity, idempotence of disjunction and intersection;

associativity of concatenation; identities sigma^* and sigma^+. Connectives are:
SDisj: disjunction SConj: intersection SConcat: concatenation

For parsing use str2sre



alphabeticLength()

Returns

epsilonLength()

Returns

first()

First set

Returns
first position set

Return type
set

followLists()

Follow set

Returns
for each key position a set of follow positions

Return type
dict

followListsD()

Follow set

Returns
for each key position a set of follow positions

Return type
dict

last()

Last set

Returns
last position set

Return type
set

linearForm()

Returns
linear form

Return type
dict

mark()

Make all atoms maked (tag False)

Return type
RegExp

nfaPD(*pdmethod='nfaPDNaive'*)

Computes the partial derivative automaton

rpn()

RPN representation

Returns
printable RPN representation

Return type
str

setOfSymbols()

Returns

snf()

Star Normal Form

starHeight()

Maximum level of nested regular expressions with a star operation applied.

For instance, starHeight(((a*b)*+b*)*) is 3.

Returns
number of nested star

Return type
int

abstract support(*side=True*)

Set of partial derivatives

syntacticLength()

Returns

abstract tailForm()

Returns
tail form

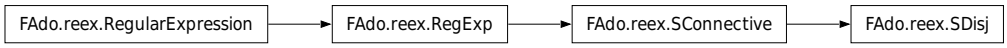
Return type
dict

treeLength()

Returns

class SDisj(*arg*, *sigma=None*)

Class that represents the disjunction operation for special regular expressions.



static cross(*ri*, *s*, *lists*)

Return type
list

derivative(*sigma*)

Parameters
sigma –

Returns

ewp()

Returns

first()

Returns

followLists(*lists=None*)

Parameters
lists –

Returns

followListsStar(*lists=None*)

Parameters
lists –

Returns

last()

Returns

linearForm()

Returns

linearFormC()

Returns

partialDerivatives(*sigma*)

Parameters
sigma –

Returns

partialDerivativesC(*sigma*)

Parameters
sigma –

Returns

support(*side=True*)

Returns

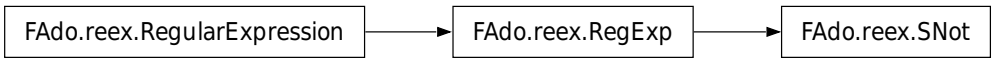
tailForm()

Returns
tail form

Return type
dict

class SNot(*arg*, *sigma=None*)

Special regular expressions modulo associativity, commutativity, idempotence of disjunction and intersection;
associativity of concatenation; identities sigma^* and sigma^+. SNot: negation



alphabeticLength()

Returns

derivative(*sigma*)
:param sigma :return:

epsilonLength()

Returns

ewp()

Returns

first()
First set

Returns
first position set

Return type
set

followLists()
Follow set

Returns
for each key position a set of follow positions

Return type
dict

`followListsD()`

Follow set

Returns
for each key position a set of follow positions

Return type
dict

`last()`

Last set

Returns
last position set

Return type
set

`linearForm()`

Returns

`linearFormC()`

Returns

`mark()`

Make all atoms maked (tag False)

Return type
RegExp

`nfaPD(pdmethod='nfaPDNaive')`

Computes the partial derivative automaton

`partialDerivatives(sigma)`

Parameters
sigma –

Returns

`partialDerivativesC(sigma)`

Parameters
sigma –

Returns

`rpn()`

RPN representation

Returns
printable RPN representation

Return type
str

`setOfSymbols()`

Returns

`snf()`

Star Normal Form

`starHeight()`

Maximum level of nested regular expressions with a star operation applied.

For instance, starHeight(((a*b)*+b*)*) is 3.

Returns
number of nested star

Return type
int

`support(side=True)`

Returns

`syntacticLength()`

Returns

`tailForm()`

Returns
tail form

Return type
dict

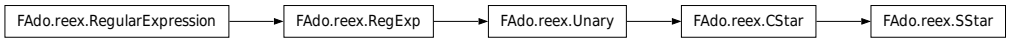
`treeLength()`

Returns

`class SStar(arg, sigma=None)`

Special regular expressions modulo associativity, commutativity, idempotence of disjunction and intersection;
associativity of concatenation; identities sigma^* and sigma^+.

SStar: Class that represents Kleene star



`derivative(sigma)`

Parameters
sigma –

Returns

`linearForm`()

Returns

`nfaPD`(*pdmethod='nfaPDNaive'*)

Computes the partial derivative automaton

`partialDerivatives`(*sigma*)

Parameters
sigma –

Returns

`partialDerivativesC`(*sigma*)

Parameters
sigma –

Returns

`support`(*side=True*)

Returns

`class SpecialConstant`(*sigma=None*)

Base class for Epsilon and EmptySet

Parameters
sigma – alphabet

`static alphabeticLength`()

Returns

`derivative`(*sigma*)

Parameters
sigma –

Returns

`distDerivative`(*sigma*)

Parameters
sigma – an arbitrary symbol.

Return type
regular expression

`static epsilonLength`()

Number of occurrences of the empty word in the regular expression.

Returns
number of epsilons

Return type
int

`static first`(*parent_first=None*)

Parameters
parent_first –

Returns

`followLists`(*lists=None*)

Parameters
lists –

Returns

`followListsD`(*lists=None*)

Parameters
lists –

Returns

`static followListsStar`(*lists=None*)

Parameters
lists –

Returns

`last`(*parent_last=None*)

Parameters
parent_last –

Returns

`linearForm`()

Returns

`mark`()
Make all atoms maked (tag False)

Return type
RegExp

`partialDerivativesC`(*sigma*)

Parameters
sigma –

Returns

`reversal`()
Reversal of RegExp

Return type
reex.RegExp

abstract rpn()

RPN representation

Returns
printable RPN representation

Return type
str

static setOfSymbols()

Returns

snf()

Star Normal Form

static starHeight()

Maximum level of nested regular expressions with a star operation applied.

For instance, starHeight(((a*b)*+b*)*) is 3.

Returns
number of nested star

Return type
int

support(*side=True*)

Returns

supportlast(*side=True*)

Returns

tailForm()

Returns

static treeLength()

Number of nodes of the regular expression's syntactical tree.

Returns
tree lenght

Return type
int

unmark()

Conversion back to unmarked atoms :rtype: SpecialConstant

unmarked()

The unmarked form of the regular expression. Each leaf in its syntactical tree becomes a RegExp(), the CEpsilon() or the CEmptySet().

Return type
(general) regular expression

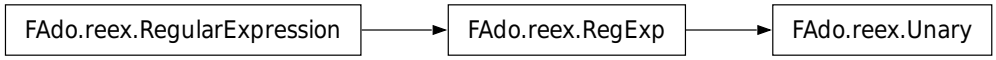
wordDerivative(*word*)

Parameters
word –

Returns

class Unary(*arg*, *sigma=None*)

Base class for unary operations: star, option, not, unary shuffle, etc



alphabeticLength()

Number of occurrences of alphabet symbols in the regular expression. :returns: alphapetic length :rtype: int

> **Attention:** Doesn't include the empty word.

abstract epsilonLength()

Number of occurrences of the empty word in the regular expression.

Returns
number of epsilons

Return type
int

abstract ewp()

Whether the empty word property holds for this regular expression's language.

Return type
bool

abstract first(*parent_first=None*)

First set

Returns
first position set

Return type
set

followLists(*lists=None*)

Follow set

Returns
for each key position a set of follow positions

Return type
dict

followListsD(*lists=None*)

Follow set

Returns
for each key position a set of follow positions

Return type
dict

abstract last(*parent_last=None*)

Last set

Returns
last position set

Return type
set

abstract linearForm()

Returns
linear form

Return type
dict

mark()

Make all atoms maked (tag False)

Return type
RegExp

reversal()

Reversal of RegEx

Return type
reex.RegExp

abstract rpn()

RPN representation

Returns
printable RPN representation

Return type
str

setOfSymbols()

Returns
set of symbols

Return type
set

snf()

Star Normal Form

abstract starHeight()

Maximum level of nested regular expressions with a star operation applied.

For instance, starHeight(((a*b)*+b*)*) is 3.

Returns
number of nested star

Return type
int

treeLength()

Number of nodes of the regular expression's syntactical tree.

Returns
tree lenght

Return type
int

unmark()

Conversion back to RegExp

Return type
reex.__class__

equivalentP(*first*, *second*)

Verifies if the two languages given by some representative (DFA, NFA or re) are equivalent

Parameters

- **first** – language
- **second** – language

Return type
bool

New in version 0.9.6.

powerset(*iterable*)

Powerset of a set. :param iterable: the set :type iterable: list

Returns
the powerset

Return type
itertools.chain

rpn2regexp(*s*, *sigma=None*, *strict=False*)

Reads a (simple) RegExp from a RPN representation

```
r   ::=    .RR | +RR | *r | L | @
L   ::=    [a-z] | [A-Z]
```

Parameters

- **s** (*str*) – RPN representation
- **strict** (*bool*) – Boolean
- **sigma** (*set*) – alphabet

Return type
reex.RegExp

Note: This method uses python stack... thus depth limitations apply

str2regexp(*s*, *parser=Lark(open('/Users/rvr/Work/FAdo/FAdo/regexp_grammar.lark')*, *parser='lalr'*, *lexer='contextual'*, *...)*, *sigma=None*, *strict=False*)

Reads a RegExp from string.

Parameters

- **s** (*string*) – the string representation of the regular expression
- **parser** – a parser generator for regexps
- **sigma** (*list or set of symbols*) – alphabet of the regular expression
- **strict** (*boolean*) – if True tests if the symbols of the regular expression are included in sigma

Return type
reex.RegExp

str2sre(*s*, *parser=Lark(open('/Users/rvr/Work/FAdo/FAdo/regexp_grammar.lark')*, *parser='lalr'*, *lexer='contextual'*, *...)*, *sigma=None*, *strict=False*)

Reads a sre from string. Arguments as str2regexp.

Return type
reex.sre

to_s(*r*)

Returns a sre from FAdo regexp.

Parameters
r (RegExp) – the FAdo representation regexp for a regular expression.

Return type
RegExp

1.6 FAdo.transducers

Finite Tranducer Support

Transducer manipulation.

New in version 1.0.

class GFT

General Form Transducer

addOutput(*sym*)

Add a new symbol to the output alphabet

There is no problem with duplicate symbols because Output is a Set. No symbol Epsilon can be added

Parameters
sym (*str*) – symbol or regular expression to be added

addTransition(*stsrc*, *wi*, *wo*, *sti2*)

Adds a new transition

Parameters

- **stsrc** (*int*) – state index of departure
- **sti2** (*int*) – state index of arrival
- **wi** (*str*) – word consumed
- **wo** (*str*) – word outputed

codeOfTransducer()

Appends into one string the codes of the alphabets and initial and final state sets and the set of transitions

Return type
tuple

listOfTransitions()

Collects into a sorted list the transitions of the transducer.

Return type
set of tuples

toSFT()

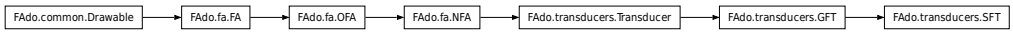
Conversion to an equivalent SFT

rtype: SFT

class NFT

Normal Form Transducer.

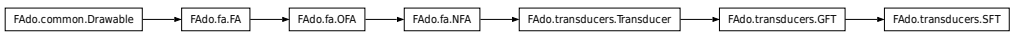
Transsitions here have labels of the form (s,Epsilon) or (Epsilon,s)



class SFT

Standard Form Tranducer

Variables
Output (*set*) – output alphabet



addEpsilonLoops()

Add a loop transition with epsilon input and output to every state in the transducer.

addTransition(*stsrc*, *symi*, *symo*, *sti2*)

Adds a new transition

Parameters

- **stsrc** (*int*) – state index of departure
- **sti2** (*int*) – state index of arrival
- **symi** (*str*) – symbol consumed
- **symo** (*str*) – symbol output

addTransitionProductQ(*src*, *dest*, *ddest*, *sym*, *out*, *futQ*, *pastQ*)

Add transition to the new transducer instance.

Version for the optimized product

Parameters

- **src** – source state
- **dest** – destination state
- **ddest** – destination as tuple
- **sym** – symbol
- **out** – output
- **futQ** (*set*) – queue for later
- **pastQ** (*set*) – past queue

addTransitionQ(*src*, *dest*, *sym*, *out*, *futQ*, *pastQ*)

Add transition to the new transducer instance.

Parameters

- **src** – source state
- **dest** – destination state
- **sym** – symbol
- **out** – output
- **futQ** (*set*) – queue for later
- **pastQ** (*set*) – past queue

composition(*other*)

Composition operation of a transducer with a transducer.

Parameters
other (SFT) – the second transducer

Return type
SFT

concat(*other*)

Concatenation of transducers

Parameters
other (SFT) – the other operand

Return type
SFT

delTransition(*sti1*, *sym*, *symo*, *sti2*, *_no_check=False*)

Remove a transition if existing and perform cleanup on the transition function's internal data structure.

Parameters

- **symo** – symbol output
- **sti1** (*int*) – state index of departure
- **sti2** (*int*) – state index of arrival
- **sym** – symbol consumed
- **_no_check** (*bool*) – dismiss secure code

deleteState(*sti*)

Remove given state and transitions related with that state.

Parameters
sti (*int*) – index of the state to be removed

Raises
DFAstateUnknown – if state index does not exist

deleteStates(*lstates*)

Delete given iterable collection of states from the automaton.

Parameters
lstates (*set*|*list*) – collection of int representing states

dup()

Duplicate of itself :rtype: SFT

> **Attention:** only duplicates the initially connected component

emptyP()

Tests if the relation realized the empty transducer

Return type
bool

epsilonOutP()

Tests if epsilon occurs in transition outputs

Return type
bool

epsilonP()

Test whether this transducer has input epsilon-transitions

Return type
bool

evalWordP(*wp*)

Tests whether the transducer returns the second word using the first one as input

Parameters
wp (*tuple*) – pair of words

Return type
bool

evalWordSlowP(*wp*)

Tests whether the transducer returns the second word using the first one as input

Note: original :param tuple wp: pair of words :rtype: bool

functionalP()

Tests if a transducer is functional using Allauzer & Mohri and Béal&Carton&Prieur&Sakarovitch algorithms.

Return type
bool

See also:

Cyril Allauzer and Mehryar Mohri, Journal of Automata Languages and Combinatorics, Efficient Algorithms for Testing the Twins Property, 8(2): 117-144, 2003.

See also:

M.P. Béal, O. Carton, C. Prieur and J. Sakarovitch. Squaring transducers: An efficient procedure for deciding functionality and sequentiality. Theoret. Computer Science 292:1 (2003), 45-63.

Note: This is implemented using nonFunctionalW()

---

inIntersection(*other*)

Conjunction of transducer and automata: X & Y.

Note: This is a fast version of the method that does not produce meaningfull state names.

Note: The resulting transducer is not trim.

Parameters
other (DFA/NFA) – the automata needs to be operated.

Return type
SFT

inIntersectionSlow(*other*)

Conjunction of transducer and automata: X & Y.

Note: This is the slow version of the method that keeps meaningfull names of states.

Parameters
other (DFA/NFA) – the automata needs to be operated.

Return type
SFT

`inverse()`

Switch the input label with the output label.

No initial or final state changed.

Returns
Transducer with transitions switched.

Return type
SFT

`nonEmptyW()`

Witness of non emptyness

Returns
pair (in-word, out-word)

Return type
tuple

`nonFunctionalW()`

Returns a witness of non funcionality (if is that the case) or a None filled triple

Returns
witness

Return type
tuple

`outIntersection(other)`

Conjunction of transducer and automaton: X & Y using output intersect operation.

Parameters
other (DFA/NFA) – the automaton used as a filter of the output

Return type
SFT

`outIntersectionDerived(other)`

Naive version of outIntersection

Parameters
other (DFA/NFA) – the automaton used as a filter of the output

Return type
SFT

`outputS(s)`

Output label coming out of the state i

Parameters
s (*int*) – index state

Return type
set

productInput(*other*)

Returns a transducer (skeleton) resulting from the execution of the transducer with the automaton as filter on the input.

Note: This version does not use stateIndex() with the price of generating some unreachable sates

Parameters
other (NFA) – the automaton used as filter

Return type
SFT

Changed in version 1.3.3.

productInputSlow(*other*)

Returns a transducer (skeleton) resulting from the execution of the transducer with the automaton as filter on the input.

Note: This is the slow version of the method that keeps meaningfull names of states.

Parameters
other (NFA) – the automaton used as filter

Return type
SFT

reversal()

Returns a transducer that recognizes the reversal of the relation.

Returns
Transducer recognizing reversal language

Return type
SFT

runOnNFA(*nfa*)

Result of applying a transducer to an automaton

Parameters
nfa (DFA / NFA) – input language to transducer

Returns
resulting language

Return type
NFA

runOnWord(*word*)

Returns the automaton accepting the outup of the transducer on the input word

Parameters
word – the word

Return type
NFA

setInitial(*sts*)

Sets the initial state of a Transducer

Parameters
sts (*list*) – list of states

square()

Conjunction of transducer with itself

Return type
NFA

square_fv()

Conjunction of transducer with itself (Fast Version)

Return type
NFA

star(*flag=False*)

Kleene star

Parameters
flag (*bool*) – plus instead of star

Returns
the resulting Transducer

Return type
SFT

toInNFA()

Delete the output labels in the transducer. Translate it into an NFA

Return type
NFA

toNFT()

Transformation into Nomal Form Transducer

Return type
NFT

toOutNFA()

Returns the result of considering the output symbols of the transducer as input symbols of a NFA (ignoring the input symbol, thus)

Returns
the NFA

Return type
NFA

toSFT()

Pacifying rule

Return type
SFT

trim()

Remove states that do not lead to a final state, or, inclusively, that can't be reached from the initial state. Only useful states remain.

Attention: in place transformation

union(*other*)

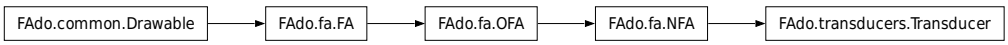
Union of the two transducers

Parameters
other (SFT) – the other operand

Return type
SFT

class Transducer

Base class for Transducers



setOutput(*listOfSymbols*)

Set Output

Parameters
listOfSymbols (*set* | *list*) – output symbols

succintTransitions()

Collects the transition information in a concat way suitable for graphical representation. :rtype: list of tupples

exception ZERO

Simple exception for functionality testing algorithm

concatN(*x*, *y*)

Concatenation of tuples of words :param x: iterable :param y: iterable :return: iterable

hypercodeTransducer(*alphabet*, *preserving=False*)

Creates an hypercode property transducer based on given alphabet

Parameters

- **preserving** (*bool*) – input preserving transducer, else input altering
- **alphabet** (*list* | *set*) – alphabet

Return type
SFT

infixTransducer(*alphabet*, *preserving=False*)

Creates an infix property transducer based on given alphabet

Parameters

- **preserving** (*bool*) – input preserving transducer, else input altering
- **alphabet** (*list* | *set*) – alphabet

Return type
SFT

isLimitExceed(*NFA0Delta*, *NFA1Delta*)

Decide if the size of NFA0 and NFA1 exceed the limit.

Size of NFA0 is denoted as N, and size of NFA1 is denoted as M. If N*N*M exceeds 1000000, return False, else return True. If bothNFA is False, then NFA0 should be NFA, and NFA1 should be Transducer. If both NFA is True, then NFA0 and NFA1 are both NFAs.

Parameters

- **NFA0Delta** (*dict*) – NFA0's transition Delta
- **NFA1Delta** (*dict*) – NFA1's transition Delta

Return type
bool

outfixTransducer(*alphabet*, *preserving=False*)

Creates an outfix property transducer based on given alphabet

Parameters

- **preserving** (*bool*) – input preserving transducer, else input altering
- **alphabet** (*list* | *set*) – alphabet

Return type
SFT

prefixTransducer(*alphabet*, *preserving=False*)

Creates an prefix property transducer based on given alphabet

Parameters

- **preserving** (*bool*) – input preserving transducer, else input altering
- **alphabet** (*list* | *set*) – alphabet

Return type
SFT

suffixTransducer(*alphabet*, *preserving=False*)

Creates an suffix property transducer based on given alphabet

Parameters

- **preserving** (*bool*) – input preserving transducer, else input altering
- **alphabet** (*list* | *set*) – alphabet

Return type
SFT

1.7 FAdo.fl

Finite languages and related automata manipulation

Finite languages manipulation

class ADFA

Acyclic Deterministic Finite Automata class

Changed in version 1.3.3.

addSuffix(*st*, *w*)

Adds a suffix starting in st

Parameters

- **st** (*int*) – state
- **w** (*Word*) – suffix

New in version 1.3.3.

Attention: in place transformation

complete(*dead=None*)

Make the ADFA complete

Parameters

dead (*int, optional*) – a state to be identified as dead state if one was not identified yet

Attention: The object is modified in place

Changed in version 1.3.3.

diss()

Evaluates the dissimilarity language

Return type

FL

New in version 1.2.1.

dissMin(*witnesses=None*)

Evaluates the minimal dissimilarity language

Parameters

witnesses (*dict*) – optional witness dictionay

Return type
FL

New in version 1.2.1.

`dup()`

Duplicate the basic structure into a new ADFA. Basically a copy.deep.

Return type
ADFA

`forceToDFA()`

Conversion to DFA

Return type
DFA

`forceToDFCA()`

Conversion to DFCA

Return type
DFA

`level()`

Computes the level for each state

Returns
levels of states

Return type
dict

New in version 0.9.8.

`minDFCA()`

Generates a minimal deterministic cover automata from a DFA

Return type
DCFA

New in version 0.9.8.

See also:

Cezar Campeanu, Andrei Päun, and Sheng Yu, An efficient algorithm for constructing minimal cover automata for finite languages, IJFCS

`minReversible()`

Returns the minimal reversible equivalent automaton

Return type
ADFA

`minimal()`

Finds the minimal equivalent ADFA

Returns
the minimal equivalent ADFA

Return type
DFA

See also:

[TCS 92 pp 181-189] Minimisation of acyclic deterministic automata in linear time, Dominique Revuz

Changed in version 1.3.3.

`minimalP`(*method=None*)

Tests if the DFA is minimal

Parameters
method (*str*) – minimization algorithm (here void)

Return type
bool

Changed in version 1.3.3.

`possibleToReverse`()

Tests if language is reversible

New in version 1.3.3.

`statePairEquiv`(*s1*, *s2*)

Tests if two states of a ADFA are equivalent

Parameters

- **s1** (*int*) – state1
- **s2** (*int*) – state2

Return type
bool

New in version 1.3.3.

`toANFA`()

Converts the ADFA in a equivalent ANFA

Return type
ANFA

`toNFA`()

Converts the ADFA in a equivalent NFA

Return type
ANFA

New in version 1.2.

`trim`()

Remove states that do not lead to a final state, or, inclusively, that can't be reached from the initial state. Only useful states remain.

Attention: in place transformation

`wordGenerator`()

Creates a random word generator

Returns
the random word generator

Return type
RndWGen

New in version 1.2.

class AFA

Base class for Acyclic Finite Automata

note: This is just a container for some common methods. **Not to be used directly!!**

abstract addState(_)

Return type
int

directRank()

Compute rank function

Returns
rank map

Return type
dict

ensureDead()

Ensures that a state is defined as dead

evalRank()

Evaluates the rank map of a automaton

Returns
pair of sets of states by rank map, reverse delta accessability map

Return type
tuple

getLeaves()

The set of leaves, i.e. final states for last symbols of language words

Returns
A set of leaves

Return type
set

ordered()

Orders states names in its topological order

Returns
ordered list of state indexes

Return type
list of int

Note: one could use the FA.toposort() method, but special care must be taken with the dead state for the algorithms related with cover automata.

`setDeadState`(*sti*)

Identifies the dead state

Parameters
sti (*string*) – index of the dead state

Attention: nothing is done to ensure that the state given is legitimate

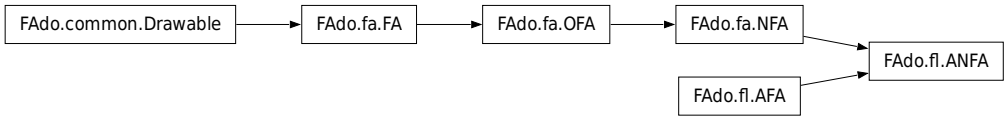
---

Note: without dead state identified, most of the methods for acyclic automata can not be applied

---

`class ANFA`

Acyclic Nondeterministic Finite Automata class



`mergeInitial`()

Merge initial states

Attention: object is modified in place

`mergeLeaves`()

Merge leaves

Attention: object is modified in place

`mergeStates`(*s1*, *s2*)

Merge state s2 into state s1

Parameters

- **s1** (*int*) – state index
- **s2** (*int*) – state index

Note: no attempt is made to check if the merging preserves the language of teh automaton

Attention: the object is modified in place

moveFinal(*st*, *stf*)

Unsets a set as final transfering transition to another final :param int st: the state to be 'moved' :param int stf: the destination final state

Note: stf must be a 'last' final state, i.e., must have no out transitions to anywhere but to a possible dead state

Attention: the object is modified in place

class BitString(*blksz*, *alphsize*, *bst=None*)

Class to represent the bitstring of a block language

Variables

- **blocksize** – the size of the block
- **alphsize** – the size of the alphabet
- **bst** – the bitstring representation of the language

reverse()

Compute the BitString representation of the reverse of the current language.

class BlockWords(*k: int*, *b: int*)

Block language iterator

DFAtoADFA(*aut*)

Transforms an acyclic DFA into a ADFA

Parameters
aut (DFA) – the automaton to be transformed

Returns
the converted automaton

Return type
ADFA

Raises
notAcyclic – if the DFA is not acyclic

class DFCA

Deterministic Cover Automata class

property length

The length of the longest word :returns: the length of the longest word :rtype: int

class FL(*wordsList=None*, *Sigma=None*)

Finite Language Class

Variables

- **Words** – the elements of the language
- **Sigma** – the alphabet

MADFA()

Generates the minimal acyclical DFA using specialized algorithm

New in version 1.3.3.

See also:

Incremental Construction of Minimal Acyclic Finite-State Automata, J.Daciuk, blksz.Mihov, B.Watson and r.E.Watson

Return type
ADFA

addWord(*word*)

Adds a word to a FL

Parameters
word (*string*) – word to be added

Return type
FL

addWords(*wList*)

Adds a list of words to a FL

Parameters
wList (*list*) – words to be added

diff(*other*)

Difference of FL: a - b

Parameters
other (FL) – left hand operand

Returns
result of the operation

Return type
FL

Raises
FAdoGeneralError – if both arguments are not FL

`filter`(*automata*)

Separates a language in two other using a DFA of NFA as a filter

Parameters
automata (*dict*) – the automata to be used as a filter

Returns
the accepted/unaccepted pair of languages

Return type
tuple of *FL*

`intersection`(*other*)

Intersection of FL: a & b

Parameters
other (FL) – left hand operand

Returns
result of the operation

Return type
FL

Raises
FAdoGeneralError – if both arguments are not FL

`multiLineAutomaton`()

Generates the trivial linear ANFA equivalent to this language

Returns
the trivial linear ANFA

Return type
ANFA

`setSigma`(*Sigma*, *Strict=False*)

Sets the alphabet of a FL

Parameters

- **Sigma** (*string*) – alphabet
- **Strict** (*bool*) – behaviour

Attention: Unless Strict flag is set to True, alphabet can only be enlarged. The resulting alphabet is in fact the union of the former alphabet with the new one. If flag is set to True, the alphabet is simply replaced.

`suffixClosedP`()

Tests if a language is suffix closed

Returns
True if language is suffix closed

Return type
bool

toDFA()

Generates a DFA recognizing the language

Returns
the DFA

Return type
ADFA

New in version 1.2.

toNFA()

Generates a NFA recognizing the language

Return type
ANFA

New in version 1.2.

trieFA()

Generates the trie automaton that recognises this language

Returns
the trie automaton

Return type
ADFA

union(*other*)

union of FL: a | b

Parameters
other (FL) – right hand operand

Returns
result of the union

Return type
FL

Raises
FAdoGeneralError – if both arguments are not FL

class RndWGen(*aut*)

Word random generator class

New in version 1.2.

Parameters
aut (ADFA) – automata recognizing the language

blockUniversalP(*a*, *n*)

Parameters

- **a** (NFA) – blksz NFA (= NFA accepting only words of same length)
- **n** (*int*) – length of accepted words

Returns
whether a is blksz universal (accepts all words of length n)

Return type
bool

coBlockDFA(*a:* DFA, *n: int*) → *DFA*

Parameters

- **a** (DFA) – automaton accepting fixed length words
- **n** (*int*) – length of words accepted, n > 0

Returns
accepts the words of length n not accepted by a

Return type
DFA

New in version 2.1.2.

firstBlockWords(*alpzs: int*, *nwords: int*, *blksz: int*) → *ADFA*

Generates the minimal ADFA that accepts exactly the first nwords (lexicographic order) of a blksz language

Parameters

- **alpzs** (*int*) – alphabet size
- **nwords** (*int*) – number of words
- **blksz** (*int*) – blksz size

Returns
the ADFA that recognises exacly those words

Return type
ADFA

genRandomTrie(*maxL*, *Sigma*, *safe=True*)

Generates a random trie automaton for a finite language with a given length for max word

Parameters

- **maxL** (*int*) – length of the max word
- **Sigma** (*set*) – alphabet to be used
- **safe** (*bool*) – should a word of size maxl be present in every language?

Returns
the generated trie automaton

Return type
ADFA

genRndBitString(*b: int*, *k: int*) → *BitString*

Generates a random bitstring with alphabet size k and block size b

Parameters

- **b** (*int*) – The size of the block
- **k** (*int*) – The size of the alphabet

Returns
the random bitstring

Return type
bitarray

genRndTrieBalanced(*maxL*, *Sigma*, *safe=True*)

Generates a random trie automaton for a binary language of balanced words of a given leght for max word

Parameters

- **maxL** (*int*) – the length of the max word
- **Sigma** (*set*) – the alphabet to be used
- **safe** (*bool*) – should a word of size maxl be present in every language?

Returns
the generated trie automaton

Return type
ADFA

genRndTriePrefix(*maxL*, *Sigma*, *ClosedP=False*, *safe=True*)

Generates a random trie automaton for a finite (either prefix free or prefix closed) language with a given length for max word

Parameters

- **maxL** (*int*) – length of the max word
- **Sigma** (*set*) – alphabet to be used
- **ClosedP** (*bool*) – should it be a prefix closed language?
- **safe** (*bool*) – should a word of size maxl be present in every language?

Returns
the generated trie automaton

Return type
ADFA

genRndTrieUnbalanced(*maxL*, *Sigma*, *ratio*, *safe=True*)

Generates a random trie automaton for a binary language of balanced words of a given length for max word

Parameters

- **maxL** (*int*) – length of the max word
- **Sigma** (*set*) – alphabet to be used
- **ratio** (*int*) – the ratio of the unbalance
- **safe** (*bool*) – should a word of size maxl be present in every language?

Returns
the generated trie automaton

Return type
ADFA

generateBlockTrie(*sz: int*, *alpsz: int*) → *ADFA*

Generates a trie for a blksz language

Parameters

- **sz** (*int*) – size of the blksz
- **alpsz** (*int*) – size of the alphabet

Returns
the automaton

Return type
ADFA

New in version 2.1.3.

`sigmaInitialSegment`(*Sigma: list*, *l: int*, *exact=False*) → *ADFA*

Generates the ADFA recognizing Sigma^i for i<=l

Parameters

- **Sigma** (*list*) – the alphabet
- **l** (*int*) – length
- **exact** (*bool*) – only the words with exactly that length?

Returns
the automaton

Return type
ADFA

`stringToADFA`(*s*)

Convert a canonical string representation of a ADFA to a ADFA

Parameters
s (*str*) – the string in its canonical order

Returns
the ADFA

Return type
ADFA

See also:

Marco Almeida, Nelma Moreira, and Rogério Reis. Exact generation of minimal acyclic deterministic finite automata. International Journal of Foundations of Computer Science, 19(4):751-765, August 2008.

1.8 FAdo.cgf

Context Free Grammars Manipulation.

Basic context-free grammars manipulation for building uniform random generetors

`class CFGGenerator`(*cfgr*, *size*)

CFG uniform genetaror

..seealso: Generating words in a context-free language uniformly at random. Harry Mairson
Information Processing Letters, 49-2, 95-92. 1994

Object initialization :param cfgr: grammar for the random objects :type cfgr: CNF :param size: size of objects :type size: integer

`generate`()

Generates a new random object generated from the start symbol

Returns
object

Return type
string

class CFGrammar(*gram*)

Class for context-free grammars

Variables

- **Rules** – grammar rules
- **Terminals** – terminals symbols
- **Nonterminals** – nonterminals symbols
- **Start** (*str*) – start symbol
- **ntr** – dictionary of rules for each nonterminal

Initialization

Parameters
gram – is a list for productions; each production is a tuple (LeftHandside, RightHandside) with LeftHandside nonterminal, RightHandside list of symbols, First production is for start symbol

NULLABLE()

Determines which nonterminals X ->* []

makenonterminals()

Extracts C{nonterminals} from grammar rules.

maketerminals()

Extracts C{terminals} from the rules. Nonterminals must already exist

class CNF(*gram*, *mark='A@'*)

Chomsky Normal Form. No useless nonterminals or eepsipsilon rules are ALLOWED... Given a CFG grammar description generates one in CNF Then its possible to random generate words of a given size. Before some pre-calculations are needed.

Initialization

Parameters
gram – is a list for productions; each production is a tuple (LeftHandside, RightHandside) with LeftHandside nonterminal, RightHandside list of symbols, First production is for start symbol

chomsky()

Transform to CNF

elim_unitary()

Elimination of unitary rules

CYKParserTable(*gramm*, *word*)

Evaluates CYK parser table

Parameters

- **gramm** (CNF) – grammar
- **word** (*str*) – word to be parsed

Returns
the CYK table

Return type
list of lists of symbols

class REStringRGenerator(*Sigma=None*, *size=10*, *cfgr=None*, *epsilon=None*, *empty=None*, *ident='Ti'*)

Uniform random Generator for reStrings

Uniform random generator for regular expressions. Used without arguments generates an uncollapsible re

over {a,b} with size 10. For generate an arbitary re over an alphabet of 10 symbols of size 100: reStringRGenerator (smallAlphabet(10),100,reGrammar["g_regular_base"])

Parameters

- **Sigma** (*list* | *set*) – re alphabet (that will be the set of grammar terminals)
- **size** (*int*) – word size
- **cfgr** – base grammar
- **epsilon** – if not None is added to a grammar terminals
- **empty** – if not None is added to a grammar terminals

Note: the grammar can have already this symbols

gRules(*rules_list*, *rulesym='->'*, *rhssep=None*, *rulesep='|'*)

Transforms a list of rules into a grammar description.

Parameters

- **rules_list** (*list*) – is a list of rule where rule is a string of the form: Word rulesym Word1 ... Word2 or Word rulesym []
- **rulesym** (*str*) – LHS and RHS rule separator
- **rhssep** (*str*) – RHS values separator (None for white chars)
- **rulesep** (*str*) – rule separator

Returns
a grammar description

Return type
list

smallAlphabet(*k*, *sigma_base='a'*)

Easy way to have small alphabets

Parameters

- **k** – alphabet size (must be less than 52)
- **sigma_base** – initial symbol

Returns
alphabet

Return type
list

1.9 FAdo.rndfap

Random DFA generation (alternative version in python)

ICDFA Random generation binding

New in version 1.0.

class ICDFArgen(*n*, *k*, *nd=False*, *pn=1*, *seed=0*)

Generic ICDFA random generator class

Variables

- **n** (*int*) – number of states
- **k** (*int*) – size of the alphabet
- **pn** (*int*) – how more problable shall a non defined transition be?
- **seed** (*int*) – seed for the random generator. Default is to generate a time & system dependent.

See also:

Marco Almeida, Nelma Moreira, and Rogério Reis. Enumeration and generation with a string automata representation. Theoretical Computer Science, 387(2):93-102, 2007

Changed in version 1.3.4: seed added to the random generator

genFinalities()

Generate bit map of final states

Return type
list

class ICDFArnd(*n*, *k*, *seed=0*)

Complete ICDFA random generator class

This is the class for the uniform random generator for Initially Connected DFAs

Variables

- **n** (*int*) – number of states
- **k** (*int*) – size of alphabet
- **seed** (*int*) – seed for the random generator (if 0 uses time as seed)

Note: This is an abstract class, not to be used directly

Changed in version 1.3.4: seed added to the random generator

class ICDFArndIncomplete(*n*, *k*, *bias=None*, *seed=0*)

Incomplete ICDFA random generator class

Variables

- **n** (*int*) – number of states
- **k** (*int*) – size of alphabet
- **bias** (*float*) – how often must the gost sink state appear (default None)
- **seed** (*int*) – seed for the random generator (if 0 uses time as seed)

Raises
IllegalBias – if a bias >=1 or <=0 is provided

Changed in version 1.3.4: seed added to the random generator

1.10 FAdo.rndadfa

Random ADFA generation

ADFA Random generation binding

New in version 1.2.1.

`class ADFArnd(n, k=2, s=1)`

Sets a random generator for Adfas by sources. By default, s=1 to be initially connected

Variables

- **n** (*int*) – number of states
- **k** (*int*) – size of the alphabet
- **s** (*int*) – number of sources

Note: For ICDFA s=1

`alpha(n, s, k=2)`

Number of labeled acyclic initially connected DFA by states and by sources

Parameters

- **k** (*int*) – alphabet size
- **n** (*int*) – number of states
- **s** (*int*) – number of souces

Return type
int

Note: uses countAdfabySource

`alpha0(n, s, k=2)`

Number of labeled acyclic initially connected DFA by states and by sources

Parameters

- **k** (*int*) – alphabet size
- **n** (*int*) – number of states
- **s** (*int*) – number of souces

Return type
int

Note: uses gamma instead of beta or rndAdfa

---

beta(*n*, *s*, *u*, *k=2*)

Number of valid configurations of transitions

Parameters

- **k** (*int*) – alphabet size
- **n** (*int*) – number of states
- **s** (*int*) – number of souces
- **u** (*int*) – number of souces of n-s

Return type
int

Note: not used by alpha or rndAdfa

---

beta0(*n*, *s*, *u*, *k=2*)

Function beta computed using sets

countAdfaBySources(*n*, *s*, *k=2*)

Number of labelled (initially connected) acyclic automata with n states, alphabet size k, and s sources

Parameters

- **k** (*int*) – alphabet size
- **n** (*int*) – number of states
- **s** (*int*) – number of souces

Raises
IndexError – if number of states less than number of sources

gamma(*t*, *u*, *r*)

Parameters

- **t** (*int*) – size of T
- **u** (*int*) – size of U
- **r** (*int*) – size of r

Return type
int

rndAdfa(*n*, *s*)

Recursively generates a initially connected adfa

Parameters

- **n** (*int*) – number of states
- **s** (*int*) – number of sources

See also:

Felice & Nicaud, CSR 2013 Lncs 7913, pp 88-99, Random Generation of Deterministic Acyclic Automata Using the Recursive Method, DOI:10.1007/978-3-642-38536-0_8

rndNumberSecondSources(*n*, *s*)

Uniformaly random generates the number of secondary sources

Parameters

- **n** (*int*) – number of states
- **s** (*int*) – number of sources

Return type
int

rndTransitionsFromSources(*n*, *s*, *u*)

Generates the transitions from the sources, ensuring that all secondary sources are connected

Parameters

- **n** (*int*) – number of states
- **s** (*int*) – number of sources
- **u** (*int*) – number of secondary sources

binomial(*n*, *k*)

Binomial coeficient

Parameters

- **n** – n
- **k** – k

countadfa(*n: int*, *k: int*)

Acyclic (complete) deterministic finite automata structure (unlabeled)

Parameters

- **n** ((*int*)) – number of states
- **k** ((*int*)) – alphabetic size

countadfaL(*n*, *k*)

Acyclic (complete) deterministic finite automata structure (labeled) initially connected (one dead state)

Parameters

- **n** (*int*) – number of states
- **k** (*int*) – alphabetic size

countafa(*n*, *k*)

(Quasi) Acyclic deterministic finite automata structure (unlabeled)

countafaL(*n*, *k*, *r=1*)

(Quasi) Acyclic deterministic finite automata structure (labeled)

Parameters

- **n** (*int*) – number of states
- **k** (*int*) – alphabetic size
- **r** (*int*) – number of dead states

See also:

V. A. Liskovets. Exact enumeration of acyclic deterministic automata. Discrete Applied Mathematics, 154(3):537-551, March 2006.

rndBlockADFA(*alpzs: int*, *nwords: int*, *blksz: int*) → *ADFA*

Random generation of a block language

Parameters

- **alpzs** (*int*) – alphabet size
- **nwords** (*int*) – desired number of words
- **blksz** (*int*) – desired size of the block

Returns
the random block language

Return type
AFDA

New in version 2.1.3.

surj(*n*, *m*)

Counting surgections from [n] to [m]

Parameters

- **n** (*int*) – cardinality of domain
- **m** (*int*) – cardinality of image

Note: not used by rndAdfa

1.11 FAdo.comboperations

Several combined operations for DFAs

Combined operations

concatWStar(*fa1*, *fa2*, *strict=False*)

Concatenation combined with star: (L1.L2*)

Parameters

- **fa1** (DFA) – first automaton
- **fa2** (DFA) – second automaton
- **strict** (*bool*) – should the alphabets be necessary equal?

Return type
DFA

See also:

Bo Cui, Yuan Gao, Lila Kari, and Sheng Yu. 'State complexity of two combined operations: Reversal-catenation and star-catenation'. CoRR, abs/1006.4646, 2010.

disjWStar(*f1*, *f2*, *strict=True*)

Union with Star: (L1 + L2*)

Parameters

- **f1** (DFA) – first automaton
- **f2** (DFA) – second automaton
- **strict** (*bool*) – should the alphabets be necessary equal?

Return type
DFA

See also:

Yuan Gao and Sheng Yu. 'State complexity of union and intersection combined with Star and reversal'. CoRR, abs/1006.3755, 2010.

interWStar(*f1*, *f2*, *strict=True*)

Intersection with Star: (L1 & L2*)

Parameters

- **f1** (DFA) – first automaton
- **f2** (DFA) – second automaton
- **strict** (*bool*) – should the alphabets be necessary equal?

Return type
DFA

See also:

Yuan Gao and Sheng Yu. 'State complexity of union and intersection combined with Star and reversal'. CoRR, abs/1006.3755, 2010.

starConcat(*fa1*, *fa2*, *strict=False*)

Star of concatenation of two languages: (L1.L2)*

Parameters

- **fa1** (DFA) – first automaton
- **fa2** (DFA) – second automaton
- **strict** (*bool*) – should the alphabets be necessary equal?

Return type
DFA

See also:

Yuan Gao, Kai Salomaa, and Sheng Yu. 'The state complexity of two combined operations: star of catenation and star of reversal'. Fundamenta Informaticae, 83:75–89, Jan 2008.

starDisj(*fa1*, *fa2*, *strict=False*)

Star of Union of two DFAs: (L1 + L2)*

Parameters

- **fa1** (DFA) – first automaton
- **fa2** (DFA) – second automaton
- **strict** (*bool*) – should the alphabets be necessary equal?

Return type
DFA

See also:

Arto Salomaa, Kai Salomaa, and Sheng Yu. 'State complexity of combined operations'. Theor. Comput. Sci., 383(2-3):140–152, 2007.

`starInter`(*fa1*, *fa2*, *strict=False*)

Star of Intersection of two DFAs: (L1 & L2)*

Parameters

- **fa1** (DFA) – first automaton
- **fa2** (DFA) – second automaton
- **strict** (*bool*) – should the alphabets be necessary equal?

Return type
DFA

`starInter0`(*fa1*, *fa2*, *strict=False*)

Star of Intersection of two DFAs: (L1 & L2)*

Parameters

- **fa1** (DFA) – first automaton
- **fa2** (DFA) – second automaton
- **strict** (*bool*) – should the alphabets be necessary equal?

Return type
DFA

See also:

Arto Salomaa, Kai Salomaa, and Sheng Yu. 'State complexity of combined operations'. Theor. Comput. Sci., 383(2-3):140–152, 2007.

`starWConcat`(*fa1*, *fa2*, *strict=False*)

Star combined with concatenation: (L1*.L2)

Parameters

- **fa1** (DFA) – first automaton
- **fa2** (DFA) – second automaton
- **strict** (*bool*) – should the alphabets be necessary equal?

Return type
DFA

See also:

Bo Cui, Yuan Gao, Lila Kari, and Sheng Yu. 'State complexity of catenation combined with Star and reversal'. CoRR, abs/1008.1648, 2010

1.12 FAdo.codes

Code theory module

New in version 1.0.

class CodeProperty(*name*, *alph*)

See: K. Dudzinski and s. Konstantinidis: Formal descriptions of code properties: decidability, complexity,
implementation. International Journal of Foundations of Computer Science 23:1 (2012), 67–85.

Variables
Sigma – the alphabet

abstract maximalP(*aut*, *U=None*)

Tests if the language is maximal w.r.t. the property

Parameters

- **U** (DFA / NFA) – Universe of permitted words (sigma^* as default)
- **aut** (DFA / NFA) – the automaton

Return type
bool

abstract notMaximalW(*aut*, *U=None*)

Witness of non maximality

Parameters

- **aut** (DFA / NFA) – the automaton
- **U** (DFA / NFA) – Universe of permitted words (sigma^* as default)

Returns
a witness

Return type
str

abstract notSatisfiesW(*aut*)

Return a witness of non-satisfaction of the property by the automaton language

Parameters
aut (DFA / NFA) – the automaton

Returns
word witness tuple

Return type
tuple

abstract satisfiesP(*aut*)

Satisfaction of the property by the automaton language

Parameters
aut (NFA / DFA) – the automaton

Return type
bool

class ErrCorrectProp(*t*)

Error Correcting Property

Constructor :param SFT aut: Input preserving transducer

notMaximalW(*aut*, *U=None*)

Tests if the language is maximal w.r.t. the property

Parameters

- **aut** (DFA / NFA) – the automaton
- **U** (DFA / NFA) – Universe of permitted words (sigma^* as default)

Return type
bool

notSatisfiesW(*aut*)

Satisfaction of the code property by the automaton language

Parameters
aut (DFA / NFA) – the automaton

Return type
tuple

satisfiesP(*aut*)

Satisfaction of the property by the automaton language

See also:

s. Konstantinidis: Transducers and the Properties of Error-Detection, Error-Correction and Finite-Delay Decodability. Journal Of Universal Computer Science 8 (2002), 278-291.

Parameters
aut (DFA / NFA) – the automaton

Return type
bool

ErrDetectProp

alias of *IPTProp*

class FixedProp(*name*, *alph*)

Abstract class for fixed properties

abstract maximalP(*aut*, *U=None*)

Tests if the language is maximal w.r.t. the property

Parameters

- **U** (DFA / NFA) – Universe of permitted words (sigma^* as default)

- **aut** (DFA/NFA) – the automaton

Return type
bool

abstract notMaximalW(*aut*, *U=None*)

Witness of non maximality

Parameters

- **aut** (DFA/NFA) – the automaton
- **U** (DFA/NFA) – Universe of permitted words (sigma^* as default)

Returns
a witness

Return type
str

abstract notSatisfiesW(*aut*)

Test whether the language is a code.

Parameters
aut (DFA/NFA) – the automaton

Returns
two different factorizations of the same word

Return type
tuple of list

abstract satisfiesP(*aut*)

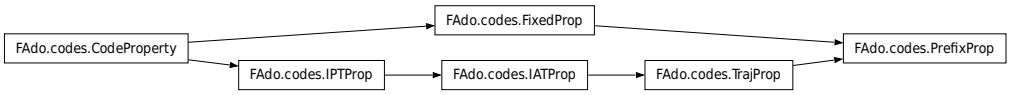
Satisfaction of the property by the automaton language

Parameters
aut (NFA/DFA) – the automaton

Return type
bool

class HypercodeProp(*t*)

Hypercode Property



Constructor

Parameters

- **aut** (DFA/NFA) – regular expression over {0,1}
- **Sigma** (set) – the alphabet

class IATProp(*aut*, *name=None*)

Input Altering Transducer Property

Constructor :param SFT aut: Input preserving transducer

notSatisfiesW(*aut*)

Return a witness of non-satisfaction of the property by the automaton language

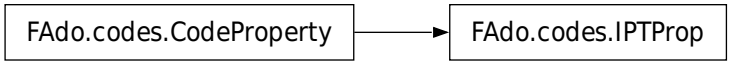
Parameters
aut (DFA/NFA) – the automaton

Returns
word witness pair

Return type
tuple

class IPTProp(*aut*, *name=None*)

Input Preserving Transducer Property



Variables

- **Aut** (SFT) – the transducer defining the property
- **sigma** (*set*) – alphabet

Constructor :param SFT aut: Input preserving transducer

addToCode(*aut*, *N*, *n=2000*)

Returns an NFA and a list W of up to N words of length ell, such that the NFA accepts L(aut) union W, which is an error-detecting language. ell is computed from aut

Parameters

- **aut** (NFA) – the automaton
- **N** (*int*) – the number of words to construct
- **n** (*int*) – number of tries when needing a new word

Returns
an automaton and a list of strings

Return type
tuple

makeCode(*N*, *ell*, *s*, *n=2000*, *ov_free=False*)

Returns an NFA and a list W of up to N words of length ell, such that the NFA accepts W, which is an error-detecting language. The alphabet to use is {0,1,...,s-1}. where s <= 10.

Parameters

- **N** (*int*) – the number of words to construct
- **ell** (*int*) – the codeword length
- **s** (*int*) – the alphabet size (must be <= 10)
- **n** (*int*) – number of tries when needing a new word

Returns
an automaton and a list of strings

Return type
tuple

makeCode0(*N*, *ell*, *s*, *n=2000*, *end=None*, *ov_free=False*)

Returns an NFA and a list W of up to N words of length ell, such that the NFA accepts W, which is an error-detecting language. The alphabet to use is {0,1,...,s-1}. where s <= 10.

Parameters

- **N** (*int*) – the number of words to construct
- **ell** (*int*) – the codeword length
- **s** (*int*) – the alphabet size (must be <= 10)
- **n** (*int*) – number of tries when needing a new word
- **end** (*Word*) – a Word or None that should much the end of code words
- **ov_free** (*Boolean*) – if True code words much be overlap free

Returns
an automaton and a list of strings

Return type
tuple

Note: not ov_free and end defined simultaneously Note: end should be a Word

maximalP(*aut*, *U=None*)

Tests if the language is maximal w.r.t. the property

Parameters

- **aut** (*NFA*) – the automaton
- **U** (*NFA*) – Universe of permitted words (sigma^* as default)

Return type
bool

notMaxStatW(*aut*, *ell*, *n=2000*, *ov_free=False*)

Returns a word of length ell to add into aut or None; simpler version of function nonMaxStatFEpsW

Parameters

- **aut** (NFA) – the automaton
- **ell** (*int*) – the length of the words in aut
- **n** (*int*) – number of words to try

Returns
a string or None

Return type
str

notMaximalW(*aut*, *U=None*)

Tests if the language is maximal w.r.t. the property

Parameters

- **aut** (DFA / NFA) – the automaton
- **U** (DFA / NFA) – Universe of permitted words (sigma^* as default)

Return type
bool

Raises
PropertyNotSatisfied – if not satisfied

notSatisfiesW(*aut*)

Return a witness of non-satisfaction of the property by the automaton language

Parameters
aut (DFA / NFA) – the automaton

Returns
word witness pair

Return type
tuple

satisfiesP(*aut*)

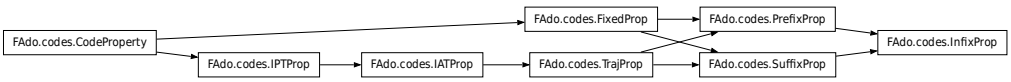
Satisfaction of the property by the automaton language

Parameters
aut (DFA / NFA) – the automaton

Return type
bool

class InfixProp(*t*)

Infix Property



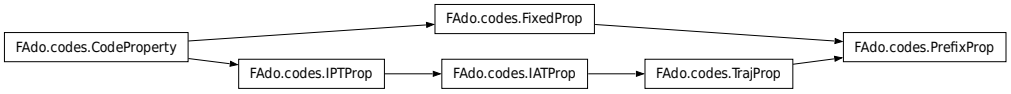
Constructor

Parameters

- **aut** (DFA/NFA) – regular expression over {0,1}
- **Sigma** (set) – the alphabet

class OutfixProp(*t*)

Outfix Property



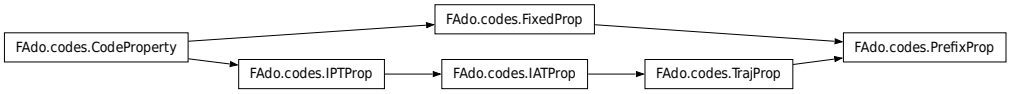
Constructor

Parameters

- **aut** (DFA/NFA) – regular expression over {0,1}
- **Sigma** (set) – the alphabet

class PrefixProp(*t*)

Prefix Property



Constructor

Parameters

- **aut** (DFA/NFA) – regular expression over {0,1}
- **Sigma** (set) – the alphabet

satisfiesPrefixP(*aut*)

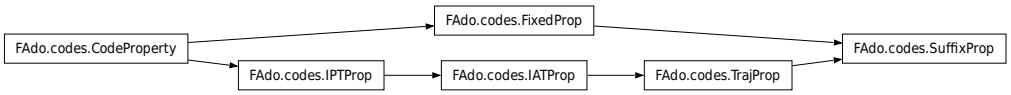
Satisfaction of property by the automaton language: faster than satisfiesP

Parameters
aut (DFA/NFA) – the automaton

Return type
bool

class SuffixProp(*t*)

Suffix Property



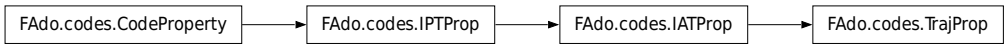
Constructor

Parameters

- **aut** (DFA / NFA) – regular expression over {0,1}
- **Sigma** (set) – the alphabet

class TrajProp(*aut*, *Sigma*)

Class of trajectoty properties



Constructor

Parameters

- **aut** (DFA / NFA) – regular expression over {0,1}
- **Sigma** (set) – the alphabet

static trajToTransducer(*traj*, *Sigma*)

Input Altering Tranducer corresponding to a Trajectory

Parameters

- **traj** (NFA) – trajectory language
- **Sigma** (set) – alphabet

Return type

SFT

class UDCodeProp(*alphabet*)

Uniquely decodable Code Property

maximalP(*aut*, *U=None*)

Tests if the language is maximal w.r.t. the property :param DFA|NFA aut: the automaton :param DFA|NFA U: Universe of permitted words (sigma^* as default) :rtype: bool

notSatisfiesW(*aut*)

Test whether the language is a code.

Parameters
aut (DFA / NFA) – the automaton

Returns
two different factorizations of the same word

Return type
tuple of list

satisfiesP(*aut*)

Satisfaction of the code property by the automaton language

Parameters
aut (DFA / NFA) – the automaton

Return type
bool

buildErrorCorrectPropF(*fname*)

Builds an Error Correcting Property

Parameters
fname (*str*) – file name

Return type
ErrCorrectProp

buildErrorCorrectPropS(*s*)

Builds an Error Correcting Property from string

Parameters
s (*str*) – transducer string

Return type
ErrCorrectProp

buildErrorDetectPropF(*fname*)

Builds an Error Detecting Property

Parameters
fname (*str*) – file name

Return type
ErrDetectProp

buildErrorDetectPropS(*s*)

Builds an Error Detecting Property from string

Parameters
s (*str*) – transducer string

Return type
ErrDetectProp

buildHypercodeProperty(*alphabet*)

Builds a Hypercode Property

Parameters
alphabet (*set*) – alphabet

Return type
PrefixProp

buildIATPropF(*fname*)

Builds a IATProp from a FAdo SFT file

Parameters
fname (*str*) – file name

Return type
IATProp

buildIATPropS(*s*)

Builds a IATProp from a FAdo SFT string

Parameters
s (*str*) – string containing SFT

Return type
IATProp

buildIPTPropF(*fname*)

Builds a IPTProp from a FAdo SFT file

Parameters
fname (*str*) – file name

Return type
IPTProp

buildIPTPropS(*s*)

Builds a IPTProp from a FAdo SFT string

Parameters
s (*str*) – file name

Return type
IPTProp

buildInfixProperty(*alphabet*)

Builds a Suffix Code Property

Parameters
alphabet (*set*) – alphabet

Return type
PrefixProp

buildOutfixProperty(*alphabet*)

Builds a Outfix Code Property

Parameters
alphabet (*set*) – alphabet

Return type
PrefixProp

buildPrefixProperty(*alphabet*)

Builds a Prefix Code Property

Parameters
alphabet (*set*) – alphabet

Return type
PrefixProp

buildSuffixProperty(*alphabet*)

Builds a Suffix Code Property

Parameters
alphabet (*set*) – alphabet

Return type
PrefixProp

buildTrajPropS(*regex*, *sigma*)

Builds a TrajProp from a string RegExp

Parameters

- **regex** (*str*) – the regular expression
- **sigma** (*set*) – alphabet

Return type
TrajProp

buildUDCodeProperty(*alphabet*)

Builds a UDCodeProp (from thin air ;-)

Parameters
alphabet (*set*) – alphabet

Return type
UDCodeProp

constructCode(*n*, *l*, *p*, *ipt=False*, *seed=None*)

Returns up to n words of length l satisfying the property p, the first one being seed.
If ipt is True, the property is assumed to be input-preserving transducer type

Return type
list

createInputAlteringSIDTrans(*n*, *sigmaSet*)

Create an input-altering SID transducer based

Parameters

- **n** (*int*) – max number of errors
- **sigmaSet** (*set*) – alphabet

Returns
a transducer representing the SID channel

Return type
SFT

editDistanceW(*auto*)

Compute the edit distance of a given regular language accepted by the NFA via Input-altering transducer.

Parameters
auto (NFA) – language recogniser

Returns
The edit distance of the given regular language plus a witness pair

Return type
tuple

Attention: language should have at least two words

See also:

Lila Kari, Stavros Konstantinidis, Steffen Kopecki, Meng Yang. An efficient algorithm for computing the edit distance of a regular language via input-altering transducers. arXiv:1406.1041 [cs.FL]

exponentialDensityP(*aut*)

Checks if language density is exponential
Using breadth first search (BFS)

Parameters
aut (NFA) – the representation of the language

Return type
bool

Attention: aut should not have Epsilon transitions

fixedHierSubset(*x*, *y*)

Returns whether x==y, or the fixed property with name x is a subset of y Currently (Jan 2015) the fixed properties names are 'UD_codes', 'Prefix_codes', 'Suffix_codes', 'Infix_codes', 'Outfix_codes', 'Hypercodes'

Parameters

- **x** (*tuple*) – first argument
- **y** (*tuple*) – second argument

Return type
bool

isSubclass(*p1*, *p2*)

Which property (language class) is a subclass of the other (if any).
It returns 1 if p1 is a subclass of p2; 2 if p2 is a subclass of p1; 3 if they are equal; 0 otherwise

Parameters

- **p1** (IPTProp) – an input preserving transducer property
- **p2** (IPTProp) – an input preserving transducer property

Return type
int

list2string(*lt*, *dy*)

Parameters

- **lt** (*list*) – list of nonnegative integers from some set {0,1,...,q-1}
- **dy** (*dict*) – mapping from {0,1,...,q-1} to some alphabet symbols

Returns
string of symbols corresponding to the integers in lt

Return type
str

long2base(*n*, *q*)

Maps n to a list of digits corresponding to the base q representation of n in reverse order

Parameters

- **n** (*int*) – a positive integer
- **q** (*int*) – base to represent n

Returns
list of q-ary 'digits', that is, elements of {0,1,..,q-1}

Return type
list

notUniversalStatW(*a*, *l*, *maxIter=20000*)

Tests statistically whether the NFA a is l-non-universal, by evaluating a on <= maxIter randomly chosen words of length l

Parameters
l (*int*) – nonnegative integer

Returns
(w,i) where w is the word found at i-th try; or (None, i) after i tries

Return type
tuple

pickFrom(*s*, *ell*)

Returns a random string of length ell over {0,1,...,s-1}

Parameters

- **s** (*int*) – the size of the alphabet (should be <= 10)
- **ell** (*int*) – the length of the desired string

Returns
a string

Return type
str

symmAndRefl(*t*, *ipt=False*)

Return the transducer t | t.inverse, if ipt is True;
return the transducer t | t.inverse | id, otherwise

Return type
SFT

unionOfIDs(*first*, *second*)

Returns the "union" of the two sets: property ID X and propoerty ID Y. The result is the set union minus any element in one set that names a property containing a property named in the other set

Parameters

- **first** (*set*) – first argument
- **second** (*set*) – second argument

Return type
set

1.13 FAdo.prax

Polynomial Random Approximation Algorithms

class Dirichlet(*t=2.000001*, *d=1*)

Dirichlet distribution function

Parameters

- **d** (*int | float*) – displacement
- **t** (*int | float*) –

Return type
float

New in version 2.0.4.

max_length(*e*)

Computes the maximal length that needs to be considered for a given error

Parameters
e (*float*) – error

Return type
int

sum_list2(*L*, *F*)

Returns the Dirichlet D_{t,d} probability of the set of words
which is the union of (1/2**F[i]) of the words of length L[i], for i=0,...,len(L)-1

Parameters

- **t** (*float*) – parameter in (1,+∞) of the Dirichlet distribution
- **d** (*int*) – minimum length of a word w for which D_{t,d}(w)>0
- **L** (*list*) – list of word lengths (integers)
- **F** (*list*) – list of lengths (integers)

Return type
DFA

sum_minus2(*L*, *F*)

Returns the probability of the complement of the set referred to in the function sum_dirichlet_list2(t,d,L,F)

class GenWordDis(*f*, *alf*, *e*, *strict=False*)

Word generator according to a given distribution function (used for sizes), for prax test

Variables

- **sigma** (*list*) – alphabet
- **pd** (PDistribution) – distribution
- **e** (*float*) – acceptable error
- **n_tries** (*int*) – size of the sample
- **max_length** (*int*) – maximal size of the words sampled
- **dist** (*list*) – cumulative probability for each size considered (up to max_length)

class Lambert(*d=1*, *z=0.9*)

Laplace distribution function

Parameters

- **d** (*int*) – displacement
- **z** (*float*) – a number 9<z<1

Return type
float

Raises
FAdoGeneralError – if z is null

class PDistribution

Probability Distribution

block_maximality_index(*eps*, *a*, *prop*)

Maximality index of a block automaton for a given transducer property prop
Using approx tolerance eps

Parameters

- **eps** (*float*) – approximation tolerance
- **a** (FA) – block automaton (accepting words of fixed length)
- **prop** (IPTProp) – transducer property

Returns
block maximality index

Return type
float

block_unive_index(*eps*, *a*)

Universality index of a block automaton (accepting words of fixed length)
Using approx tolerance eps

Parameters

- **eps** (*float*) – approximation tolerance
- **a** (FA) – block automaton (accepting words of fixed length)

Returns
block universality index

Return type
float

maximal_index_p(*g*, *aut*, *prop*)

Maximality index of a automaton for a given distribution and code property (parallel version)

Parameters

- **g** (GenWordDis) – distribution
- **aut** (FA) – automaton
- **prop** (CodeProperty) –

Returns
maximality index

Return type
float

maximality_index(*g*, *aut*, *prop*)

Maximality index (approximate) of a automaton for a given
distribution and transducer property prop

Parameters

- **g** (GenWordDis) – distribution
- **aut** (FA) – automaton
- **prop** (IPTProp) – transducer property

Returns
universality index

Return type
float

minI(*a*, *t*, *u=None*)

An operator that returns a t-independent language containing L(a)

Parameters

- **a** (FA) – the initial automaton
- **t** (Transducer) – input-altering transducer
- **u** (FA | *None*) – universe to consider

Return type
NFA

prax_block_maximal_nfa(*eps*, *a*, *prop*, *debug=False*)

Polynomial Randomized Approximation (PRAX) for block NFA maximality
wrt a code property

Parameters

- **eps** (*float*) – approximation tolerance

- **a** (FA) – block automaton (accepting words of fixed length)
- **prop** (IPTProp) – transducer property
- **debug** (*bool*) –

Return type
bool

prax_block_univ_nfa(*eps*, *a*, *debug=False*)

Polynomial Randomized Approximation (PRAX) for block NFA universality

Parameters

- **eps** (*float*) – approximation tolerance
- **a** (FA) – block automaton (accepting words of fixed length)
- **debug** (*bool*) –

Return type
bool

See also:

S.Konstantinidis, M.Mastnak, N.Moreira, R.Reis. Approximate NFA Universality and Related Problems Motivated by Information Theory, arXiv, 2022.

prax_maximal_nfa(*g*, *a*, *prop*, *debug=False*)

Polynomial Randomized Approximation (PRAX) for NFA maximality wrt a code property

Parameters

- **g** (GenWordDis) – distribution
- **a** (FA) – automaton
- **prop** (IPTProp) – transducer property
- **debug** (*bool*) –

Return type
bool

prax_univ_nfa(*g*, *a*, *debug=False*)

Polynomial Randomized Approximation (PRAX) for NFA universality

Parameters

- **a** (FA) – the automaton being tested
- **g** (GenWordDis) – word generator
- **debug** (*bool*) –

Return type
bool

See also:

S.Konstantinidis, M.Mastnak, N.Moreira, R.Reis. Approximate NFA Universality and Related Problems Motivated by Information Theory, arXiv, 2022.

New in version 2.0.4.

random() → x in the interval [0, 1).

unive_index(*g*, *aut*)

Universality index (approximate) of an automaton for a given distribution

Parameters

- **g** (GenWordDis) – distribution
- **aut** (FA) – automaton

Returns
universality index

Return type
float

1.14 FAdo.sst

Set Specification Transducer supportt

New in version 1.4.

NFA2SSFA(*aut*)

Transforms a NFA to and SSFA

Parameters
aut (fa.NFA) – NFA

Return type
SSFA

class PSP

Relation pair of set specifications

class PSPDiff(*arg1*, *arg2*)

Relation pair of two set specifications (constrained by non equality)

inIntersection(*other*, *alph*)

Evaluates the intersect on input wit anothe Set Specification

Parameters

- **other** (SetSpec) – the other
- **alph** (set) – alphabet

Return type
PSP

class PSPEqual(*arg1*)

Relation pair of two set specifications (constrained by equality)

inIntersection(*other*, *alph*)

Evaluates the intersect on input wit anothe Set Specification

Parameters

- **other** (SetSpec) – the other
- **alph** (set) – alphabet

Return type
PSP

class PSPVanila(*arg1*, *arg2*)
Relation pair of two set specifications

alphabet()
The covering alphabet of a PSP

Return type
set

behaviour(*sigma*)
Expansion of a PSP

Return type
(set, set)

inIntersection(*other*, *alph*)
Evaluates the intersect on input with another Set Specification

Parameters

- **other** (SetSpec) – the other
- **alph** (set) – alphabet

Return type
PSP

inverse()
Inverse of a PSP

Return type
PSPVanila

isAInvariant()
Is this an alphabet invariant PSP?

Return type
bool

class SSAnyOf
Set specification for 'any'

SSConditionalNoneOf(*oset*)
Auxiliary function that coalesces an SSNoneOf into an SSAnyOf if oset is empty

class SSEmpty

class SSEpsilon

class SSFA(*alph*)
NFAs with Set Specifications as transition labels

Parameters
alph – alphabet

addEpsilonLoops()

Add epsilon loops to every state

> **Attention:** in-place modification

New in version 1.0.

addTransition(*sti1*, *spec*, *sti2*)

Add af Set Specification transition

Parameters

- **sti1** (*int*) – start state index
- **sti2** (*int*) – end state index
- **spec** (SetSpec) – symbolic spec

emptyP()

Tests if the automaton accepts an empty language

Return type
bool

New in version 1.0.

epsilonP()

Whether this NFA has epsilon-transitions

Return type
bool

toNFA()

Dummy identity function

Return type
NFA

witness()

Witness of non emptyness

Returns
word

Return type
str

class SSNoneOf(*oset*)

Set specification for 'none of…'

class SSOneOf(*oset*)

Set specification for 'one of…'

class SST(*sigma=None*)

SFT with set specification labels

addEpsilonLoops()

Add a loop transition with epsilon input and output to every state in the transducer.

addToSigma(*sym*)

Adds a new symbol to the alphabet (it it is not already there)

Parameters
sym (*unicode*) – symbol to add

Return type
int

Returns
the index of the new symbol

addTransition(*stsrc*, *pair*, *sti2*)

Parameters
sti2 – int

addTransitionProductQ(*src*, *dest*, *ddest*, *sym*, *futQ*, *pastQ*)

Add transition to the new transducer instance.

Version for the optimized product

Parameters

- **src** – source state
- **dest** – destination state
- **ddest** – destination as tuple
- **sym** – symbol
- **futQ** (*set*) – queue for later
- **pastQ** (*set*) – past queue

epsilonOutP()

Tests if epsilon occurs in transition outputs

Return type
bool

epsilonP()

Test whether this transducer has input epsilon-transitions

Return type
bool

inIntersection(*other*)

Conjunction of transducer and automata: X & Y.

Note: This is a fast version of the method that does not produce meaningfull state names.

Note: The resulting transducer is not trim.

Parameters
other (DFA/NFA) – the automata needs to be operated.

Return type
SFT

inverse()

Switch the input label with the output label.

No initial or final state changed.

Returns
Transducer with transitions switched.

Return type
SFT

nonEmptyW()

Witness of non emptyness

Returns
pair (in-word, out-word)

Return type
tuple

outIntersection(*other*)

Conjunction of transducer and automaton: X & Y using output intersect operation.

Parameters
other (DFA/NFA) – the automaton used as a filter of the output

Return type
SFT

productInput(*other*)

Returns a transducer (skeleton) resulting from the execution of the transducer with the automaton as filter on the input.

Note: This version does not use stateIndex() with the price of generating some unreachable sates

Parameters
other (SSFA) – the automaton used as filter

Return type
SST

Changed in version 1.3.3.

reversal()

Returns a transducer that recognizes the reversal of the relation.

Returns
Transducer recognizing reversal language

Return type
SFT

toInNFA()

Delete the output labels in the transducer. Translate it into an NFA

Return type
NFA

toInSSFA()

Delete the output labels in the transducer. Translate it into an SSFA

Return type
SSFA

toOutNFA()

Returns the result of considering the output symbols of the transducer as input symbols of a NFA (ignoring the input symbol, thus)

Returns
the NFA

Return type
NFA

toOutSSFA()

Returns the result of considering the output symbols of the transducer as input symbols of a SSFA (ignoring the input symbol, thus)

Returns
the SSFA

Return type
SSFA

toSFT()

Expands a SST to an SFT

Return type
SFT

toXSSFA(*side*)

Skeleton of a method that extracts both left & right language of a PSP

class SetSpec

Set Specification labels

1.15 FAdo.semigroup

Syntactic SemiGroup.

Deterministic and non-deterministic automata manipulation, conversion and evaluation.

`class SSemiGroup`

Class support for the Syntactic SemiGroup.

Variables

- **elements** – list of tuples representing the transformations
- **words** – a list of pairs (index of the prefix transformation, index of the suffix char)
- **gen** – a list of the max index of each generation
- **Sigma** – set of symbols

`WordI`(*i*) → str

Representative of an element given as index

Parameters
i (*int*) – index of the element

Returns
the first word originating the element

Return type
str

`WordPS`(*pref*, *sym*)

Representative of an element given as prefix symb

Parameters

- **pref** (*int*) – prefix index
- **sym** (*int*) – symbol index

Returns
word

Return type
str

`add`(*tr*, *pref*, *sym*, *tmplists*)

Try to add a new transformation to the monoid

Parameters

- **tr** (*tuple of int*) – transformation
- **pref** (*int or None*) – prefix of the generating word
- **sym** (*int*) – suffix symbol
- **tmplists** (*pairs of lists as (elements,words)*) – this generation lists

`addGen`(*tmplist*) → None

Add a new generation to the monoid

Parameters
tmplist (*pair of lists as (elements, words)*) – the new generation data

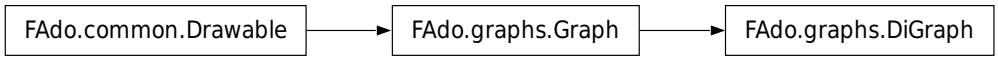
1.16 FAdo.graphs

Graph support

Basic Graph object support and manipulation

class DiGraph

Directed graph base class



addEdge(*v1*, *v2*)

Adds an edge

Parameters

- **v1** (*int*) – vertex 1 index
- **v2** (*int*) – vertex 2 index

static dotDrawEdge(*st1*, *st2*, *sep='\n'*)

Draw a transition in Dot Format

Parameters

- **st1** (*str*) – starting state
- **st2** (*str*) – ending state
- **sep** (*str*) – separator

Return type

str

dotDrawVertex(*sti*, *sep='\n'*)

Draw a Vertex in Dot Format

Parameters

- **sti** (*int*) – index of the state
- **sep** (*str*) – separator

Return type

str

dotFormat(*size='20,20'*, *direction='LR'*, *sep='\n'*, *strict=False*, *maxLblSz=10*)

A dot representation

Parameters

- **direction** (*str*) – direction of drawing
- **size** (*str*) – size of image
- **sep** (*str*) – line separator

- **maxLblSz** – max size of labels before getting removed
- **strict** – use limitations of label sizes

Returns
the dot representation

Return type
str

New in version 0.9.6.

Changed in version 0.9.8.

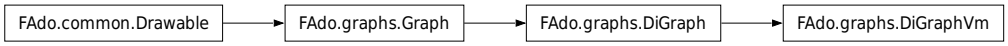
inverse()

Inverse of a digraph

class DiGraphVm

Directed graph with marked vertices

Variables
MarkedV (*set*) – set of marked vertices



markVertex(*v*)

Mark vertex v

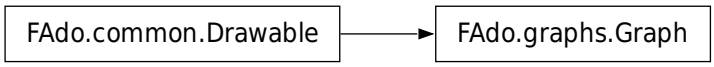
Parameters
v (*int*) – vertex

class Graph

Graph base class

Variables

- **Vertices** (*list*) – Vertices' names
- **Edges** (*set*) – set of pairs (always sorted)



addEdge(*v1*, *v2*)

Adds an edge :param int v1: vertex 1 index :param int v2: vertex 2 index :raises GraphError: if edge is loop

addVertex(*vname*)

Adds a vertex (by name)

Parameters
vname – vertex name

Returns
vertex index

Return type
int

Raises
DuplicateName – if vname already exists

abstract dotFormat(*size*)

Some dot representation

Parameters

- **size** (*str*) – size parameter for dotviz
- **filename** (*str*) – filename
- **direction** (*str*) –
- **strict** (*bool*) –
- **maxlblsz** (*int*) –
- **sep** (*str*) –

Returns: str:

vertexIndex(*vname*, *autoCreate=False*)

Return vertex index

Parameters

- **autoCreate** (*bool*) – auto creation of non existing states
- **vname** – vertex name

Return type
int

Raises
GraphError – if vname not found

1.17 FAdo.ordered

CHAPTER

TWO

VERSIONS

2.1 2.1.2

- bug introduced with the new Lark grammars when reading FAs with labelled states including curly brackets. (Fixed)

2.2 2.1.1

- ssemigroup (is now in the distribution)

2.3 2.1

- PRAX (Polynomial Random Approximation Algorithms) introduced
- Compliance with python3.10

2.4 2.0.2

- Better documentation (we hope!)
- LARK grammars now leave in their own files

2.5 2.0.1

- Port to python3

2.6 1.3.5.1

- fa: NFA.HKequivalence() added Hopcroft & Karp linear equivalence test added
- fa: type of operands of NFA intersection fixed
- reex: counting snfs

2.7 1.3.5 (Giessen)

- Myhill-Nerode relation computed for DFAs (DFA.MyhillNerodePartition())

2.8 1.3.4.1

- fixed bug in the random generator seed initialisation (thanks to Héctor L Palacios V <hectorpal@gmail.com>)

2.9 1.3.4

- rndfa & rndfap Generators now accept a seed for the random generation

2.10 1.3.3

- fa.DFA.RegExp bug fixed
- Watson-Daciuk's ADFA incremental minimisation
- DFA reversability test implemented
- DFA intersection made faster
- FA state deletion made faster
- FA product construction made faster
- makeCode made faster

2.11 1.3.2.1

- Some bugs solved (thanks to David Purser to spot them)

2.12 1.3.1

- fa.DFA.succintTransitions and fa.NFA.succintTransitions corrected

2.13 1.3

- Methods added to construct error detecting languages

2.14 1.2.1

- Random generator for ADFAs (rndadfa.py)
- Implementation of Asperti, Coen and Tassi "au-point" conversion of RegExp to DFA RegExp.dfaAuPoint()
- Implementation of Yamada, McNaughton and Glushkov conversion to DFA RegExp.dfaYMG()
- JSON format for Automata
- Ipython suport

2.15 1.2

- Better interface to FL objects
- enumDFA() and enumNFA() added to enumerate languages
- CodePprop is now UDCodeProp
- IPTProp removed
- Strict concatenation implemented (DFA.sop() for now!)
- binary operations with NFAs now deal correctly with CEpsilon-NFAs
- uniform random word generator for finite languages
- Codes hierarchy implemented
- Words are now objects (defined in commom.py)

2.16 1.1.1

- corrected bug in fa.NFA.elimEpsilon()
- Resulting NFA from __or__ gets the union of both alphabets as its alphabet

2.17 1.1

- FL.ADFA.minDFCA() corrected with the addition of ADFA.forceToDCFA()
- random generation via cfg bug fixed
- ICDFArndIncomplete bug fixed
- xre fixed with context for negation
- fa.DFA.hasTrapStateP() added
- ICDFA random generator flag bug fixed
- ICDFA random generators now written in python
- New display methods to be usable inside IPython notebooks
- Problems in Linux instalation
- SFT.evalWordP() was returning the negation of what it should.

2.18 1.0 (Halifax)

- addState() does not create states with clashing names (int/str) anymore.
- New property builders having transducers as strings instead of stored in files.
- yappy_parser permanent tables recover from reading problems, and quit shelf usage as last resort solution. Now it should work in all OS and in every possible conditions, even in a Apache execution environment
- Intersection of properties corrected for input altering transducers
- Normal Form Transducers
- Conjunction of input properties described by input altering and input preserving transducers fixed
- Infix Property added
- Error corrected with the import of new yappy_parser
- Now display() works in MSwindows, with ‘’start” instead of ‘’open”

2.19 0.9.8

- Cover automata
- New fio module that deals with i/o
- Two-way automata starting to be supported
- Distinguishability of a language
- New xre (extended Regular Expressions) module

2.20 0.9.7

- stupid error in DFA.__repr__() fixed
- better dealing of incomplete automata
- new DFA and NFA file format
- better integration with GRAIL+

2.21 0.9.6

- some random bugs corrected in combo and single operations

2.22 0.9.5

- Star and concatenation for DFAs aiming minimal transition complexity
- new API documentation
- better regular expression random generation

2.23 0.9.4

- A primitive (but working) uninstall.
- New setup for generator (bug fixed)
- Shufle was migrated to fa.py
- Shuffle for NFAs
- comboperations: Shuffle corrected
- fa: dump added to NFA and DFA

2.24 0.9.3

- Prefix-free and prefix-closed finite languages random trie added
- Renaming of AcyclicP to acyclicP. Loops are now excluded from the test unless a strict flag is passed as an argument.
- trimP corrected accordingly
- Version in package now reflects the proper version and not the major one
- Corrections and simplifications added to ADFA.minimal()
- Random balanced and “unbalanced” trie generation
- Solved a bug with a mutual inclusions between fa and fl.
- DFAtoADFA now resides in fl.
- sigmaInititialSegment() added to fl

- fa: product of dfas now ensures that its argument is a dfa.

2.25 0.9.2

- Grammar tables for grail, reex and FAdo now start with a "."
- fl.py (Finite Languages) added to the project: AFA, ADFA and ANFA supported
- Grail+ interface inproved. Now, only if the command hasmore than one argument a temporary file is created.
- Uniform random generation of trie automata with (at least) a word of a maximum lenght added (fl.py)
- rndfa.py added: a wrap for the ICDFA random Generator.
- Errors corrected in minimazation methods.
- readFromFile now supports comments as documented.
- saveToFile deals correctly with append flag.
- Bugs on deleteState() were corrected.

CHAPTER

THREE

SMALL TUTORIAL

FAdo: Tools for Language Models Manipulation

Authors: Rogério Reis & Nelma Moreira

The support of transducers and all its operations, as well of Set Specifications, is a joint work with *Stavros Konstantinidis* (St. Mary's University, Halifax, NS, Canada) (http://cs.smu.ca/~stavros/).

Contributions by

- Marco Almeida
- Ivone Amorim
- Rafaela Bastos
- Guilherme Duarte
- Miguel Ferreira
- Hugo Gouveia
- Rizó Isrof
- Eva Maia
- Casey Meijer
- Davide Nabais
- João Pires
- Meng Yang
- Joshua Young

Page of the project: http://fado.dcc.fc.up.pt.

Version: 2.2.0

Faculdade de Ciências da Universidade do Porto

Centro de Matemática da Universidade do Porto

CHAPTER

FOUR

WHAT IS FADO?

The **FAdo** system aims to provide an open source extensible high-performance software library for the symbolic manipulation of automata and other models of computation.

To allow high-level programming with complex data structures, easy prototyping of algorithms, and portability (to use in computer grid systems for example), are its main features. Our main motivation is the theoretical and experimental research, but we have also in mind the construction of a pedagogical tool for teaching automata theory and formal languages.

4.1 Regular Languages

It currently includes most standard operations for the manipulation of regular languages. Regular languages can be represented by regular expressions (RegExp) or finite automata, among other formalisms. Finite automata may be deterministic (DFA), non-deterministic (NFA) or generalized (GFA). In **FAdo** these representations are implemented as Python classes.

Elementary regular languages operations as union, intersection, concatenation, complementation and reverse are implemented for each class. Also several other regular operations (e.g shuffle) and combined operations are available for specific models.

- Several conversions between these representations are implemented:
  - NFA -> DFA: subset construction
  - NFA -> RE: recursive method
  - GFA -> RE: state elimination, with possible choice of state orderings (several heuristics)
  - RE -> NFA: Thompson, Glushkov/Position, epsilon-Follow, Follow, Partial Derivatives (naive and compressed RE), Prefix; and their duals.
  - SRE -> DFA: Brzozowski (SRE are RegExp ACIA, using sets)
  - RE -> DFA: AuPoint (Marked before) and YMG (Marked after)
- For DFAs several minimization algorithms are available: Moore, Hopcroft, and some incremental algorithms. Brzozowski minimization is available for NFAs.
- An algorithm for hyper-minimization of DFAs
- For DFAs tests for reversability
- Language equivalence of two DFAs can be determined by reducing their correspondent minimal DFA to a canonical form, or by the Hopcroft and Karp algorithm.
- For NFAs reductions by left and right bisimilarity
- Enumeration of the first words of a language or all words of a given length (Cross Section)

- Some support for the transition semigroups of DFAs

4.2 Finite Languages

Special methods for finite languages are available:

- Construction of a ADFA (acyclic finite automata) from a set of words
- Minimization of ADFAs
- Several methods for ADFAs random generation
- Methods for deterministic cover finite automata (DCFA)
- Special methods for Block languages where all words have the same length

4.3 Transducers

Several methods for transducers in standard form (SFT) are available:

- Rational operations: union, inverse, reversal, composition, concatenation, Star
- Test if a transducer is functional
- Input intersection and Output intersection operations

4.4 Codes

A *language property* is a set of languages. Given a property specified by a transducer, several language tests are possible.

- Satisfaction i.e. if a language satisfies the property
- Maximality i.e. the language satisfies the property and is maximal
- Properties implemented by transducers include: input preserving, input altering, trajectories, and fixed properties
- Computation of the edit distance of a regular language, using input altering transducers

4.5 PRAX

Polynomial Random Approximation Algorithms allow to decide hard automata problems considering cetrain natural distributions on set of words.

In particular, using the notion of approximate universality

- Test NFA universality for finite languagens
- Test NFA universality for infinite languages using tractable word distributions (Lambert and Dirichlet)

CHAPTER

FIVE

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

f

INDEX

C

D

E

F

G

M

N

O

P

Q

R

S

T