
FAdo Documentation

Release 2.0.2

Rogério Reis & Nelma Moreira

Feb 20, 2022

CONTENTS

1	What is FAdo?	3
1.1	Regular Languages	3
1.2	Finite Languages	4
1.3	Transducers	4
1.4	Codes	4
2	Module: Finite Automata (fa)	5
2.1	Classes	5
2.1.1	FA	5
2.1.2	OFA	12
2.1.3	DFA	15
2.1.4	NFA	29
2.1.5	NFAr	37
2.1.6	SSemiGroup	39
2.1.7	EnumL	40
2.2	Functions	41
2.2.1	saveToString	41
2.2.2	stringToDFA	42
3	Module: Conversion of Finite Automata: (conversions)	43
3.1	Classes	43
3.1.1	GFA	43
3.2	Functions	46
3.2.1	FA2GFA	46
3.2.2	FAallRegExps	46
3.2.3	cutPoints	46
3.2.4	FA2regexSE	46
3.2.5	FA2regexSE_nn	47
3.2.6	SP2regex	47
3.2.7	FAeliminateSingles	47
3.2.8	FA2regexCG	48
3.2.9	FA2regexCG_nn	48
3.2.10	FA2regexSEO	48
3.2.11	FA2regexDynamicCycleHeuristic	48
3.2.12	FA2regexStaticCycleHeuristic	49
3.2.13	DFA2regexDijkstra	49
3.2.14	DFAsyncWords	49
4	Module: Common Definitions (common)	51
4.1	Classes	51

4.1.1	Word	51
4.1.2	Drawable	51
5	Module: FAdo IO Functions (fio)	53
5.1	Classes	53
5.1.1	Class BuildFadoObject	53
5.2	Functions	53
5.2.1	readFromFile	53
5.2.2	readOneFromFile	54
5.2.3	readOneFromString	54
5.2.4	saveToFile	55
5.2.5	saveToJson	55
5.2.6	saveToString	55
5.2.7	toJson	55
5.3	Constants	55
6	Module: Regular Expressions (reex)	57
6.1	Classes	57
6.1.1	RegularExpression	57
6.1.2	RegExp	57
6.1.3	SpecialConstant	62
6.1.4	CEpsilon	65
6.1.5	CEmptySet	66
6.1.6	SigmaP	66
6.1.7	SigmaS	67
6.1.8	Connective	67
6.1.9	Star	68
6.1.10	Concat	68
6.1.11	Disj	68
6.1.12	Power	68
6.1.13	Option	69
6.1.14	Conj	69
6.1.15	Shuffle	69
6.1.16	Atom	70
6.1.17	Position	70
6.1.18	SConnective	70
6.1.19	SConcat	72
6.1.20	SStar	73
6.1.21	SDisj	74
6.1.22	SConj	75
6.1.23	SNot	76
6.1.24	DAG	78
6.1.25	DNode	78
6.1.26	MAtom	78
6.1.27	BuildRegexp	78
6.1.28	BuildRPNRegexp	79
6.1.29	BuildRPNSRE	79
6.1.30	BuildSRE	79
6.2	Functions	79
6.2.1	str2regexp	79
6.2.2	str2sre	79
6.2.3	rpn2regexp	80
6.2.4	to_s	80

7	Module: Transducers (transducers)	81
7.1	Classes	81
7.1.1	Transducer	81
7.1.2	SFT	81
7.1.3	NFT	87
7.1.4	GFT	87
7.2	Functions	88
7.2.1	hypercodeTransducer	88
7.2.2	infixTransducer	88
7.2.3	isLimitExceed	88
7.2.4	outfixTransducer	89
7.2.5	prefixTransducer	89
7.2.6	suffixTransducer	89
8	Module: Finite Languages (fl)	91
8.1	Classes	91
8.1.1	FL	91
8.1.2	DCFA	93
8.1.3	AFA	93
8.1.4	ADFA	94
8.1.5	ANFA	97
8.1.6	RndWGen	98
8.2	Functions	98
8.2.1	sigmaInitialSegment	98
8.2.2	genRndTrieBalanced	98
8.2.3	genRndTrieUnbalanced	98
8.2.4	genRandomTrie	99
8.2.5	genRndTriePrefix	99
8.2.6	DFatoADFA	99
8.2.7	stringToADFA	99
9	Module: Context Free Grammars Manipulation (cfg)	101
9.1	Classes	101
9.1.1	CFGrammar	101
9.1.2	CNF	102
9.1.3	cfgGenerator	102
9.1.4	reStringRGenerator	102
9.2	Functions	103
9.2.1	gRules	103
9.2.2	smallAlphabet	103
9.3	Constants	103
10	Module: Random DFA Generator (rndfap)	105
10.1	Classes	105
10.1.1	ICDFArgen	105
10.1.2	ICDFArnd	106
10.1.3	ICDFArndIncomplete	106
11	Module: Random ADFA Generator (rndadfa)	107
11.1	Classes	107
11.1.1	ADFArnd	107
12	Module: Combo Operations (comboperations)	111
12.1	Functions	111
12.1.1	starConcat	111

12.1.2	concatWStar	111
12.1.3	starWConcat	112
12.1.4	starDisj	112
12.1.5	starInter0	112
12.1.6	starInter	113
12.1.7	disjWStar	113
12.1.8	interWStar	113
13	Module: Codes (codes)	115
13.1	Classes	115
13.1.1	CodeProperty	115
13.1.2	TrajProp	116
13.1.3	IPTProp	116
13.1.4	IATProp	118
13.1.5	PrefixProp	119
13.1.6	ErrDetectProp	119
13.1.7	ErrCorrectProp	119
13.2	Functions	120
13.2.1	buildTrajPropS	120
13.2.2	buildIATPropF	120
13.2.3	buildIPTPropF	121
13.2.4	buildIATPropS	121
13.2.5	buildIPTPropS	121
13.2.6	buildErrorDetectPropF	121
13.2.7	buildErrorCorrectPropF	121
13.2.8	buildErrorCorrectPropF	121
13.2.9	buildErrorDetectPropS	122
13.2.10	buildErrorCorrectPropS	122
13.2.11	buildPrefixProperty	122
13.2.12	editDistanceW	122
13.2.13	exponentialDensityP	123
13.2.14	createInputAlteringSIDTrans	123
14	Module: Set Specification Transducers and Automata (sst)	125
14.1	Classes	125
14.1.1	PSP	125
14.1.2	PSPVanila	125
14.1.3	PDPEqual	126
14.1.4	PSPDiff	126
14.1.5	SetSpec	126
15	Module: graphs (graph creation and manipulation)	127
15.1	Classes	127
15.1.1	Graph	127
15.1.2	DiGraph	128
15.1.3	DiGraphVm	129
16	Small Tutorial	131
17	A small tutorial for FAdo	133
18	More classes and modules	139
19	Indices and tables	141

Python Module Index	143
Index	145

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
- Miguel Ferreira
- Hugo Gouveia
- Rizó Isrof
- Eva Maia
- Casey Meijer
- Davide Nabais
- Meng Yang
- Joshua Young

Page of the project: <http://fado.dcc.fc.up.pt>.

Version: 2.0.2

Copyright: 1999-2022 Rogério Reis & Nelma Moreira {rogerio.reis,nelma.moreira}@fc.up.pt

Faculdade de Ciências da Universidade do Porto

Centro de Matemática da Universidade do Porto

Licence:

This program is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation; either version 2 of the License, or (at your Option) any later version.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with this program; if not, write to the Free Software Foundation, Inc., 675 Mass Ave, Cambridge, MA 02139, USA.

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.

1.1 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 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
- RE -> NFA: Thompson method, Glushkov method, follow, Brzozowski, and partial derivatives.
- 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
- 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.
- 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

1.2 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)

1.3 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

1.4 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

MODULE: FINITE AUTOMATA (FA)

Finite automata manipulation.

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>.

2.1 Classes

2.1.1 FA

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

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.

deleteState(*sti: int*)

Remove the given state and the transitions related with that state.

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

Raises **DFAstateUnknown** – if state index does not exist

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, 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
- **sym** (*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.

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 all 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, 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 **DFASateUnknown** – 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*)

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 **DFASateUnknown** – 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() function instead

abstract succinctTransitions()

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.

<p>Attention: Does not generate the empty word</p>

New in version 0.9.8.

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 drawing - “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.

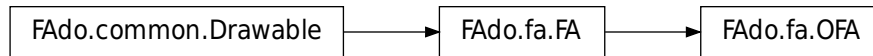
2.1.2 OFA

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

abstract deleteStates(*del_states*)

To be implemented below

Parameters **del_states** (*list*) – states to be deleted

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*

dup()

Duplicate OFA

Returns duplicate object

Return type *OFA*

eliminateStout(*st*)

Eliminate all transitions outgoing from a given state

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

Attention: performs in place alteration of the automata
--

New in version 0.9.6.

emptyP()

Tests if the automaton accepts an empty language

Return type `bool`

New in version 1.0.

abstract evalSymbol(*stil*, *sym*)

Eval symbol

abstract finalCompP(*s*)

To be implemented below

Parameters *s* – state

Return type `list`

abstract initialComp()

Initial component

Return type `list`

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`

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

To be implemented below

Parameters

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

Return type `list`

abstract succinctTransitions()

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*

abstract uniqueRepr()

Abstract method

abstract usefulStates()

To be implemented below

2.1.3 DFA

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** – 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 hyperMinimal

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** (*str*) – symbol consumed

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 they 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** – the other 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

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

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 int representing states

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

Returns set of minimal distinguishing words

Return type *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

:rtype *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 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.

hyperMinimal(*strict=False*)

Hyperminimization 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 of States

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*

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 <code>`trim()</code> .

minimal(*method*='minimalHopcroft', *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)

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 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 necessarily 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 approach 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 $\text{pref}(L(\text{self}))$

Return type *DFA*

New in version 1.1.

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 meaningful.

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 meaningful 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 Descriptive 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 counterpart

New in version 1.3.3.

Return type *list*

subword()

Returns 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. :rtype: list of tuples

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 alphabetic symbol is a tuple (symbol, index)

Returns a NFA

Return type *NFA*

usefulStates(initial_states=None)

Set of states reachable 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

static vDescription()

Generation of Verso interface description

New in version 0.9.5.

Returns the interface list

witness()

Witness of non emptiness

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

2.1.4 NFA

class NFA

Class for Non-deterministic Finite Automata (CEpsilon-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

See also:

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

Parameters

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

Returns Boolean**addEpsilonLoops()**

Add epsilon loops to every state :return: self

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
- **str** (*str*) – symbol consumed

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

Add transition to the new transducer instance.

Parameters

- **qpast** (*set*) – past queue
- **qfuture** (*set*) – future queue
- **ymb** – symbol
- **dest** (*int*) – destination state
- **srci** (*int*) – source state

New in version 1.0.

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 of frozensets**See also:**

Ilie&Yu, 2003

autobisimulation2()

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 of tuples

closeEpsilon(*st*)

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

Parameters *st* (*int*) – state index

computeFollowNames()

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()

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 CEpsilon-transitions from this automaton.

:rtype : NFA

Attention: performs in place modification of automaton

Changed in version 1.1.1.

eliminateEpsilonTransitions()

Eliminates all epsilon-transitions with no state addition

Attention: in-place modification

eliminateTSymbol (*symbol*)

Delete all transitions through a given symbol

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

Attention: in place alteration of the automata

New in version 0.9.6.

enumNFA (*n*=None)

returns the set 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

epsilonClosure(*st*)

Returns the set of states CEpsilon-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 CEpsilon connected to **st**

Return type set of int

Attention: **st** must exist.

epsilonP()

Whether this NFA has CEpsilon-transitions

Return type *bool*

epsilonPaths(*start*, *end*)

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 CEpsilon paths from start to end

Return type set of states

equivReduced(*equiv_classes*)

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 reachable from given states through given symbol and CEpsilon 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

Returns :: *bool*

followFromPosition()

computes follow automaton from a Position automaton :rtype: NFA

half()

Half operation

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()

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 (skeleton) resulting of the simultaneous execution of two DFA.

Parameters **other** (*NFA*) – the other automata

Return type *NFA*

Note: No final states are set.

Attention:

- 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

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 NFA: the resulting 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, 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 children states

Return type Set of int

stronglyConnectedComponents()

Strong components

Return type *list*

New in version 1.0.

subword()

returns a 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 CEpsilon-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()

Dummy representation. Used DFA.uniqueRepr() :rtype: tuple

usefulStates(*initial_states=None*)

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

Parameters *initial_states* (set of int or list of int) – set of initial states

Returns set of state indexes

Return type set of int

static vDescription()

Generation of Verso interface description

Return type list

New in version 0.9.5.

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

Returns the set of ending states

Return type Set of int

2.1.5 NFAr

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*) – 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 or list of int*) – collection of int representing states

elimEpsilon0()

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

Returns itself

Return type *bool*

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

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

Attention: The states of the list will become unreachable, 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 CEpsilon, then remove such transition and return the source state.

Parameters **state** (*int*) – state to check

Returns source state

Return type *int* or 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 CEpsilon, then remove such transition and return the target state.

Parameters **state** (*int*) – state to check

Returns target state

Return type *int* or None

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

2.1.6 SSemiGroup

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*)

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(*tmpLists*)

Add a new generation to the monoid

Parameters **tmpLists** (*pair of lists as (elements, words)*) – the new generation data

2.1.7 EnumL

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

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 *n*th 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

2.2 Functions

2.2.1 saveToString

saveToString(*aut*, *sep*='&')

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*

2.2.2 stringToDFA

stringToDFA(*s*, *f*, *n*, *k*)

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

MODULE: CONVERSION OF FINITE AUTOMATA: (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>.

3.1 Classes

3.1.1 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 automaton's 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:

deleteStates(*del_states*)

To be implemented below

Parameters *del_states* (*list*) – states to be deleted

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

Parameters

- *s* – state
- *visited* – list of 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

finalCompP(*s*)

To be implemented below

Parameters *s* – state

Return type *list*

initialComp()

Initial component

Return type `list`

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

uniqueRepr()

Abstract method

usefulStates()

To be implemented below

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

:members:

3.2 Functions

3.2.1 FA2GFA

FA2GFA(*aut*)

Creates a GFA equivalent to NFA

Parameters *aut* (OFA) – the automaton

Returns a GFA deep copy

Return type *GFA*

3.2.2 FAallRegExps

FAallRegExps(*aut*)

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

Parameters *aut* (OFA) – the automaton

Return type list of tuples (*int*, list of states)

3.2.3 cutPoints

cutPoints(*aut*)

Set of FA's cut points

Parameters *aut* (OFA) – the automaton

Returns set of states

Return type set of *int*

3.2.4 FA2regexSE

FA2regexSE(*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*

3.2.5 FA2regexpSE_nn

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`

3.2.6 SP2regexp

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

3.2.7 FAeliminateSingles

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*

3.2.8 FA2regexpCG

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`

3.2.9 FA2regexpCG_nn

FA2regexpCG_nn(*aut*: FAdo.fa.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`

3.2.10 FA2regexpSEO

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`

3.2.11 FA2regexpDynamicCycleHeuristic

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 Descriptive Complexity of Formal Systems (DCFS10), pages 169-180.2010. DOI: 10.4204/EPTCS.31.16

3.2.12 FA2regexpStaticCycleHeuristic

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 Descriptive Complexity of Formal Systems (DCFS10), pages 169-180.2010. DOI: 10.4204/EPTCS.31.16

3.2.13 DFA2regexpDijkstra

DFA2regexpDijkstra(*aut*) → `FAdo.reex.RegExp`

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

Parameters *aut* (**DFA**) – the automaton

Returns a regexp equivalent to the current DFA

Return type `reex.RegExp`

3.2.14 DFAsyncWords

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`

MODULE: COMMON DEFINITIONS (COMMON)

Common definitions for FAdo files

4.1 Classes

4.1.1 Word

class **Word**(*data=None, it=None*)

Class to implement generic words as iterables with pretty-print

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

4.1.2 Drawable

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.

MODULE: FADO IO FUNCTIONS (FI0)

In/Out.

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

5.1 Classes

5.1.1 Class BuildFadoObject

class BuildFadoObject(*visit_tokens=True*)
Semantics of the FAdo grammars' objects

5.2 Functions

5.2.1 readFromFile

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 a CEpsilon-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, decreares 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 | /CEpsilon
Alphabet  ::= "$" LsSymbols | /CEpsilon
Output    ::= "$" LsSymbols | /CEpsilon
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.

5.2.2 readOneFromFile

readOneFromFile(*fileName*)

Read the first of the FAdo objects from File

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

Return type *DFA|FA|STF|SST*

5.2.3 readOneFromString

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*

5.2.4 saveToFile

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

5.2.5 saveToJson

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

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

5.2.6 saveToString

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

5.2.7 toJson

toJson(*aut*)

Json for a FA

Parameters **aut** (*FA*) – the automaton

Return type *str*

5.3 Constants

const *FAdo.fio.FAdoGrammar*

MODULE: REGULAR EXPRESSIONS (REEX)

Regular expressions manipulation

Regular expression classes and manipulation

6.1 Classes

6.1.1 RegularExpression

class `RegularExpression`

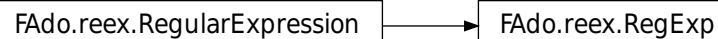
Abstract base class for all regular expression objects

6.1.2 RegExp

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.

Return type integer

Attention: Doesn't include the empty word.

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

Compare with another regular expression for equivalence. :param *r*: :param *cmp_method*: :param *nfa_method*:

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

Compare with another regular expression for equivalence through minimal DFAs. :param r: :param nfa_method:

dfaAuPoint()

DFA “au-point” according to Nipkow

Returns “au-point” DFA

Return type *fa.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

Returns word derivatives automaton

Return type *DFA*

See also:

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

dfaYMG()

DFA Yamada-McNaughton-Gluskov according 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 Boolean

abstract static epsilonLength()

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

Return type integer

static epsilonP()

Whether the regular expression is the empty word.

Return type Boolean

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* –

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

Return type *bool*

static ewp()

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

Return type Boolean

abstract first()

Return type *set*

abstract last()

Return type *set*

abstract linearForm()

Return type *dic*

abstract mark()

Make all atoms maked (tag False) :rtype: 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.

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* –

Returns NFA possibly with CEpsilon transitions

Return type NFAe

Note: The regular expression must be reduced

See also:

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

nfaGlushkov()

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

Returns NFA

nfaNaiveFollow()

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

Return type *NFA*

Note: Included for testing purposes.

See also:

Ilie & Yu (Follow Automata, 2003)

nfaPD(*pdmethod='nfaPDDAG'*)

Computes the partial derivative automaton : param pdmethod str: an implementation of the PD automaton.

Default value : nfaPDDAG :return: a PD nfa :rtype: NFA

nfaPDDAG()

”:return: a PD nfa build using a DAG :rtype: 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 NFA: partial derivatives [or equation] automaton

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.

Note: optimized version

Returns partial derivatives [or equation] automaton

Return type *NFA*

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* (*boolean*) – if not None followlists are computed as disjunct

Returns Position NFA

Return type *NFA*

nfaPre()

Prefix NFA of a regular expression States are of the form (RegExp,sym) :return: prefix automaton :rtype: NFA

See also:

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

nfaPreSlow()

Prefix NFA of a regular expression :return: prefix automaton :rtype: NFA .. seealso:: Maia et al, Prefix and Right-partial derivative automata, 11th CIE 2015, 258-267 LNCS 9136, 2015 ..note:: not working with current tailForm

notEmptyW()

Witness of non emptiness

Returns word or None

abstract rpN()

RPN representation :rtype: str :return: printable RPN representation

abstract static setOfSymbols()

Return type *set*

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

Set the alphabet for a regular expression and all its nodes

Parameters

- **symbolset** (*list or set of str*) – 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 static starHeight()

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

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

Return type `integer`

abstract `tailForm()`

Return type `dict`

toDFA()

DFA that accepts the regular expression's language

toNFA(*nfa_method*='nfaPDNaive')

NFA that accepts the regular expression's language. :param nfa_method:

abstract static `treeLength()`

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

Return type `integer`

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 of arbitrary symbols.

Return type `regular expression`

See also:

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

6.1.3 SpecialConstant

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.

Return type integer

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) :rtype: RegExp

partialDerivativesC(*sigma*)

Parameters **sigma** –

Returns

reversal()

Reversal of RegExp

Return type reex.RegExp

abstract rpn()

RPN representation :rtype: str :return: printable RPN representation

static setOfSymbols()

Returns

static starHeight()

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

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

Return type integer

support(side=True)

Returns

supportlast(side=True)

Returns

tailForm()

Returns

static treeLength()

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

Return type integer

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

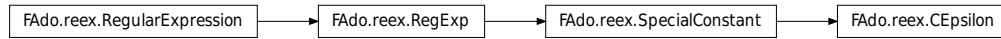
wordDerivative(word)

Parameters word –

Returns

6.1.4 CEpsilon

class CEpsilon(*sigma=None*)
 Class that represents the empty word.



Parameters *sigma* – alphabet

static epsilonLength()

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

Return type integer

static epsilonP()

Return type bool

static 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

6.1.5 CEmptySet

class CEmptySet(*sigma=None*)
Class that represents the empty set.

Parameters **sigma** – alphabet

static emptysetP()

Returns

static epsilonLength()

Returns

static epsilonP()

Returns

static ewp()

Returns

static measure(*from_parent=None*)

Parameters **from_parent** –

Returns

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

partialDerivativesC(*_*)

Returns

rpn()

Returns

6.1.6 SigmaP

SigmaP
alias of @sigmaP

6.1.7 SigmaS

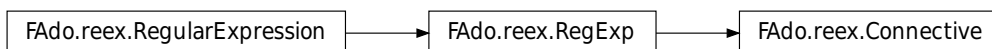
SigmaS

alias of @sigmaS

6.1.8 Connective

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.

Return type integer

Attention: Doesn't include the empty word.

epsilonLength()

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

Return type integer

first(*parent_first=None*)

Return type set

last(*parent_last=None*)

Return type set

abstract linearForm()

Return type dic

abstract mark()

Make all atoms maked (tag False) :rtype: RegExp

abstract rpn()

RPN representation :rtype: str :return: printable RPN representation

setOfSymbols()

Return type set

starHeight()

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

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

Return type integer

treeLength()

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

Return type integer

6.1.9 Star

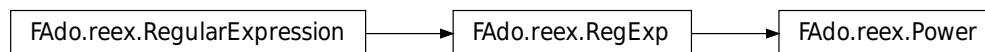
6.1.10 Concat

6.1.11 Disj

6.1.12 Power

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.

Return type integer

Attention: Doesn't include the empty word.

epsilonLength()

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

Return type integer

first()

Return type set

last()

Return type set

linearForm()

Return type dic

mark()

Make all atoms maked (tag False) :rtype: RegExp

reversal()

Reversal of RegExp

Return type reex.RegExp

rpn()

RPN representation :rtype: str :return: printable RPN representation

setOfSymbols()

Return type set

starHeight()

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

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

Return type integer

tailForm()

Return type dict

treeLength()

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

Return type integer

6.1.13 Option

Option

alias of -

6.1.14 Conj

Conj

alias of &

6.1.15 Shuffle

Shuffle

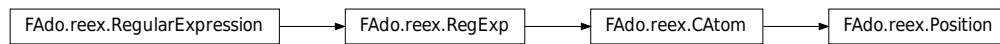
alias of :

6.1.16 Atom

6.1.17 Position

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 of symbols

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

6.1.18 SConnective

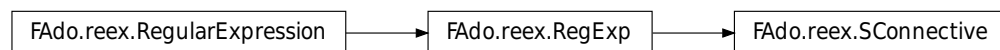
class `SConnective`(*arg*, *sigma=None*)

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

associativity of concatenation; identities σ^* and σ^+ . Connectives are: `SDisj`: disjunction

`SConj`: intersection `SConcat`: concatenation

For parsing use `str2sre`



alphabeticLength()

Returns

epsilonLength()

Returns

first()

Return type `set`

last()

Return type `set`

linearForm()

Return type `dic`

mark()

Make all atoms maked (tag False) :rtype: RegExp

nfaPD(*pdmethod='nfaPDNaive'*)

Computes the partial derivative automaton

rpn()

RPN representation :rtype: str :return: printable RPN representation

setOfSymbols()

Returns

starHeight()

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

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

Return type `integer`

syntacticLength()

Returns

abstract tailForm()

Return type `dict`

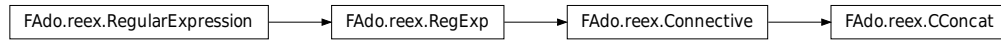
treeLength()

Returns

6.1.19 SConcat

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()`

Return type `dict`

`tail_rev()`

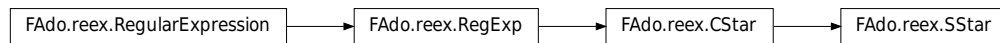
Returns

6.1.20 SStar

`class SStar(arg, sigma=None)`

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

SStar: Class that represents Kleene star



`derivative(sigma)`

Parameters `sigma` –

Returns

`linearForm()`

Returns

`nfaPD(pmethod='nfaPDNaive')`

Computes the partial derivative automaton

`partialDerivatives(sigma)`

Parameters `sigma` –

Returns

`partialDerivativesC(sigma)`

Parameters `sigma` –

Returns

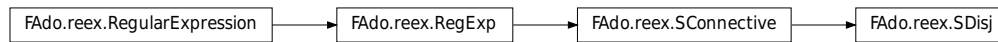
`support(side=True)`

Returns

6.1.21 SDisj

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

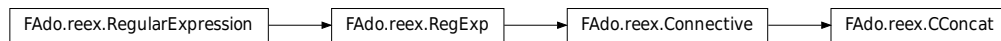
tailForm()

Return type `dict`

6.1.22 SConj

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

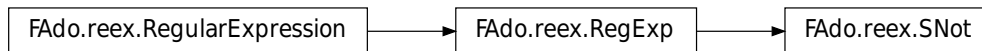
tailForm()

Return type `dict`

6.1.23 SNot

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

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



alphabeticLength()

Returns

derivative(*sigma*)

:param *sigma* :return:

epsilonLength()

Returns

ewp()

Returns

first()

Return type `set`

last()

Return type `set`

linearForm()

Returns

linearFormC()

Returns

mark()

Make all atoms maked (tag False) :rtype: RegExp

nfaPD(*pdmethod='nfaPDNaive'*)

Computes the partial derivative automaton

partialDerivatives(*sigma*)

Parameters *sigma* –

Returns

partialDerivativesC(*sigma*)

Parameters *sigma* –

Returns

rpn()

RPN representation :rtype: str :return: printable RPN representation

setOfSymbols()

Returns

starHeight()

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

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

Return type integer

support(*side=True*)

Returns

syntacticLength()

Returns

tailForm()

Return type dict

treeLength()

Returns

6.1.24 DAG

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.

Variables **reg** (*reex*) – regular expression

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

both arguments are assumed to be already present in the DAG

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*

6.1.25 DNode

class **DNode**(*op, arg1=None, arg2=None*)

6.1.26 MAtom

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*

6.1.27 BuildRegexp

class **BuildRegexp**(*context=None*)

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

6.1.28 BuildRPNRegexp

```
class BuildRPNRegexp(context=None)
```

6.1.29 BuildRPNSRE

```
class BuildRPNSRE(context=None)
```

6.1.30 BuildSRE

```
class BuildSRE(context=None)
```

Parser for sre

6.2 Functions

6.2.1 str2regexp

```
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

6.2.2 str2sre

```
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

6.2.3 rpn2regexp

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

6.2.4 to_s

to_s(*r*)

Returns a sre from FAdo regexp.

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

Return type RegExp

MODULE: TRANSDUCERS (TRANSDUCERS)

Finite Tranducer Support

Transducer manipulation.

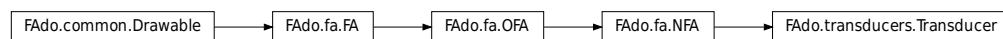
New in version 1.0.

7.1 Classes

7.1.1 Transducer

class Transducer

Base class for Transducers



setOutput (*listOfSymbols*)

Set Output

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

succintTransitions()

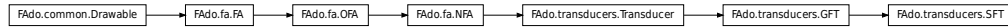
Collects the transition information in a concat way suitable for graphical representation. :rtype: list of tuples

7.1.2 SFT

class SFT

Standard Form Tranducer

Variables **Output** (*set*) – output alphabet



addEpsilonLoops()

Add a loop transition with CEpsilon 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 CEpsilon occurs in transition outputs

Return type *bool*

epsilonP()

Test whether this transducer has input CEpsilon-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 emptiness

Returns pair (in-word, out-word)

Return type *tuple*

nonFunctionalW()

Returns a witness of non functionality (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 output 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 Normal 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

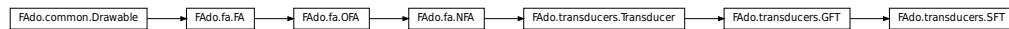
Return type *SFT*

7.1.3 NFT

class **NFT**

Normal Form Transducer.

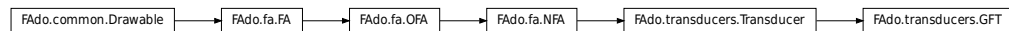
Transitions here have labels of the form (s,Epsilon) or (Epsilon,s)



7.1.4 GFT

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

:members:

7.2 Functions

7.2.1 hypercodeTransducer

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*

7.2.2 infixTransducer

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*

7.2.3 isLimitExceed

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`

7.2.4 outfixTransducer

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*

7.2.5 prefixTransducer

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*

7.2.6 suffixTransducer

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*

MODULE: FINITE LANGUAGES (FL)

Finite languages and related automata manipulation

Finite languages manipulation

8.1 Classes

8.1.1 FL

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, s.Mihov, B.Watson and r.E.Watson

Return type *ADFA*

addWord(*word*)

Adds a word to a FL :type word: Word :rtype: FL

addWords(*wList*)

Adds a list of words to a FL

Parameters **wList** (*list*) – words to add

diff(*other*)

Difference of FL: a - b

Parameters **other** (*FL*) – right hand operand

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** ([DFA](#)/[NFA](#)) – 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](#)) – right hand operand

Raises **FAdoGeneralError** – if both arguments are not FL

multiLineAutomaton()

Generates the trivial linear ANFA equivalent to this language

Return type [ANFA](#)

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

Sets the alphabet of a FL

Parameters

- **Sigma** ([set](#)) – 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

Return type [bool](#)

toDFA()

Generates a DFA recognizing the language

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

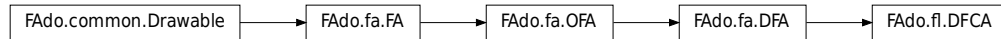
Return type *FL*

Raises **FAdoGeneralError** – if both arguments are not FL

8.1.2 DCFA

class **DFCA**

Deterministic Cover Automata class



property **length**

size of the longest word :rtype: int

Type return

8.1.3 AFA

class **AFA**

Base class for Acyclic Finite Automata

```

graph LR
    A[FAdo.fl.AFA]
  
```

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 ranf 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 accessibility map

Return type `tuple`

getLeaves()

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

Returns 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** (`int`) – index of the dead state

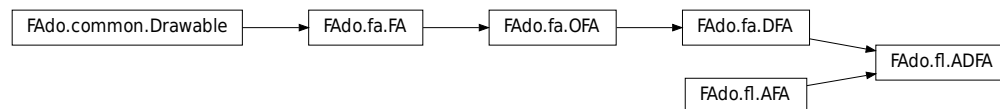
Attention: nothing is done to ensure that the state given is legitimate

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

8.1.4 ADFA

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*) – a state to be identified as dead state if one was not identified yet

Return type *ADFA*

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 :param dict witnesses: optional witness dictionary :rtype: FL

New in version 1.2.1.

dup()

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

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 *DFCA*

New in version 0.9.8.

See also:

Cezar Campeanu, Andrei Păun, 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

See also:

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

Changed in version 1.3.3.

Returns the minimal equivalent ADFA

Return type *ADFA*

minimalP(*method=None*)

Tests if the DFA is minimal

Parameters **method** – 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

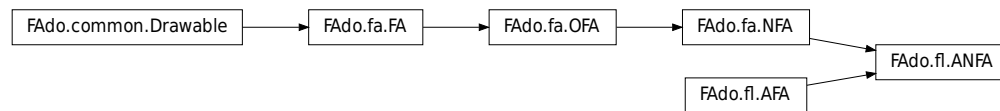
Return type *RndWGen*

New in version 1.2.

8.1.5 ANFA

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
- **s2** (*int*) – state

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

Attention: the object is modified in place

moveFinal(st, stf)

Unsets a set as final transferring 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

8.1.6 RndWGen

class **RndWGen**(*aut*)

Word random generator class

New in version 1.2.

Parameters **aut** ([ADFA](#)) – automata recognizing the language

8.2 Functions

8.2.1 sigmaInitialSegment

sigmaInitialSegment(*Sigma*, *l*, *exact=False*)

Generates the ADFA recognizing σ^i for $i \leq l$:param set Sigma: the alphabet :param int l: length :param bool exact: only the words with exactly that length? :returns: the automaton :rtype: ADFA

8.2.2 genRndTrieBalanced

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

Generates a random trie automaton for a binary language of balanced words of a given length for max word :param int maxL: length of the max word :param set Sigma: alphabet to be used :param bool safe: should a word of size maxL be present in every language? :return: the generated trie automaton :rtype: ADFA

8.2.3 genRndTrieUnbalanced

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](#)

8.2.4 genRandomTrie

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

Generates a random trie automaton for a finite language with a given length for max word :param int maxL: length of the max word :param set Sigma: alphabet to be used :param bool safe: should a word of size maxl be present in every language? :return: the generated trie automaton :rtype: ADFA

8.2.5 genRndTriePrefix

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 :param int maxL: length of the max word :param set Sigma: alphabet to be used :param bool ClosedP: should it be a prefix closed language? :param bool safe: should a word of size maxl be present in every language? :return: the generated trie automaton :rtype: ADFA

8.2.6 DFAtoADFA

DFAtoADFA(*aut*)

Transforms an acyclic DFA into a ADFA

Parameters *aut* (**DFA**) – the automaton to be transformed

Raises **notAcyclic** – if the DFA is not acyclic

Returns the converted automaton

Return type **ADFA**

8.2.7 stringToADFA

stringToADFA(*s*)

Convert a canonical string representation of a ADFA to a ADFA :param list s: the string in its canonical order :returns: the ADFA :rtype: 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.

MODULE: CONTEXT FREE GRAMMARS MANIPULATION (CFG)

Context Free Grammars Manipulation.

Basic context-free grammars manipulation for building uniform random generetors

9.1 Classes

9.1.1 CFGGrammar

class `CFGGrammar`(*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 $X \rightarrow^* []$

makenonterminals()

Extracts $C\{\text{nonterminals}\}$ from grammar rules.

maketerminals()

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

9.1.2 CNF

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

9.1.3 cfgGenerator

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

CFG uniform genetaror

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

9.1.4 reStringRGenerator

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 arbitrary re over an alphabet of 10 symbols of size 100: reStringR-Generator (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

9.2 Functions

9.2.1 gRules

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

Transforms a list of rules into a grammar description.

Parameters

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

Returns a grammar description

9.2.2 smallAlphabet

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`

9.3 Constants

const reGrammar

MODULE: RANDOM DFA GENERATOR (RNDFAP)

Random DFA generation (alternative version in python)

ICDFA Random generation binding

New in version 1.0.

10.1 Classes

10.1.1 ICDFArgen

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 probable 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*

10.1.2 ICDFArnd

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

Complete IC DFA 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

10.1.3 ICDFArndIncomplete

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

Incomplete IC DFA 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

MODULE: RANDOM ADFA GENERATOR (RNDADFA)

Random ADFA generation

ADFA Random generation binding

New in version 1.2.1.

11.1 Classes

11.1.1 ADFArnd

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 IC DFA *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 sources

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)

Uniformly 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

MODULE: COMBO OPERATIONS (COMBOOPERATIONS)

Several combined operations for DFAs

Combined operations

12.1 Functions

12.1.1 starConcat

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.

12.1.2 concatWStar

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.

12.1.3 starWConcat

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

12.1.4 starDisj

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.

12.1.5 starInter0

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.

12.1.6 starInter

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*

12.1.7 disjWStar

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.

12.1.8 interWStar

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.

MODULE: CODES (CODES)

Code theory module

New in version 1.0.

13.1 Classes

13.1.1 CodeProperty

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

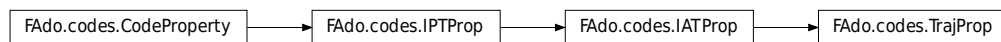
Parameters **aut** (*NFA* / *DFA*) – the automaton

Return type `bool`

13.1.2 TrajProp

class TrajProp(*aut*, *Sigma*)

Class of trajectory properties



Constructor

Parameters

- **aut** (*DFA* / *NFA*) – regular expression over {0,1}
- **Sigma** (*set*) – the alphabet

static trajToTransducer(*traj*, *Sigma*)

Input Altering Transducer corresponding to a Trajectory

Parameters

- **traj** (*NFA*) – trajectory language
- **Sigma** (*set*) – alphabet

Return type *SFT*

13.1.3 IPTProp

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 $\{0, 1, \dots, s-1\}$. where $s \leq 10$.

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 $\{0, 1, \dots, s-1\}$. where $s \leq 10$.

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 (Σ^* 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 (Σ^* 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

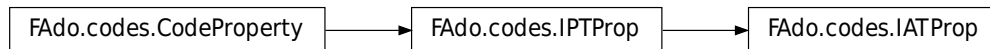
Parameters **aut** (`DFA/NFA`) – the automaton

Return type `bool`

13.1.4 IATProp

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

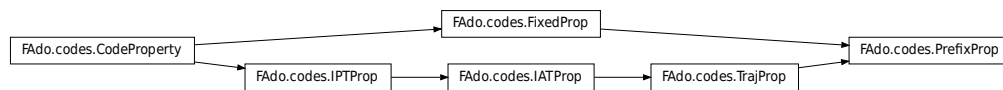
Parameters `aut` (DFA/NFA) – the automaton

Returns word witness pair

Return type tuple

13.1.5 PrefixProp

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

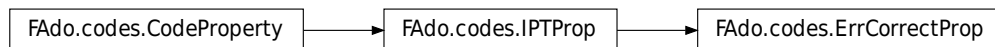
Return type bool

13.1.6 ErrDetectProp

ErrDetectProp
alias of `FAdo.codes.IPTProp`

13.1.7 ErrCorrectProp

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 (Σ^* 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`

13.2 Functions

13.2.1 buildTrajPropS

buildTrajPropS(*regex*, *sigma*)

Builds a TrajProp from a string RegExp

Parameters

- **regex** (*str*) – the regular expression
- **sigma** (*set*) – alphabet

Return type *TrajProp*

13.2.2 buildIATPropF

buildIATPropF(*fname*)

Builds a IATProp from a FAdo SFT file

Parameters **fname** (*str*) – file name**Return type** *IATProp*

13.2.3 buildIPTPropF

buildIPTPropF(*fname*)

Builds a IPTProp from a FAdo SFT file

Parameters *fname* (*str*) – file name

Return type *IPTProp*

13.2.4 buildIATPropS

buildIATPropS(*s*)

Builds a IATProp from a FAdo SFT string

Parameters *s* (*str*) – string containing SFT

Return type *IATProp*

13.2.5 buildIPTPropS

buildIPTPropS(*s*)

Builds a IPTProp from a FAdo SFT string

Parameters *s* (*str*) – file name

Return type *IPTProp*

13.2.6 buildErrorDetectPropF

13.2.7 buildErrorCorrectPropF

buildErrorDetectPropF(*fname*)

Builds an Error Detecting Property

Parameters *fname* (*str*) – file name

Return type *ErrDetectProp*

13.2.8 buildErrorCorrectPropF

buildErrorCorrectPropF(*fname*)

Builds an Error Correcting Property

Parameters *fname* (*str*) – file name

Return type *ErrCorrectProp*

13.2.9 buildErrorDetectPropS

buildErrorDetectPropS(*s*)

Builds an Error Detecting Property from string

Parameters *s* (*str*) – transducer string

Return type ErrDetectProp

13.2.10 buildErrorCorrectPropS

buildErrorCorrectPropS(*s*)

Builds an Error Correcting Property from string

Parameters *s* (*str*) – transducer string

Return type ErrCorrectProp

13.2.11 buildPrefixProperty

buildPrefixProperty(*alphabet*)

Builds a Prefix Code Property

Parameters *alphabet* (*set*) – alphabet

Return type PrefixProp

13.2.12 editDistanceW

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]

13.2.13 exponentialDensityP

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: <i>aut</i> should not have Epsilon transitions
--

13.2.14 createInputAlteringSIDTrans

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*

MODULE: SET SPECIFICATION TRANSDUCERS AND AUTOMATA (SST)

Set Specification Transducer supportt

New in version 1.4.

14.1 Classes

14.1.1 PSP

class `PSP`
Relation pair of set specifications

14.1.2 PSPVanila

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`

isInvariant()

Is this an alphabet invariant PSP?

Return type `bool`

14.1.3 PDPEqual

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`

14.1.4 PSPDiff

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`

14.1.5 SetSpec

class SetSpec

Set Specification labels

MODULE: GRAPHS (GRAPH CREATION AND MANIPULATION)

Graph support

Basic Graph object support and manipulation

15.1 Classes

15.1.1 Graph

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

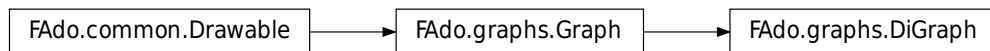
Return type *int*

Raises **GraphError** – if *vname* not found

15.1.2 DiGraph

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

15.1.3 DiGraphVm

class DiGraphVm

Directed graph with marked vertices

Variables **MarkedV** (*set*) – set of marked vertices



markVertex(*v*)

Mark vertex *v*

Parameters **v** (*int*) – vertex

SMALL TUTORIAL

A SMALL TUTORIAL FOR FADO

FAdo system is a set tools for regular languages manipulation.

Regular languages can be represented by regular expressions (RegExp) or finite automata, among other formalisms. Finite automata may be deterministic (DFA) or non-deterministic (NFA). In FAdo these representations are implemented as Python classes. A full documentation of all classes and methods is [here](#).

To work with FAdo, after installation, import the following modules on a Python interpreter:

```
>>> from FAdo.fa import *
>>> from FAdo.reex import *
>>> from FAdo.fio import *
```

The module `fa` implements the classes for finite automata and the module `reex` the classes for regular expressions. The module `fio` implements methods for IO of automata and related models.

General conventions

Methods which name ends in `P` test if the object verifies a given property and return `True` or `False`.

Finite Automata

The top class for finite automata is the class `FA`, which has two main subclasses: `OFA` for one way finite automata and the class `TFA` for two-way finite automata. The class `OFA` implements the basic structure of a finite automaton shared by DFAs and NFAs. This class defines the following attributes:

`Sigma`: the input alphabet (set)

`States`: the list of states. It is a list such that each state is referred by its index whenever it is used (transitions, Final, etc).

`Initial`: the initial state (or a set of initial states for NFA). It is an index or list of indexes.

`Final`: the set of final states. It is a list of indexes.

In general, one should not create instances (objects) of class `OFA`. The class `DFA` and `NFA` implement DFAs and NFAs, respectively. The class `GFA` implements generalized NFAs that are used in the conversion between finite automata and regular expressions. All three classes inherit from class `OFA`.

For each class there are special methods for add/delete/modify alphabet symbols, states and transitions.

DFAs

The following example shows how to build a DFA that accepts the words of $\{0,1\}^*$ that are multiples of 3.

```
>>> m3= DFA()
>>> m3.setSigma(['0','1'])
>>> m3.addState('s1')
>>> m3.addState('s2')
```

(continues on next page)

(continued from previous page)

```
>>> m3.addState('s3')
>>> m3.setInitial(0)
>>> m3.addFinal(0)
>>> m3.addTransition(0, '0', 0)
>>> m3.addTransition(0, '1', 1)
>>> m3.addTransition(1, '0', 2)
>>> m3.addTransition(1, '1', 0)
>>> m3.addTransition(2, '0', 1)
>>> m3.addTransition(2, '1', 2)
```

It is now possible, for instance, to see the structure of the automaton or to test if a word is accepted by it.

```
>>> m3
DFA([['s1', 's2', 's3'], ['1', '0'], 's1', ['s1'], "[('s1', '1', 's2'), ('s1', '0', 's1',
→ '), ('s2', '1', 's1'), ('s2', '0', 's3'), ('s3', '1', 's3'), ('s3', '0', 's2')]]")
>>> m3.evalWordP("011")
True
>>> m3.evalWordP("1011")
False
>>>
```

If graphviz is installed it is also possible to display the diagram of an automaton as follows:

```
>>>m3.display()
```

Instead of constructing the DFA directly we can load (and save) it in a simple text format. For the previous automaton the description will be:

```
@DFA 0
0 1 1
0 0 0
1 1 0
1 0 2
2 1 2
2 0 1
```

Then, if this description is saved in file mul3.fa, we have

```
>>> m3=readFromFile("mul3.fa")[0]
```

As the set of states is represented by a Python list, the list method len can be used to determine the number of states of a FA:

```
>>> len(m3.States)
3
```

For the number of Transitions the countTransitions() method must be used

```
>>> m3.countTransitions()
6
```

To minimize a DFA any of the minimization algorithms implemented can be used:

```
>>> min=m3.minimalHopcroft()
```

In this case, the DFA was already minimal so min has the same number of states as m3.

Several (regularity preserving) operations of DFAs are implemented in FAdo: boolean (union (`|` or `__or__`), intersection (`&` or `__and__`) and complementation (`~` or `__invert__`), concatenation (`Concat`), reversal (`reversal`) and Star (`Star`).

```
>>> u = m3 | ~m3
>>> u
DFA(([(1, 1), (0, 0), (2, 2)], set(['1', '0']), 0, set([0, 1, 2]), {0: {'1': 1, '0': 0}, 1: {'1': 0, '0': 2}, 2: {'1': 2, '0': 1}}))
```

```
>>> m = u.minimal()
>>> m
DFA((['(1, 1)'], ['1', '0'], '(1, 1)', ['(1, 1)'], "['(1, 1)', '1', '(1, 1)'], ('(1, 1)', '0', '(1, 1)'))
```

State names can be renamed in-place using:

```
>>> m.renameStates(range(len(m)))
```

```
DFA((['0'], ['1', '0'], '0', ['0'], "[(0, '1', 0), (0, '0', 0)]"))
```

Notice that m recognize all words over the alphabet {0,1}.

It is possible to generate a word recognisable by an automata (witness)

```
>>> u.witness()
'@CEpsilon'
```

In this case this allows to ensure that u recognizes the empty word.

This method is also useful for obtain a witness for the difference of two DFAs (`witnessDiff`).

To test if two DFAs are equivalent the the operator `==` (`equivalenceP`) can be used.

NFAs

NFAs can be built and manipulated in a similar way. There is no distinction between NFAs with and without CEpsilon-transitions. But it is possible to test if a NFA has CEpsilon-transitions and convert between a NFA with CEpsilon-transitions to a (equivalent) NFA without them.

Converting between NFAs and DFAs

The method `toDFA` allows to convert a NFA to an equivalent DFA by the subset construction method. The method `toNFA` migrates trivially a DFA to a NFA.

Regular Expressions

A regular expression can be a symbol of the alphabet, the empty set (`@emptyset`), the empty word (`@CEpsilon`) or the concatenation or the union (`+`) or the Kleene Star (`*`) of a regular expression. Examples of regular expressions are `a+b`, `(a+ba)*`, and `(@CEpsilon+ a)(ba+ab+@EmptySet)`.

The class `RegExp` is the base class for regular expressions and is used to represent an alphabet symbol. The classes `CEpsilon` and `EmptySet` are the subclasses used for the empty set and empty word, respectively. Complex regular expressions are `Concat`, `Disj`, and `Star`.

As for DFAs (and NFAs) we can build directly a regular expressions as a Python class:

```
>>> r = Star(Disj(RegExp("a"), Concat(RegExp("b"), RegExp("a"))))
>>> print r
(a + (b a))*
```

But we can convert a string to a RegExp class or subclass, using the method `str2regexp`.

```
>>> r = str2regexp("(a+ba)*")
>>> print r
(a + (b a))*
```

For regular expressions there are several measures available: alphabetic size, (parse) tree size, string length, number of epsilons and Star height. It is also possible to explicitly associate an alphabet to regular expression (even if some symbols do not appear in it) (`setSigma`)

There are several algebraic properties that can be used to obtain equivalent regular expressions of a smaller size. The method `reduced` transforms a regular expression into one equivalent without some obvious unnecessary epsilons, emptysets or stars.

Several methods that allows the manipulation of derivatives (or partial derivatives) by a symbol or by a word are implemented. However, the class `RegExp` does not deal with regular expressions module ACI properties (associativity, commutativity and idempotence of the union) (see class `xre`), so it is not possible to obtain all word derivatives of a given regular expression. This is not the case for partial derivatives.

To test if two regular expressions are equivalent the method `compare` can be used.

```
>>> r.compare(str2regexp("(a*(ba)*a)*"))
True
>>>
```

Converting Finite Automata to Regular Expressions

For pedagogical purposes, it is implemented a recursive method that constructs a regular expression equivalent to a given DFA (`regexp`).

```
>>> print m3.RegExp()
((0 + (((@CEpsilon + 0) (0* (@CEpsilon + 0)))) + ((1 + (((@CEpsilon + 0) (0* 1)))) ((1 (0* 1)
→ 1))* (1 + (1 (0
True
>>>
```

Converting Finite Automata to Regular Expressions

For pedagogical purposes, it is implemented a recursive method that constructs a regular expression equivalent to a given DFA (`regexp`).

```
>>> print m3.RegExp()
((0 + (((@CEpsilon + 0) (0* (@CEpsilon + 0)))) + ((1 + (((@CEpsilon + 0) (0* 1)))) ((1 (0* 1)
→ 1))* (1 + (1 (0
True
>>>
```

Converting Finite Automata to Regular Expressions

For pedagogical purposes, it is implemented a recursive method that constructs a regular expression equivalent to a given DFA (`RegExp`).


```
>>> print m3.regexp()
((0 + ((@CEpsilon + 0) (0* (@CEpsilon + 0)))) + ((1 + ((@CEpsilon + 0) (0* 1))) ((1 (0* 1)) * (1 + (1 (0* (@CEpsilon + 0))))))) + (((1 + ((@CEpsilon + 0) (0* 1))) ((1 (0* 1)) * 0)) ((1 + (0 ((1 (0* 1)) * 0))) * (0 ((1 (0* 1)) * (1 + (1 (0* (@CEpsilon + 0))))))))))
```

Methods based on state elimination techniques are usually more efficient, and produces much smaller regular expressions. We have implemented several heuristics for the elimination order.

```
>>> print m3.reCG()
((0 + (1 1)) + (((1 0) (1 + (0 0))*) (0 1)))
```

Converting Regular Expressions to Finite Automata

Several methods to convert between regular expressions and NFAs are implemented. With the Thompson construction a NFA with CEpsilon transitions is obtained (`nfaThompson`). Epsilon free NFAs can be obtained by the Glushkov method (Position automata) (`nfaPosition`), the partial derivatives method (`nfaPD` – several implementations) or by the follow method (`nfaFollow`). The two last methods usually allows to obtain smaller NFAs.

```
>>> r.nfaThompson()
NFA((['', '', '', '', '0', '1', '2', '3', '8', '9'], ['a', 'b'], ['8'], ['9'], "['', '@CEpsilon', ''], ('', '@CEpsilon', 0), ('', '@CEpsilon', '9'), ('', 'a', ''), ('', '@CEpsilon', ''), (0, 'b', 1), (1, '@CEpsilon', 2), (2, 'a', 3), (3, '@CEpsilon', ''), ('8', '@CEpsilon', ''), ('8', '@CEpsilon', '9'), ('9', '@CEpsilon', '8')])")
```

```
>>> r.nfaPosition()
NFA(['Initial', "('a', 1)", "('b', 2)", "('a', 3)"], ['a', 'b'], ['Initial'], ['Initial', "('a', 1)", "('a', 3)"], "['\\Initial\\', '\\a\\', '\\a\\', 1)", '\\Initial\\', '\\b\\', '\\b\\', 2)", '\\a\\', '\\a\\', 1)", '\\a\\', '\\a\\', 1)", '\\a\\', '\\b\\', '\\b\\', 2)", '\\b\\', 2)", '\\a\\', '\\a\\', 3)", '\\a\\', 3)", '\\a\\', '\\a\\', 1)", '\\a\\', 3)", '\\b\\', '\\b\\', 2)"]])
```

```
>>> r.nfaPD()
NFA(['(a + (b a))*', 'a (a + (b a))*'], ['a', 'b'], ['(a + (b a))*', '(a + (b a))*'], "[(Star(Disj(RegExp(a),Concat(RegExp(b),RegExp(a)))), 'a', Star(Disj(RegExp(a),Concat(RegExp(b),RegExp(a))))), (Star(Disj(RegExp(a),Concat(RegExp(b),RegExp(a))))), 'b', Concat(RegExp(a),Star(Disj(RegExp(a),Concat(RegExp(b),RegExp(a)))))), (Concat(RegExp(a),Star(Disj(RegExp(a),Concat(RegExp(b),RegExp(a))))), 'a', Star(Disj(RegExp(a),Concat(RegExp(b),RegExp(a))))))]")
```

General Example

Considering the several methods described before it is possible to convert between the different equivalent representations of regular languages, as well to perform several regularity preserving operations.

```
>>> r.nfaPosition().toDFA().minimal(complete=False)
DFA(['0', '2'], ['a', 'b'], '0', ['0'], "['0', 'a', '0'), ('0', 'b', '2'), ('2', 'a', '0')])")
>>> m3 == m3.reCG().nfaPD().toDFA().minimal()
True
>>>
```


MORE CLASSES AND MODULES

Several other classes and modules are also available, including:

class ICDFArnd (module rndfa.py): Random DFA generation

class FL (module fl.py): special methods for finite languages

module comboperations.py: implementation of several algorithms for several combined operations with DFAs and NFAs

module grail.py: compatibility with GRAIL

module transducers.py: several classes and methods for transducers

module codes.py: language tests for a property (set of languages) specified by a transducer

INDICES AND TABLES

- `genindex`
- `modindex`
- `search`

PYTHON MODULE INDEX

f

- FAdo.cfg, [101](#)
- FAdo.codes, [115](#)
- FAdo.comboperations, [111](#)
- FAdo.common, [51](#)
- FAdo.conversions, [43](#)
- FAdo.fa, [5](#)
- FAdo.fio, [53](#)
- FAdo.fl, [91](#)
- FAdo.graphs, [127](#)
- FAdo.rndadfa, [107](#)
- FAdo.rndfap, [105](#)
- FAdo.sst, [125](#)
- FAdo.transducers, [81](#)

A

acyclicP() (*OFA method*), 13
 add() (*SSemiGroup method*), 40
 addEdge() (*DiGraph method*), 128
 addEdge() (*Graph method*), 127
 addEpsilonLoops() (*NFA method*), 30
 addEpsilonLoops() (*SFT method*), 82
 addFinal() (*FA method*), 5
 addGen() (*SSemiGroup method*), 40
 addInitial() (*NFA method*), 30
 addOutput() (*GFT method*), 87
 addSigma() (*FA method*), 5
 addState() (*AFA method*), 93
 addState() (*FA method*), 5
 addSuffix() (*ADFA method*), 94
 addToCode() (*IPTProp method*), 117
 addTransition() (*DFA method*), 16
 addTransition() (*GFA method*), 43
 addTransition() (*GFT method*), 87
 addTransition() (*NFA method*), 30
 addTransition() (*NFAr method*), 37
 addTransition() (*OFA method*), 13
 addTransition() (*SFT method*), 82
 addTransitionProductQ() (*SFT method*), 82
 addTransitionQ() (*NFA method*), 30
 addTransitionQ() (*SFT method*), 82
 addVertex() (*Graph method*), 127
 addWord() (*FL method*), 91
 addWords() (*FL method*), 91
 ADFA (*class in FAdo.fl*), 94
 ADFArnd (*class in FAdo.rndadfa*), 107
 aEquiv() (*DFA method*), 16
 AFA (*class in FAdo.fl*), 93
 alpha() (*ADFArnd method*), 107
 alpha0() (*ADFArnd method*), 107
 alphabet() (*PSPVanila method*), 125
 alphabeticLength() (*Connective method*), 67
 alphabeticLength() (*Power method*), 68
 alphabeticLength() (*RegExp static method*), 57
 alphabeticLength() (*SConnective method*), 70
 alphabeticLength() (*SNot method*), 76

alphabeticLength() (*SpecialConstant static method*), 62
 ANFA (*class in FAdo.fl*), 97
 assignLow() (*GFA method*), 43
 assignNum() (*GFA method*), 44
 autobisimulation() (*NFA method*), 30
 autobisimulation2() (*NFA method*), 30

B

behaviour() (*PSPVanila method*), 125
 beta() (*ADFArnd method*), 108
 beta0() (*ADFArnd method*), 108
 buildErrorCorrectPropF() (*in module FAdo.codes*), 121
 buildErrorCorrectPropS() (*in module FAdo.codes*), 122
 buildErrorDetectPropF() (*in module FAdo.codes*), 121
 buildErrorDetectPropS() (*in module FAdo.codes*), 122
 BuildFadoObject (*class in FAdo.fio*), 53
 buildIATPropF() (*in module FAdo.codes*), 120
 buildIATPropS() (*in module FAdo.codes*), 121
 buildIPTPropF() (*in module FAdo.codes*), 121
 buildIPTPropS() (*in module FAdo.codes*), 121
 buildPrefixProperty() (*in module FAdo.codes*), 122
 BuildRegexp (*class in FAdo.reex*), 78
 BuildRPNRegexp (*class in FAdo.reex*), 79
 BuildRPNSRE (*class in FAdo.reex*), 79
 BuildSRE (*class in FAdo.reex*), 79
 buildTrajPropS() (*in module FAdo.codes*), 120

C

catLF() (*DAG method*), 78
 CEmptySet (*class in FAdo.reex*), 66
 CEpsilon (*class in FAdo.reex*), 65
 cfgGenerator (*class in FAdo.cfg*), 102
 CFGGrammar (*class in FAdo.cfg*), 101
 chomsky() (*CNF method*), 102
 closeEpsilon() (*NFA method*), 31
 CNF (*class in FAdo.cfg*), 102
 codeOfTransducer() (*GFT method*), 87

CodeProperty (class in FAdo.codes), 115
 compare() (RegExp method), 57
 compareMinimalDFA() (RegExp method), 57
 compat() (DFA method), 16
 complete() (ADFA method), 95
 complete() (DFA method), 17
 completeDelta() (GFA method), 44
 completeMinimal() (DFA method), 17
 completeP() (DFA method), 17
 completeProduct() (DFA method), 17
 composition() (SFT method), 82
 computeFollowNames() (NFA method), 31
 computeKernel() (DFA method), 17
 concat() (DFA method), 17
 concat() (NFA method), 31
 concat() (SFT method), 83
 concatI() (DFA method), 18
 concatWStar() (in module FAdo.comboperations), 111
 Conj (in module FAdo.reex), 69
 conjunction() (FA method), 6
 Connective (class in FAdo.reex), 67
 countAdfaBySources() (ADFArnd method), 108
 countTransitions() (FA method), 6
 countTransitions() (NFA method), 31
 createInputAlteringSIDTrans() (in module FAdo.codes), 123
 cross() (SDisj static method), 74
 cutPoints() (in module FAdo.conversions), 46

D

DAG (class in FAdo.reex), 78
 deleteState() (FA method), 6
 deleteState() (GFA method), 44
 deleteState() (SFT method), 83
 deleteStates() (DFA method), 18
 deleteStates() (GFA method), 44
 deleteStates() (NFA method), 31
 deleteStates() (NFAr method), 38
 deleteStates() (OFA method), 13
 deleteStates() (SFT method), 83
 delFinal() (FA method), 6
 delFinals() (FA method), 6
 Delta() (DFA method), 15
 delTransition() (DFA method), 18
 delTransition() (NFA method), 31
 delTransition() (NFAr method), 38
 delTransition() (SFT method), 83
 derivative() (SConcat method), 72
 derivative() (SConj method), 75
 derivative() (SDisj method), 74
 derivative() (SNot method), 76
 derivative() (SpecialConstant method), 62
 derivative() (SSStar method), 73
 deterministicP() (DFA static method), 18

deterministicP() (NFA method), 31
 detSet() (NFA method), 31
 DFA (class in FAdo.fa), 15
 DFA2regexpDijkstra() (in module FAdo.conversions), 49
 dfaAuPoint() (RegExp method), 58
 dfaBrzozowski() (RegExp method), 58
 DFASyncWords() (in module FAdo.conversions), 49
 DFAtoADFA() (in module FAdo.fl), 99
 dfaYMG() (RegExp method), 58
 DFCA (class in FAdo.fl), 93
 DFS() (GFA method), 43
 dfs_visit() (GFA method), 44
 diff() (FL method), 91
 DiGraph (class in FAdo.graphs), 128
 DiGraphVm (class in FAdo.graphs), 129
 directRank() (AFA method), 93
 disj() (FA method), 6
 disjunction() (FA method), 6
 disjWStar() (in module FAdo.comboperations), 113
 display() (Drawable method), 51
 diss() (ADFA method), 95
 dissMin() (ADFA method), 95
 dist() (DFA method), 18
 distDerivative() (SpecialConstant method), 63
 distMin() (DFA method), 19
 distR() (DFA method), 19
 distRMin() (DFA method), 19
 distTS() (DFA method), 19
 DNode (class in FAdo.reex), 78
 dotDrawEdge() (DiGraph static method), 128
 dotDrawState() (FA method), 6
 dotDrawState() (SemiDFA method), 11
 dotDrawTransition() (FA static method), 7
 dotDrawTransition() (OFA method), 13
 dotDrawTransition() (SemiDFA static method), 12
 dotDrawVertex() (DiGraph method), 128
 dotFormat() (DiGraph method), 129
 dotFormat() (Drawable method), 51
 dotFormat() (FA method), 7
 dotFormat() (Graph method), 127
 dotFormat() (NFA method), 32
 dotFormat() (SemiDFA method), 12
 dotLabel() (Drawable method), 52
 Drawable (class in FAdo.common), 51
 dump() (OFA method), 13
 dup() (ADFA method), 95
 dup() (DFA method), 19
 dup() (GFA method), 44
 dup() (NFA method), 32
 dup() (OFA method), 13
 dup() (SFT method), 83

E

[editDistanceW\(\)](#) (in module *FAdo.codes*), 122
[elim_unitary\(\)](#) (CNF method), 102
[elimEpsilon\(\)](#) (NFA method), 32
[elimEpsilonO\(\)](#) (NFAR method), 38
[eliminate\(\)](#) (GFA method), 44
[eliminateAll\(\)](#) (GFA method), 44
[eliminateDeadName\(\)](#) (FA method), 7
[eliminateEpsilonTransitions\(\)](#) (NFA method), 32
[eliminateState\(\)](#) (GFA method), 44
[eliminateStout\(\)](#) (OFA method), 13
[eliminateTSymbol\(\)](#) (NFA method), 32
[emptyP\(\)](#) (OFA method), 14
[emptyP\(\)](#) (SFT method), 83
[emptysetP\(\)](#) (CEmptySet static method), 66
[emptysetP\(\)](#) (RegExp static method), 58
[ensureDead\(\)](#) (AFA method), 93
[enum\(\)](#) (EnumL method), 40
[enumCrossSection\(\)](#) (EnumL method), 41
[enumDFA\(\)](#) (DFA method), 19
[EnumL](#) (class in *FAdo.fa*), 40
[enumNFA\(\)](#) (NFA method), 32
[epsilonClosure\(\)](#) (NFA method), 32
[epsilonLength\(\)](#) (CEpsilon static method), 66
[epsilonLength\(\)](#) (CEpsilon static method), 65
[epsilonLength\(\)](#) (Connective method), 67
[epsilonLength\(\)](#) (Power method), 68
[epsilonLength\(\)](#) (RegExp static method), 58
[epsilonLength\(\)](#) (SConnective method), 70
[epsilonLength\(\)](#) (SNot method), 76
[epsilonLength\(\)](#) (SpecialConstant static method), 63
[epsilonOutP\(\)](#) (SFT method), 83
[epsilonP\(\)](#) (CEpsilon static method), 66
[epsilonP\(\)](#) (CEpsilon static method), 65
[epsilonP\(\)](#) (NFA method), 33
[epsilonP\(\)](#) (RegExp static method), 58
[epsilonP\(\)](#) (SFT method), 83
[epsilonPaths\(\)](#) (NFA method), 33
[equal\(\)](#) (DFA method), 19
[equivalentP\(\)](#) (FA method), 7
[equivalentP\(\)](#) (RegExp method), 59
[equivP\(\)](#) (RegExp method), 58
[equivReduced\(\)](#) (NFA method), 33
[ErrCorrectProp](#) (class in *FAdo.codes*), 119
[ErrDetectProp](#) (in module *FAdo.codes*), 119
[evalNumberOfStateCycles\(\)](#) (GFA method), 44
[evalRank\(\)](#) (AFA method), 93
[evalSymbol\(\)](#) (DFA method), 19
[evalSymbol\(\)](#) (FA method), 8
[evalSymbol\(\)](#) (GFA method), 44
[evalSymbol\(\)](#) (NFA method), 33
[evalSymbol\(\)](#) (OFA method), 14
[evalSymbolI\(\)](#) (DFA method), 20
[evalSymbolL\(\)](#) (DFA method), 20

[evalSymbolLI\(\)](#) (DFA method), 20
[evalWord\(\)](#) (DFA method), 21
[evalWordP\(\)](#) (DFA method), 21
[evalWordP\(\)](#) (NFA method), 33
[evalWordP\(\)](#) (RegExp method), 59
[evalWordP\(\)](#) (SFT method), 83
[evalWordSlowP\(\)](#) (SFT method), 83
[ewp\(\)](#) (CEpsilon static method), 66
[ewp\(\)](#) (CEpsilon static method), 65
[ewp\(\)](#) (RegExp static method), 59
[ewp\(\)](#) (SConcat method), 72
[ewp\(\)](#) (SConj method), 75
[ewp\(\)](#) (SDisj method), 74
[ewp\(\)](#) (SNot method), 76
[exponentialDensityP\(\)](#) (in module *FAdo.codes*), 123

F

[FA](#) (class in *FAdo.fa*), 5
[FA2GFA\(\)](#) (in module *FAdo.conversions*), 46
[FA2regexpCG\(\)](#) (in module *FAdo.conversions*), 48
[FA2regexpCG_nn\(\)](#) (in module *FAdo.conversions*), 48
[FA2regexpDynamicCycleHeuristic\(\)](#) (in module *FAdo.conversions*), 48
[FA2regexpSE\(\)](#) (in module *FAdo.conversions*), 46
[FA2regexpSE_nn\(\)](#) (in module *FAdo.conversions*), 47
[FA2regexpSE0\(\)](#) (in module *FAdo.conversions*), 48
[FA2regexpStaticCycleHeuristic\(\)](#) (in module *FAdo.conversions*), 49
[FAallRegExps\(\)](#) (in module *FAdo.conversions*), 46
[FAdo.cfg](#)
 module, 101
[FAdo.codes](#)
 module, 115
[FAdo.comboperations](#)
 module, 111
[FAdo.common](#)
 module, 51
[FAdo.conversions](#)
 module, 43
[FAdo.fa](#)
 module, 5
[FAdo.fio](#)
 module, 53
[FAdo.fl](#)
 module, 91
[FAdo.graphs](#)
 module, 127
[FAdo.reex](#)
 module, 57
[FAdo.rndadfa](#)
 module, 107
[FAdo.rndfap](#)
 module, 105
[FAdo.sst](#)

module, 125
FAdo.transducers
 module, 81
FAeliminateSingles() (in module *FAdo.conversions*),
 47
fillStack() (*EnumL* method), 41
filter() (*FL* method), 91
finalCompP() (*DFA* method), 21
finalCompP() (*GFA* method), 44
finalCompP() (*NFA* method), 33
finalCompP() (*OFA* method), 14
finalP() (*FA* method), 8
finalsP() (*FA* method), 8
first() (*Connective* method), 67
first() (*Power* method), 68
first() (*RegExp* method), 59
first() (*SConnective* method), 71
first() (*SDisj* method), 74
first() (*SNot* method), 76
first() (*SpecialConstant* static method), 63
FL (class in *FAdo.fl*), 91
followFromPosition() (*NFA* method), 33
followLists() (*SDisj* method), 74
followLists() (*SpecialConstant* method), 63
followListsD() (*SpecialConstant* method), 63
followListsStar() (*SDisj* method), 74
followListsStar() (*SpecialConstant* static method),
 63
forceToDFA() (*ADFA* method), 95
forceToDFCA() (*ADFA* method), 95
functionalP() (*SFT* method), 84

G

gamma() (*ADFArnd* method), 108
generate() (*cfgGenerator* method), 102
genFinalities() (*ICDFArgen* method), 105
genRandomTrie() (in module *FAdo.fl*), 99
genRndTrieBalanced() (in module *FAdo.fl*), 98
genRndTriePrefix() (in module *FAdo.fl*), 99
genRndTrieUnbalanced() (in module *FAdo.fl*), 98
getLeaves() (*AFA* method), 94
GFA (class in *FAdo.conversions*), 43
GFT (class in *FAdo.transducers*), 87
Graph (class in *FAdo.graphs*), 127
gRules() (in module *FAdo.cfg*), 103

H

half() (*NFA* method), 34
hasStateIndexP() (*FA* method), 8
hasTransitionP() (*NFA* method), 34
hasTrapStateP() (*DFA* method), 21
head() (*SConcat* method), 72
head_rev() (*SConcat* method), 72
HKeqP() (*DFA* method), 16

HKeqP() (*NFA* method), 29
homogeneousFinalityP() (*NFA* method), 34
homogenousP() (*NFA* method), 34
homogenousP() (*NFAr* method), 38
hypercodeTransducer() (in module
 FAdo.transducers), 88
hyperMinimal() (*DFA* method), 21

I

IATProp (class in *FAdo.codes*), 118
ICDFArgen (class in *FAdo.rndfap*), 105
ICDFArnd (class in *FAdo.rndfap*), 106
ICDFArndIncomplete (class in *FAdo.rndfap*), 106
iCompleteP() (*EnumL* method), 41
images() (*FA* method), 8
inDegree() (*DFA* method), 21
indexList() (*FA* method), 8
infix() (*DFA* method), 21
infixTransducer() (in module *FAdo.transducers*), 88
inIntersection() (*PSPDiff* method), 126
inIntersection() (*PSPEqual* method), 126
inIntersection() (*PSPVanila* method), 125
inIntersection() (*SFT* method), 84
inIntersectionSlow() (*SFT* method), 84
initialComp() (*DFA* method), 22
initialComp() (*GFA* method), 44
initialComp() (*NFA* method), 34
initialComp() (*OFA* method), 14
initialP() (*DFA* method), 22
initialP() (*FA* method), 8
initialSet() (*DFA* method), 22
initialSet() (*FA* method), 8
initStack() (*EnumL* method), 41
inputS() (*FA* method), 9
intersection() (*FL* method), 92
interWStar() (in module *FAdo.comboperations*), 113
inverse() (*DiGraph* method), 129
inverse() (*PSPVanila* method), 125
inverse() (*SFT* method), 84
IPTProp (class in *FAdo.codes*), 116
isAInvariant() (*PSPVanila* method), 125
isLimitExceed() (in module *FAdo.transducers*), 88

J

joinStates() (*DFA* method), 22

L

last() (*Connective* method), 67
last() (*Power* method), 68
last() (*RegExp* method), 59
last() (*SConnective* method), 71
last() (*SDisj* method), 74
last() (*SNot* method), 76
last() (*SpecialConstant* method), 63

length (*DFCA property*), 93
 lEquivNFA() (*NFA method*), 34
 level() (*ADFA method*), 95
 linearForm() (*Connective method*), 67
 linearForm() (*Power method*), 68
 linearForm() (*RegExp method*), 59
 linearForm() (*SConcat method*), 72
 linearForm() (*SConj method*), 75
 linearForm() (*SConnective method*), 71
 linearForm() (*SDisj method*), 74
 linearForm() (*SNot method*), 76
 linearForm() (*SpecialConstant method*), 63
 linearForm() (*SStar method*), 73
 linearFormCC() (*SConcat method*), 72
 linearFormCC() (*SDisj method*), 74
 linearFormCC() (*SNot method*), 77
 listOfTransitions() (*GFT method*), 88
 lrEquivNFA() (*NFA method*), 34

M

MADFA() (*FL method*), 91
 makeCode() (*IPTProp method*), 117
 makeCode0() (*IPTProp method*), 117
 makenonterminals() (*CFGGrammar method*), 101
 makePNG() (*Drawable method*), 52
 makeReversible() (*DFA method*), 22
 maketerminals() (*CFGGrammar method*), 101
 mark() (*Connective method*), 67
 mark() (*Power method*), 69
 mark() (*RegExp method*), 59
 mark() (*SConnective method*), 71
 mark() (*SNot method*), 77
 mark() (*SpecialConstant method*), 63
 marked() (*RegExp method*), 59
 markNonEquivalent() (*DFA method*), 22
 markVertex() (*DiGraphVm method*), 129
 MAtom (*class in FAdo.reex*), 78
 maximalPC() (*CodeProperty method*), 115
 maximalPC() (*IPTProp method*), 117
 measure() (*CEmptySet static method*), 66
 measure() (*CEpsilon static method*), 65
 mergeInitial() (*ANFA method*), 97
 mergeLeaves() (*ANFA method*), 97
 mergeStates() (*ANFA method*), 97
 mergeStates() (*DFA method*), 22
 mergeStates() (*NFAr method*), 38
 mergeStatesSet() (*NFAr method*), 38
 minDFCA() (*ADFA method*), 95
 minimal() (*ADFA method*), 96
 minimal() (*DFA method*), 22
 minimal() (*NFA method*), 34
 minimalBrzozowski() (*OFA method*), 14
 minimalBrzozowskiPC() (*OFA method*), 14
 minimalDFA() (*NFA method*), 35

minimalHopcroft() (*DFA method*), 23
 minimalHopcroftPC() (*DFA method*), 23
 minimalIncremental() (*DFA method*), 23
 minimalIncrementalPC() (*DFA method*), 23
 minimalMoore() (*DFA method*), 23
 minimalMooreSq() (*DFA method*), 23
 minimalMooreSqPC() (*DFA method*), 24
 minimalNCompletePC() (*DFA method*), 24
 minimalNotEquivPC() (*DFA method*), 24
 minimalPC() (*ADFA method*), 96
 minimalPC() (*DFA method*), 24
 minimalWatson() (*DFA method*), 24
 minimalWatsonPC() (*DFA method*), 24
 minReversible() (*ADFA method*), 95
 minWord() (*EnumL method*), 41
 minWordT() (*EnumL method*), 41
 module

- FAdo.cfg, 101
- FAdo.codes, 115
- FAdo.comboperations, 111
- FAdo.common, 51
- FAdo.conversions, 43
- FAdo.fa, 5
- FAdo.fio, 53
- FAdo.fl, 91
- FAdo.graphs, 127
- FAdo.reex, 57
- FAdo.rndadfa, 107
- FAdo.rndfap, 105
- FAdo.sst, 125
- FAdo.transducers, 81

moveFinal() (*ANFA method*), 97
 multiLineAutomaton() (*FL method*), 92
 MyhillNerodePartition() (*DFA method*), 16

N

nextWord() (*EnumL method*), 41
 NFA (*class in FAdo.fa*), 29
 nfaFollow() (*RegExp method*), 59
 nfaFollowEpsilon() (*RegExp method*), 59
 nfaGlushkov() (*RegExp method*), 60
 nfaNaiveFollow() (*RegExp method*), 60
 nfaPD() (*CEmptySet method*), 66
 nfaPD() (*RegExp method*), 60
 nfaPD() (*SConnective method*), 71
 nfaPD() (*SNot method*), 77
 nfaPD() (*SStar method*), 73
 nfaPDDAG() (*RegExp method*), 60
 nfaPDNaive() (*RegExp method*), 60
 nfaPDO() (*RegExp method*), 60
 nfaPosition() (*RegExp method*), 61
 nfaPre() (*RegExp method*), 61
 nfaPreSlow() (*RegExp method*), 61
 nfaPSNF() (*RegExp method*), 61

NFAr (class in *FAdo.fa*), 37
nfaThompson() (*CEpsilon method*), 65
NFT (class in *FAdo.transducers*), 87
noBlankNames() (*FA method*), 9
nonEmptyW() (*SFT method*), 84
nonFunctionalW() (*SFT method*), 85
normalize() (*GFA method*), 45
notEmptyW() (*RegExp method*), 61
notequal() (*DFA method*), 24
notMaximalW() (*CodeProperty method*), 115
notMaximalW() (*ErrCorrectProp method*), 119
notMaximalW() (*IPTProp method*), 118
notMaxStatW() (*IPTProp method*), 118
notSatisfiesW() (*CodeProperty method*), 115
notSatisfiesW() (*ErrCorrectProp method*), 120
notSatisfiesW() (*IATProp method*), 118
notSatisfiesW() (*IPTProp method*), 118
NULLABLE() (*CFGGrammar method*), 101

O

OFA (class in *FAdo.fa*), 12
Option (in module *FAdo.reex*), 69
ordered() (*AFA method*), 94
orderedStrConnComponents() (*DFA method*), 24
outfixTransducer() (in module *FAdo.transducers*), 89
outIntersection() (*SFT method*), 85
outIntersectionDerived() (*SFT method*), 85
outputS() (*SFT method*), 85

P

pairGraph() (*DFA method*), 25
partialDerivatives() (*CEpsilon method*), 65
partialDerivatives() (*SConcat method*), 72
partialDerivatives() (*SConj method*), 75
partialDerivatives() (*SDisj method*), 75
partialDerivatives() (*SNot method*), 77
partialDerivatives() (*SStar method*), 73
partialDerivativesCC() (*CEmptySet method*), 66
partialDerivativesCC() (*CEpsilon method*), 65
partialDerivativesCC() (*SConcat method*), 72
partialDerivativesCC() (*SConj method*), 75
partialDerivativesCC() (*SDisj method*), 75
partialDerivativesCC() (*SNot method*), 77
partialDerivativesCC() (*SpecialConstant method*), 63
partialDerivativesCC() (*SStar method*), 73
plus() (*FA method*), 9
plusLF() (*DAG static method*), 78
Position (class in *FAdo.reex*), 70
possibleToReverse() (*ADFA method*), 96
possibleToReverse() (*DFA method*), 25
Power (class in *FAdo.reex*), 68
pref() (*DFA method*), 25
PrefixProp (class in *FAdo.codes*), 119
prefixTransducer() (in module *FAdo.transducers*), 89

print_data() (*DFA method*), 25
product() (*DFA method*), 25
product() (*NFA method*), 35
productInput() (*SFT method*), 85
productInputSlow() (*SFT method*), 85
productSlow() (*DFA method*), 25
PSP (class in *FAdo.sst*), 125
PSPDiff (class in *FAdo.sst*), 126
PSPEqual (class in *FAdo.sst*), 126
PSPVanila (class in *FAdo.sst*), 125

R

readFromFile() (in module *FAdo.fio*), 53
readOneFromFile() (in module *FAdo.fio*), 54
readOneFromString() (in module *FAdo.fio*), 54
RegExp (class in *FAdo.reex*), 57
RegularExpression (class in *FAdo.reex*), 57
renameState() (*FA method*), 9
renameStates() (*FA method*), 9
renameStatesFromPosition() (*NFA method*), 35
reorder() (*DFA method*), 25
reorder() (*GFA method*), 45
reorder() (*NFA method*), 35
rEquivNFA() (*NFA method*), 35
reStringRGenerator (class in *FAdo.cfg*), 102
reversal() (*FA method*), 10
reversal() (*NFA method*), 35
reversal() (*Power method*), 69
reversal() (*SFT method*), 85
reversal() (*SpecialConstant method*), 63
reverseTransitions() (*DFA method*), 26
reverseTransitions() (*NFA method*), 35
reversibleP() (*DFA method*), 26
rndAdfa() (*ADFArnd method*), 108
rndNumberSecondSources() (*ADFArnd method*), 109
rndTransitionsFromSources() (*ADFArnd method*), 109
RndWGen (class in *FAdo.fl*), 98
rpn() (*CEmptySet method*), 66
rpn() (*CEpsilon method*), 65
rpn() (*Connective method*), 67
rpn() (*Power method*), 69
rpn() (*RegExp method*), 61
rpn() (*SConnective method*), 71
rpn() (*SNot method*), 77
rpn() (*SpecialConstant method*), 64
rpn2regex() (in module *FAdo.reex*), 80
runOnNFA() (*SFT method*), 85
runOnWord() (*SFT method*), 86

S

same_nullability() (*FA method*), 10
satisfiesP() (*CodeProperty method*), 116
satisfiesP() (*ErrCorrectProp method*), 120

satisfiesP() (*IPTProp method*), 118
 satisfiesPrefixP() (*PrefixProp method*), 119
 saveToFile() (*in module FAdo.fio*), 55
 saveToJson() (*in module FAdo.fio*), 55
 saveToString() (*in module FAdo.fa*), 41
 saveToString() (*in module FAdo.fio*), 55
 SConcat (*class in FAdo.reex*), 72
 SConj (*class in FAdo.reex*), 75
 SConnective (*class in FAdo.reex*), 70
 SDisj (*class in FAdo.reex*), 74
 SemiDFA (*class in FAdo.fa*), 11
 setDeadState() (*AFA method*), 94
 setFinal() (*FA method*), 10
 setInitial() (*FA method*), 10
 setInitial() (*NFA method*), 35
 setInitial() (*SFT method*), 86
 setOfSymbols() (*Connective method*), 67
 setOfSymbols() (*Position method*), 70
 setOfSymbols() (*Power method*), 69
 setOfSymbols() (*RegExp static method*), 61
 setOfSymbols() (*SConnective method*), 71
 setOfSymbols() (*SNot method*), 77
 setOfSymbols() (*SpecialConstant static method*), 64
 setOutput() (*Transducer method*), 81
 setSigma() (*FA method*), 10
 setSigma() (*FL method*), 92
 setSigma() (*RegExp method*), 61
 SetSpec (*class in FAdo.sst*), 126
 SFT (*class in FAdo.transducers*), 81
 Shuffle (*in module FAdo.reex*), 69
 shuffle() (*DFA method*), 26
 shuffle() (*NFA method*), 36
 sigmaInitialSegment() (*in module FAdo.fl*), 98
 SigmaP (*in module FAdo.reex*), 66
 SigmaS (*in module FAdo.reex*), 67
 simDiff() (*DFA method*), 26
 smallAlphabet() (*in module FAdo.cfg*), 103
 sMonoid() (*DFA method*), 26
 snf() (*CEpsilon method*), 65
 SNot (*class in FAdo.reex*), 76
 sop() (*DFA method*), 26
 SP2regexp() (*in module FAdo.conversions*), 47
 SpecialConstant (*class in FAdo.reex*), 62
 square() (*SFT method*), 86
 square_fv() (*SFT method*), 86
 SSemiGroup (*class in FAdo.fa*), 39
 sSemigroup() (*DFA method*), 26
 SStar (*class in FAdo.reex*), 73
 star() (*DFA method*), 27
 star() (*NFA method*), 36
 star() (*SFT method*), 86
 starConcat() (*in module FAdo.comboperations*), 111
 starDisj() (*in module FAdo.comboperations*), 112
 starHeight() (*Connective method*), 67
 starHeight() (*Power method*), 69
 starHeight() (*RegExp static method*), 61
 starHeight() (*SConnective method*), 71
 starHeight() (*SNot method*), 77
 starHeight() (*SpecialConstant static method*), 64
 starI() (*DFA method*), 27
 starInter() (*in module FAdo.comboperations*), 113
 starInter0() (*in module FAdo.comboperations*), 112
 starWConcat() (*in module FAdo.comboperations*), 112
 stateAlphabet() (*FA method*), 10
 stateChildren() (*DFA method*), 27
 stateChildren() (*GFA method*), 45
 stateChildren() (*NFA method*), 36
 stateChildren() (*OFA method*), 14
 stateIndex() (*FA method*), 10
 stateName() (*FA method*), 11
 statePairEquiv() (*ADFA method*), 96
 str2regexp() (*in module FAdo.reex*), 79
 str2sre() (*in module FAdo.reex*), 79
 stringToADFA() (*in module FAdo.fl*), 99
 stringToDFA() (*in module FAdo.fa*), 42
 stronglyConnectedComponents() (*DFA method*), 27
 stronglyConnectedComponents() (*NFA method*), 36
 subword() (*DFA method*), 27
 subword() (*NFA method*), 36
 succinctTransitions() (*DFA method*), 27
 succinctTransitions() (*FA method*), 11
 succinctTransitions() (*GFA method*), 45
 succinctTransitions() (*NFA method*), 36
 succinctTransitions() (*OFA method*), 14
 succinctTransitions() (*Transducer method*), 81
 suff() (*DFA method*), 27
 suffixClosedP() (*FL method*), 92
 suffixTransducer() (*in module FAdo.transducers*), 89
 support() (*SConcat method*), 72
 support() (*SConj method*), 76
 support() (*SDisj method*), 75
 support() (*SNot method*), 77
 support() (*SpecialConstant method*), 64
 support() (*SStar method*), 73
 supportlast() (*SpecialConstant method*), 64
 syncPower() (*DFA method*), 27
 syntacticLength() (*SConnective method*), 71
 syntacticLength() (*SNot method*), 77

T

tail() (*SConcat method*), 72
 tail_rev() (*SConcat method*), 73
 tailForm() (*Power method*), 69
 tailForm() (*RegExp method*), 62
 tailForm() (*SConcat method*), 73
 tailForm() (*SConj method*), 76
 tailForm() (*SConnective method*), 71
 tailForm() (*SDisj method*), 75

tailForm() (*SNot method*), 77
tailForm() (*SpecialConstant method*), 64
to_s() (*in module FAdo.reex*), 80
toADFA() (*DFA method*), 28
toANFA() (*ADFA method*), 96
toDFA() (*DFA method*), 28
toDFA() (*FL method*), 92
toDFA() (*NFA method*), 36
toDFA() (*RegExp method*), 62
toInNFA() (*SFT method*), 86
toJson() (*in module FAdo.fio*), 55
toNFA() (*ADFA method*), 96
toNFA() (*DFA method*), 28
toNFA() (*FL method*), 92
toNFA() (*NFA method*), 36
toNFA() (*NFAr method*), 39
toNFA() (*RegExp method*), 62
toNFAr() (*NFA method*), 36
toNFT() (*SFT method*), 86
toOutNFA() (*SFT method*), 86
topoSort() (*OFA method*), 14
toSFT() (*GFT method*), 88
toSFT() (*SFT method*), 86
TrajProp (*class in FAdo.codes*), 116
trajToTransducer() (*TrajProp static method*), 116
Transducer (*class in FAdo.transducers*), 81
transitions() (*DFA method*), 28
transitionsA() (*DFA method*), 28
treeLength() (*Connective method*), 68
treeLength() (*Power method*), 69
treeLength() (*RegExp static method*), 62
treeLength() (*SConnective method*), 71
treeLength() (*SNot method*), 77
treeLength() (*SpecialConstant static method*), 64
trieFA() (*FL method*), 92
trim() (*ADFA method*), 96
trim() (*OFA method*), 14
trim() (*SFT method*), 86
trimP() (*OFA method*), 15

U

union() (*FA method*), 11
union() (*FL method*), 92
union() (*SFT method*), 87
unionSigma() (*RegExp method*), 62
uniqueRepr() (*DFA method*), 28
uniqueRepr() (*GFA method*), 45
uniqueRepr() (*NFA method*), 37
uniqueRepr() (*OFA method*), 15
universalP() (*DFA method*), 28
unlinkSoleIncoming() (*NFAr method*), 39
unlinkSoleOutgoing() (*NFAr method*), 39
unmark() (*DFA method*), 28
unmark() (*MAtom method*), 78

unmark() (*SpecialConstant method*), 64
unmarked() (*Position method*), 70
unmarked() (*SpecialConstant method*), 64
usefulStates() (*DFA method*), 28
usefulStates() (*GFA method*), 45
usefulStates() (*NFA method*), 37
usefulStates() (*OFA method*), 15

V

vDescription() (*DFA static method*), 28
vDescription() (*NFA static method*), 37
vertexIndex() (*Graph method*), 128

W

weight() (*GFA method*), 45
weightWithCycles() (*GFA method*), 45
witness() (*DFA method*), 29
witness() (*NFA method*), 37
witnessDiff() (*DFA method*), 29
Word (*class in FAdo.common*), 51
wordDerivative() (*RegExp method*), 62
wordDerivative() (*SpecialConstant method*), 64
wordGenerator() (*ADFA method*), 96
WordI() (*SSemiGroup method*), 40
wordImage() (*NFA method*), 37
WordPS() (*SSemiGroup method*), 40
words() (*FA method*), 11