# CONTENTS

1 What is FA/do? .................................................. 3
  1.1 Regular Languages ............................................ 3
  1.2 Finite Languages ............................................. 3
  1.3 Transducers .................................................. 4
  1.4 Codes ........................................................ 4

2 Module: Finite Automata (fa) ..................................... 5
  2.1 Class FA (abstract class for Finite Automata) ................... 5
  2.2 Class OFA (one-way finite automata class) ....................... 12
  2.3 Class DFA (Deterministic Finite Automata) ....................... 16
  2.4 Class NFA (Nondeterministic Finite Automata) .................... 30
  2.5 Class NFAr (Nondeterministic Finite Automata w/ reverse transition f.) . . . 39
  2.6 Class GFA (Generalized Finite Automata) ......................... 41
  2.7 Class SSemiGroup (Syntactic SemiGroup) ......................... 43
  2.8 Class EnumL (Language Enumeration) ............................ 44
  2.9 Functions .................................................... 44

3 Module: Common Definitions (common) ......................... 47
  3.1 Class Word .................................................... 47

4 Module: FA/do IO Functions (fio) ................................ 49
  4.1 Functions .................................................... 49

5 Module: Regular Expressions (reex) ......................... 51
  5.1 Class RegularExpression ......................................... 51
  5.2 Class regexp (regular expression) ................................ 51
  5.3 Class specialConstant ........................................... 56
  5.4 Class epsilon ................................................... 58
  5.5 Class emptyset .................................................. 58
  5.6 Class sigmaP .................................................... 59
  5.7 Class sigmaS .................................................... 60
  5.8 Class connective ................................................ 61
  5.9 Class star ....................................................... 62
  5.10 Class concat .................................................... 64
  5.11 Class disj ..................................................... 65
  5.12 Class power .................................................... 66
  5.13 Class option ..................................................... 68
  5.14 Class conj (intersection) ....................................... 70
  5.15 Class shuffle .................................................. 70
  5.16 Class atom ..................................................... 71
14.2 Class Grail .......................................................... 129
14.3 Functions .......................................................... 130

15 Module: Set Specification Transducers and Automata (sst) .............................................. 131
15.1 Class PSP .......................................................... 131

16 Small Tutorial ...................................................... 133

17 A small tutorial for FAdo ........................................... 135

18 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 Konstadinidis (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


Version: 1.3.5.1

Copyright: 1999-2015 Rogério Reis & Nelma Moreira {rvr,nam}@dcc.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.
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
- 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
Finite automata manipulation.
Deterministic and non-deterministic automata manipulation, conversion and evaluation.

2.1 Class FA (abstract class for Finite Automata)

class fa.FA
    Bases: common.Drawable
    Base class for Finite Automata.

    States
    set of states.
    Type list

    Sigma
    alphabet set.
    Type set

    Initial
    the initial state index.
    Type int

    Final
    set of final states indexes.
    Type set

    delta
    the transition function.
    Type dict

Note: This is just an abstract class. Not to be used directly!!

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 DFAepsilonRedefenition – if sym is Epsilon

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

addState \((name=None)\)
Adds a new state to an FA. If no name is given a new name is created.

Parameters name \((Object, \text{optional})\) – 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 \((sti)\)
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 sizes

**Returns**

string to be added to the dot file.

**Return type**

*str*

**dotDrawTransition** *(st1, sym, st2, sep)*
Draw a transition in dot format

**Parameters**

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

**dotFormat** *(size='20, 20', direction='LR', sep='\n', strict=False, maxLblSz=6)*
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 1.2.1.

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

**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.

`evalSymbol ()`
Evaluation of a single symbol

`finalP (state)`
Tests if a state is final

- Parameters `state (int)` – state index.
- Returns `is the state final?`
- Return type `bool`

`finalsP (states)`
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)`
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 `DFStateUnknown` – if a state name is unknown

`initialP (state)`
Tests if a state is initial

- Parameters `state` – state index
- Returns `is the state initial?`
Return type \texttt{bool}

\texttt{initialSet}()

The set of initial states

\texttt{Returns} set of States.

\texttt{Return type} \texttt{set}

\texttt{inputS}(i)

Input labels coming out of state i

\texttt{Parameters} i (\texttt{int}) – state

\texttt{Returns} set of input labels

\texttt{Return type} set of str

New in version 1.0.

\texttt{noBlankNames}()

Eliminates blank names

\texttt{Returns} self

\texttt{Return type} FA

\begin{center}
\textbf{Attention:} in place transformation
\end{center}

\texttt{plus}()

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

New in version 0.9.6.

\texttt{renameState}(st, name)

Rename a given state.

\texttt{Parameters}

\begin{itemize}
  \item st (\texttt{int}) – state index.
  \item name (\texttt{object}) – name.
\end{itemize}

\texttt{Returns} self.

\texttt{Return type} FA

\begin{center}
\textbf{Note:} Deals gracefully both with int and str names in the case of name collision.
\end{center}

\texttt{Attention:} the object is modified in place

\texttt{renameStates}(nameList=None)

Renames all states using a new list of names.

\texttt{Parameters} nameList (\texttt{list}) – list of new names.

\texttt{Returns} self.

\texttt{Return type} FA

\texttt{Raises} DFAerror – if provided list is too short.

\section{2.1. Class FA (abstract class for Finite Automata)}
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, s2)
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

Ags: 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 (symbolSet)
Defines the alphabet for the FA.

Parameters symbolSet (list|set) – alphabet symbols

stateAlphabet (sti)
Active alphabet for this state

Parameters sti (int) – state
Return type list

stateIndex (name, autoCreate=False)
Index of given state name.

Parameters
• name (object) – name of the state.
• autoCreate (bool, optional) – flag to create state if not already done.

Returns state index
Return type int
**Raises** `DFAstateUnknown` – if the state name is unknown and `autoCreate==False`  

**Note:** Replaces `stateName`  

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

New in version 1.0.

```python
stateName(**kwargs)
```

Index of given state name.

**Parameters**
- `name` (object) – name of the state
- `autoCreate` (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

```python
succintTransitions()
```

Colapsed transitions

```python
union(other)
```

A simple literate invocation of `__or__`

**Parameters** `other` (`FA`) – right hand operand.

**Returns** Union of self and other.

**Return type** `FA`

```python
words(stringo=True)
```

Lexicografical word generator

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

**Yields** `Word` – the next word generated.

**Attention:** Does not generate the empty word

New in version 0.9.8.

### 2.1.1 Class SemiDFA (Semi-Automata class)

```python
class fa.SemiDFA
```

Bases: `common.Drawable`

Class of automata without initial or final states

**States**
- list of states.

**Type** `list`

---

2.1. Class FA (abstract class for Finite Automata)
delta
    transition function.
    Type dict

Sigma
    alphabet.
    Type set
dotDrawState(sti, sep='\n')
    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, lbl1, st2, sep='\n')
    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', direction='LR', sep='\n')
    Dot format of automata

    Parameters
    • size (str, optional) – image size.
    • direction (str, optional) – direction of drawing.
    • sep (str, optional) – separator.

    Returns line to add to the dot file.
    Return type str

2.2 Class OFA (one-way finite automata class)
class fa.OFA
    Bases: fa.FA

    Base class for one-way automata .. inheritance-diagram:: OFA

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

Return type `reex.regexp`

Raises NotSP – if the automaton is not Serial-Parallel

See also:

Note: Automata must be Serial-Parallel

```python
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

Return type bool
```

```python
addTransition (st1, sym, st2)
Add transition :param int st1: departing state :param str sym: label :param int st2: arriving state
```

```python
allRegExps ()
Evaluates the alphabetic length of the equivalent regular expression using every possible order of state elimination.

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

```python
cutPoints ()
Set of FA’s cut points

Returns set of states

Return type set of int
```

```python
deleteStates (del_states)
To be implemented below

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

```python
static dotDrawTransition (st1, label, st2, sep='n')
Draw a transition in Dot Format

Parameters
• st1 (str) – starting state
• st2 (str) – ending state
• label (str) – symbol
• sep (str) – separator

Return type str
```

```python
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

eliminateSingles ()
Eliminates every state that only have one successor and one predecessor.

**Returns**  GFA after eliminating states
**Return type**  GFA

duplicateStout (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.

duplicateStout ()
Tests if the automaton accepts a empty language

**Return type**  bool

New in version 1.0.

evalNumberOfStateCycles ()
Evaluates the number of cycles each state participates

**Returns**  state->list of cycle lengths

**Return type**  dict

evalSymbol ()
Eval symbol

finalCompP (s)
To be implemented below

**Parameters**  s – state

**Return type**  list

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

reCG ()
Regular expression from state elimination whose language is recognised by the FA. Uses a heuristic to choose the order of elimination.
Returns the equivalent regular expression

Return type reex.regexp

reCG_nn()

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.

Returns the equivalent regular expression

Return type reex.regexp

reDynamicCycleHeuristic()

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

Returns an equivalent regular expression

Return type reex.regexp

See also:


reStaticCycleHeuristic()

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

Returns a equivalent regular expression

Return type reex.regexp

See also:


re_stateElimination(order=None)

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

Parameters order (list) – state elimination sequence

Returns the equivalent regular expression

Return type reex.regexp

re_stateElimination_nn(order=None)

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

Parameters order (list) – state elimination sequence

Returns the equivalent regular expression

Return type reex.regexp

regexpSE()

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

Returns the equivalent regular expression

Return type reex.regexp
stateChildren(_, a=None)
   To be implemented below
   Parameters s – state
   Return type list

succintTransitions()
   Collapsed transitions

toGFA()
   To be implemented below

topoSort()
   Topological order for the FA
   Returns List of state indexes
   Return type list of int

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.

Attention: in place transformation

trimP()
   Tests if the FA is trim: initially connected and co-accessible
   Returns bool

uniqueRepr()
   Abstract method

usefulStates()
   To be implemented below

2.3 Class DFA (Deterministic Finite Automata)

class fa.DFA
   Bases: fa.OFA
   Class for Deterministic Finite Automata.
**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 –

**Returns** bool

See also:

**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 Algorithm, Symposium on Theoretical Aspects of Computer Science

**aEquiv ()**
Computes almost equivalence, used by hyperMinimal

**Returns** partition of states

**Return type** dictionary

**Note:** may be optimized to avoid duped

**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 **DFA\_not\_NFA** – if one tries to add a non deterministic transition

```python
compat (s1, s2, data)
```
Tests compatibility between two states.

Parameters

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

Return type `bool`

```python
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.

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

Return type `DFA`

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

```python
completeP()
```
Checks if it is a complete FA (if delta is total)

Returns `bool`

```python
completeProduct (other)
```
Product structure

Parameters **other** – the other DFA

```python
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.

---

**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.

---

**deleteState** *(sti)*
Delete state from a DFA

**Parameters** `sti` *(int)* – state index

---

**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.

\textbf{static deterministicP} ()
Yes it is deterministic!
\textbf{Return type} bool

\textbf{dist} ()
Evaluate the distinguishability language for a DFA
\textbf{Return type} DFA
\textbf{See also:}
Cezar Câmpeanu, Nelma Moreira, Rogério Reis: The distinguishability operation on regular languages. NCMA 2014: 85-100
New in version 0.9.8.

\textbf{distMin} ()
Evaluates the list of minimal words that distinguish each pair of states
\textbf{Returns} set of minimal distinguishing words
\textbf{Return type} FL
New in version 0.9.8.

\textbf{Attention:} If the DFA is not minimal, the method loops forever

\textbf{distR} ()
Evaluate the right distinguishability language for a DFA
\textbf{Return type} DFA
\textbf{..seealso::} Cezar Câmpeanu, Nelma Moreira, Rogério Reis: The distinguishability operation on regular languages. NCMA 2014: 85-100

\textbf{distRMin} ()
Compute distRMin for DFA
:rtype FL
\textbf{..seealso::} Cezar Câmpeanu, Nelma Moreira, Rogério Reis: The distinguishability operation on regular languages. NCMA 2014: 85-100

\textbf{distTS} ()
Evaluate the two-sided distinguishability language for a DFA
\textbf{Return type} DFA
\textbf{..seealso::} Cezar Câmpeanu, Nelma Moreira, Rogério Reis: The distinguishability operation on regular languages. NCMA 2014: 85-100

\textbf{dup} ()
Duplicate the basic structure into a new DFA. Basically a copy.deep.
\textbf{Return type} DFA
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
• DFAunknown – 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 DFAunknown – 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

2.3. Class DFA (Deterministic Finite Automata)
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 \( (\text{word}, \text{initial}=None) \)
Verifies if the DFA recognises a given word
Parameters

- \( \text{word} (\text{list of symbols.}) \) – word to be recognised
- \( \text{initial} (\text{int}) \) – starting state index

Return type bool

finalCompP \( (s) \)
Verifies if there is a final state in strongly connected component containing \( s \).
Parameters \( s (\text{int}) \) – 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 \( (\text{strict}=False) \)
Hyperminization of a minimal DFA
Parameters \( \text{strict} (\text{bool}) \) – if strict=True it first minimizes the DFA
Returns an hyperminimal DFA
Return type DFA

See also:

Note: if strict=False minimality is assumed

inDegree \( (st) \)
Returns the in-degree of a given state in an FA
Parameters \( st (\text{int}) \) – 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 list (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 `trim()`.

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 nlog{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:

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 Watson’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 \texttt{bool}

\texttt{orderedStrConnComponents}()

Topological ordered list of strong components

New in version 1.3.3.

Return type \texttt{list}

\texttt{pairGraph}()

Returns pair graph

\textbf{Return type} \texttt{DiGraphVM}

See also:


\texttt{possibleToReverse}()

Tests if language is reversible

New in version 1.3.3.

\texttt{pref}()

Returns a dfa that recognizes \texttt{pref(L(self))}

\textbf{Return type} \texttt{DFA}

New in version 1.1.

\texttt{print\_data(data)}

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

\textbf{Parameters} \texttt{data} – data to print

\texttt{product(other)}

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

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

\textbf{Parameters} \texttt{other} – the other DFA

\textbf{Return type} \texttt{DFA}

\texttt{productSlow(other, complete=True)}

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

\textbf{Note:} this is a slow implementation for those that need meaningfull state names

New in version 1.3.3.

\textbf{Parameters}

\begin{itemize}
  \item \texttt{other} – the other DFA
  \item \texttt{complete (bool)} – evaluate product as a complete DFA
\end{itemize}

\textbf{Return type} \texttt{DFA}
regexp ()
   Returns a regexp for the current DFA considering the recursive method. Very inefficient.
   
   Returns  a regexp equivalent to the current DFA
   Return type  reex.regexp

reorder (dict)
   Reorders states according to given dictionary. Given a dictionary (not necessarily complete)... reorders states accordingly.
   
   Parameters  dict (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)
   Shuffle of two languages: L1 W L2
   
   Parameters
   
   • other (DFA) – second automaton
   • strict (bool) – should the alphabets be necessary equal?
   
   Return type  DFA

See also:


simDiff (other)
   Symmetrical difference
   
   Parameters  other –
   Returns

sop (other)
   Strange operation
   
   Parameters  other (DFA) – the other automaton
   Return type  DFA
See also:


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 -&gt; multiplicity

**Return type** dictionary

**stronglyConnectedComponents** ()
Dummy method that uses the NFA conterpart

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

csyncWords()
Evaluates the regular expression corresponding to the synchronizing pwords of the automata.

Returns a regular expression of the sync words of the automata

Return type reex.regexp
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
toGFA()
Creates a GFA equivalent to DFA

Returns GFA deep copy

Return type GFA
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.

See also:

Returns normalised representation

Return type list

Raises DFAnotComplete – if DFA is not complete
universalP (minimal=False)
Checks if the automaton is universal through minimisation

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

Return type bool

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

Returns a NFA

Return type NFA

usefulStates (initial_states=None)
Set of states 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 emptyness

Returns word

Return type str

witnessDiff (other)
Returns a witness for the difference of two DFAs and:

<table>
<thead>
<tr>
<th>0</th>
<th>if the witness belongs to the other language</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>if the witness belongs to the self language</td>
</tr>
</tbody>
</table>

Parameters other (DFA) – the other DFA

Returns a witness word

Return type list of symbols

Raises DFAequivalent – if automata are equivalent

2.4 Class NFA (Nondeterministic Finite Automata)

class fa.NFA
Bases: fa.OFA

Class for Non-deterministic Finite Automata (epsilon-transitions allowed).
**HKeqP** *(other, strict=True)*  
Test NFA equivalence with extended Hopcroft-Karp method

**See also:**  

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

**addTransitionQ** *(srcI, dest, symb, qfuture, qpast)*  
Add transition to the new transducer instance.

**Parameters**  
- **qpast** *(set)* – past queue  
- **qfuture** *(set)* – future queue  
- **symb** – symbol  
- **dest** – 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 epsilon transitions from the states in the epsilon 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 (NFA|DFA) – the other NFA

Returns the result of the concatenation

Return type NFA

countTransitions()
Number of transitions of a NFA

Return type int

delTransition(st1, 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 (str) – symbol consumed
• _no_check (bool) – dismiss secure code

Note: unused alphabet symbols will be discarded from Sigma.
deleteStates \((\text{del\_states})\)

Delete given iterable collection of states from the automaton.

**Parameters**

- **del\_states** \((\text{set/list})\) – 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 \((\text{generic=False})\)

Computes the determination uppon a followFromPosition result

**Return type** \(\text{NFA}\)

deterministicP ()

Verify whether this NFA is actually deterministic

**Return type** \(\text{bool}\)

dotFormat \((\text{size='20, 20'}, \text{direction='LR'}, \text{sep='\n'}, \text{strict=False}, \text{maxLblSz=6})\)

A dot representation

**Parameters**

- **direction** \((\text{str})\) – direction of drawing
- **size** \((\text{str})\) – size of image
- **sep** \((\text{str})\) – line separator
- **maxLblSz** – max size of labels before getting removed
- **strict** – use limitations of label sizes

**Returns** the dot representation

**Return type** \(\text{str}\)

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** \(\text{NFA}\)

elimEpsilon ()

Eliminate epsilon-transitions from this automaton.

:**type:** \(\text{NFA}\)

**Attention:** performs in place modification of automaton

Changed in version 1.1.1.

eliminateEpsilonTransitions ()

Eliminates all epslilon-transitions with no state addition

**Attention:** in-place modification
eliminateTSymbol(symbol)
Delete all 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.

epsilonClosure(st)
Returns the set of states epsilon-connected to from given state or set of states.

    Parameters st (int/set) – state index or set of state indexes
    Returns the list of state indexes epsilon connected to st
    Return type set of int

| Attention: st must exist. |

epsilonP()
Whether this NFA has epsilon-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 epsilon 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 epsilon closure.

    Parameters
      • stil (set/list) – set of current states
      • sym (str) – symbol to be consumed
    Returns set of reached state indexes
    Return type set
    Raises DFAUnknown – 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

oldAnd(other)
Conjunction of automata

Parameters other (NFA|DFA) – the right hand operand

Return type: NFA

Raises FAdoGeneralError – if any operand is not an NFA

product(other)
Returns a NFA (skeletom) 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

**Return type**

- NFA

**star** *(flag=False)*
Kleene star of a NFA

**Parameters**

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

**Returns**

- the resulting NFA

**Return type**

- NFA

**stateChildren** *(state, 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

**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 epsilon-NFA

**toGFA()**
Creates a GFA equivalent to NFA

Returns  a GFA deep copy

Return type  **GFA**

**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

New in version 0.9.5.

Returns  the interface list

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

Returns  the set of ending states
2.5 Class NFAr (Nondeterministic Finite Automata w/ reverse transition f.)

```python
class fa.NFAr
    Bases: fa.NFA

Class for Non-deterministic Finite Automata with reverse delta function added by construction.
```

**Variables** `deltaReverse` – the reversed transition function

**Note:** 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

### elimEpsilonO()

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

**Returns** itself
Return type

Attention: performs inplace modification of automaton

**homogenousP** (*inplace=False*)
Checks if the automaton is homogenous, i.e. the transitions that reach a state have all the same label.

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

**Returns** True if homogenous

**Return type** bool

**mergeStates** (*f, t*)
Merge the first given state into the second. If first state is an initial or final state, the second becomes respectively an initial or final state.

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

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

**mergeStatesSet** (*tomerge, target=None*)
Merge a set of states with a target merge state. If the states in the set have transitions among them, those transitions will be directly merged into the target state.

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

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

Attention: The states of the list will become 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 epsilon, 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 epsilon, 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.

### 2.6 Class GFA (Generalized Finite Automata)

**class fa.GFA**

- **Bases:** `fa.OFA`

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

```python
common.Drawlable -> fa.FA -> fa.OFA -> fa.GFA
```

**DFS (io)**

Depth first search

- **Parameters**
  - `io`

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

Adds a new transition from `st1` to `st2` consuming symbol `sym`. Label of the transition function is a regexp.

- **Parameters**
  - `st1 (int)` – state index of departure
  - `st2 (int)` – state index of arrival
  - `sym (str)` – symbol consumed

  **Raises**
  - `DFAepsilonRedefenition` – if `sym` is Epsilon

**assignLow (st)**

- **Parameters**
  - `st`

**assignNum (st)**
Parameters \texttt{st} –

\texttt{completeDelta()}\newline
Adds empty set transitions between the automatons final and initial states in order to make it complete. It’s only meant to be used in the final stage of SEA…

\texttt{deleteState(st)}\newline
deletes a state from the GFA :param \texttt{sti}:

\texttt{deleteStates(del\_states)}\newline
To be implemented below

Parameters \texttt{del\_states(list)} – states to be deleted

\texttt{dfs\_visit(s, visited, io)} –

Parameters

• \texttt{s} – state

• \texttt{visited} – list of states visited

• \texttt{io} –

\texttt{dup()}\newline
Returns a copy of a GFA

Return type \texttt{GFA}

\texttt{eliminate(st)}\newline
Eliminate a state.

Parameters \texttt{st(int)} – state to be eliminated

\texttt{eliminateAll(lr)}\newline
Eliminate a list of states.

Parameters \texttt{lr(list)} – list of states indexes

\texttt{eliminateState(st)}\newline
Deletes a state and updates the automaton

Parameters \texttt{st(int)} – the state to be deleted

\texttt{evalSymbol()}\newline
Eval symbol

\texttt{finalCompP(s)}\newline
To be implemented below

Parameters \texttt{s} – state

Return type \texttt{list}

\texttt{initialComp()}\newline
Initial component

Return type \texttt{list}

\texttt{normalize()}\newline
Create a single initial and final state with Epsilon transitions.

\textbf{Attention:} works in place
reorder \( (\text{dictio}) \)
Reorder states indexes according to given dictionary.

**Parameters**

\( \text{dictio (dict)} \) – order

**Note:** dictionary does not have to be complete

\( \text{stateChildren (state, strict=False)} \)
Set of children of a state

**Parameters**

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

**Returns**

map: children -> alphabetic length

**Return type**

dictionary

\( \text{succintTransitions ()} \)
Collapsed transitions

\( \text{toGFA ()} \)
To be implemented below

\( \text{uniqueRepr ()} \)
Abstract method

\( \text{usefulStates ()} \)
To be implemented below

\( \text{weight (state)} \)
Calculates the weight of a state based on a heuristic

**Parameters**

- \( \text{state (int)} \) – state

**Returns**

the weight of the state

**Return type**

int

\( \text{weightWithCycles (state, cycles)} \)

**Parameters**

- \( \text{state} \)
- \( \text{cycles} \)

**Returns**


2.7 Class SSemiGroup (Syntactic SemiGroup)

class fa.SSemiGroup

**Bases:** object

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

### 2.8 Class EnumL (Language Enumeration)

### 2.9 Functions

**fa.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
```python
fa.stringToDFA(s, f, n, k)
```

Converts a string to DFA’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
3.1 Class Word

class common.Word(data=None, it=None)

    Bases: object

    Class to implement generic words as iterables with pretty-print

    Basically a unified way to deal with words with characters of different sizes without much fuss
4.1 Functions

`fio.readFile(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 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 an NFA declaration, the “symbol” `@epsilon` is interpreted as a epsilon-transition
- the source state of the first transition is the initial state
- in the case of an 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 single name, declares a state
\[ \text{dTrans} ::= \text{stateid symbol stateid} | \]
\[ \text{stateid symbol stateid CR dTrans} \]
\[ \text{nTrans} ::= \text{stateid nSymbol stateid} | \]
\[ \text{stateid nSymbol stateid CR nTrans} \]
\[ \text{tTrans} ::= \text{stateid nSymbol nSymbol stateid} | \]
\[ \text{stateid nSymbol nSymbol stateid CR nTrans} \]

**Note:** If an error occurs, 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.

\[ \text{fio.saveToFile}(\text{FileName, fa, mode='a'}) \]

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**

- \[ \text{FileName (str)} \] – file name
- \[ \text{fa (list of FA)} \] – the FA
- \[ \text{mode (str)} \] – writing mode
Regular expressions manipulation

Regular expression classes and manipulation

5.1 Class RegularExpression

class reex.RegularExpression
   Bases: object

   Abstract base class for all regular expression objects

5.2 Class regexp (regular expression)

class reex.regexp(sigma=None)
   Bases: reexRegularExpression

   Base class for regular expressions.

   Variables  Sigma – alphabet set of strings

   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.

**Parameters**
- *r*: RE
- *nfa_method*: 'nfaPosition'

**Returns**
- RE

**See also:**
- Andrea Asperti, Claudio Sacerdoti Coen and Enrico Tassi, Regular Expressions, au point. arXiv 2010
- Tobias Nipkow and Dmitriy Traytel, Unified Decision Procedures for Regular Expression Equivalence

**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
- 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:**

**dfaYMG**()

DFA Yamada-McNaughton-Glushkov 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

**epsilonLength**()

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

**Return type** integer

**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*: RE
- *strict*: if True checks for same alphabets
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
ewp ()
Whether the empty word property holds for this regular expression’s language.
Return type Boolean
first ()
Return type set
last ()
Return type set
linearForm ()
Return type dic
mark ()
Make all atoms maked (tag False) :rtype: reex.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 reex.regexp
See also:
. . 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 epsilon transitions
Return type  NFAe

Note:  The regular expression must be reduced

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

nfaGlushkov()
Position or Glushkov automaton of the regular expression. Recursive method.

Returns  NFA

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

Returns  partial derivatives [or equation] automaton

Return type  NFA

See also:

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 dijunct

Returns: position NFA
Return type: NFA

nfaPre()
Prefix NFA of a regular expression
:return: prefix automaton
:type: NFA

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

notEmptyW()
Witness of non emptiness

Returns: word or None

rpn()
RPN representation
:type: str
:return: printable RPN representation

setOfSymbols()

Returns: 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()

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

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

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

treeNode()
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:


### 5.3 Class specialConstant

class `reex.specialConstant` *(sigma=\texttt{None})*

Bases: `reex.regexp`

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 **first** *(parent_first=\texttt{None})*
  Parameters **parent_first** –
  Returns
- **followLists** *(lists=\texttt{None})*
  Parameters **lists** –
  Returns
- **followListsD** *(lists=\texttt{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: reex.regexp

partialDerivativesC (sigma)
  Parameters sigma –
  Returns

reversal()
  Reversal of regexp
    Return type reex.regexp

rpn()
  RPN representation :rtype: str :return: printable RPN representation

static setOfSymbols()
  Returns

support (side=True)
  Returns

supportlast (side=True)
  Returns

tailForm()
  Returns

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 epsilon() or the emptyset().
    Return type (general) regular expression

wordDerivative (word)
  Parameters word –
  Returns
5.4 Class epsilon

class reex.epsilon (sigma=None)
    Bases: reex.specialConstant

Class that represents the empty word.

Parameters sigma – alphabet

static epsilonLength ()
    Return type int

static epsilonP ()
    Return type bool

static ewp ()
    Return type bool

static measure (from_parent=None)
    Parameters from_parent –
    Returns measures

nfaThompson()
    Return type NFA

static partialDerivatives(_)
    Returns

partialDerivativesC(_)
    Parameters sigma –
    Returns

rpn()
    Returns str

snf (_hollowdot=False)
    Parameters _hollowdot –
    Returns

5.5 Class emptyset

class reex.emptyset (sigma=None)
    Bases: reex.specialConstant
Class that represents the empty set.

Parameters sigma – alphabet

static emptysetP()
  Returns

epsilonLength()
  Returns

epsilonP()
  Returns
ewp()
  Returns

static measure(from_parent=None)
  Parameters from_parent –
  Returns

partialDerivativesC(_)
  Parameters sigma –
  Returns

rpn()
  Returns

5.6 Class sigmaP

class reex.sigmaP(sigma=None)
  Bases: reex.specialConstant

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

sigmaP: Class that represents the complement of the emptyset word (Sigma^+)

Parameters sigma – alphabet
5.7 Class sigmaS

class reex.sigmaS (sigma=None)

Bases: reex.specialConstant

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

sigmaS: Class that represents the complement of the emptyset set (Sigma^*)

Parameters sigma – alphabet

derivative (sigma)

Parameters sigma –

Returns

ewp()

Returns

linearForm()

Returns

linearFormC()

Returns

partialDerivatives (sigma)

Parameters sigma –

Returns

static partialDerivativesC (_)

Parameters _ –

Returns

support (side=True)

Returns
Returns

```
linearFormC()
```

Returns

```
partialDerivatives(sigma)
```

Parameters `sigma` –

Returns

```
partialDerivativesC(sigma)
```

Parameters `sigma` –

Returns

```
support(side=True)
```

Returns

## 5.8 Class `connective`

```python
class reex.connective(arg1, arg2, sigma=None)
    Bases: reex.regexp
```

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

**linearForm()**

**Return type** dic

---

5.8. Class `connective` 61
mark()
Make all atoms maked (tag False) :rtype: 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

treeLength()
Number of nodes of the regular expression’s syntactical tree.
Return type integer

5.9 Class star

class reex.star(arg, sigma=None)
Bases: reex.regexp
Class for iteration operation (aka Kleene star, or Kleene closure) 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

ewp()
Whether the empty word property holds for this regular expression’s language.
Return type Boolean

first(parent_first=None)
Return type set
last \((parent.first=None)\)

Return type \(\text{set}\)

linearForm()

Return type \(\text{dic}\)

mark()

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

nfaThompson()

Returns a NFA that accepts the RE.

Return type \(\text{NFA}\)

reversal()

Reversal of regexp

Return type \(\text{reex.regexp}\)

rpn()

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

setOfSymbols()

Return type \(\text{set}\)

5.9. Class star
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

unmark()
  Conversion back to regexp
  
  Return type reex.star

5.10 Class concat

class reex.concat(arg1, arg2, sigma=None)
  Bases: reex.connective

  Class for catenation operation on regular expressions.

  reex.RegularExpression  reex.regex  reex.connective  reex.concat

ewp()
  Whether the empty word property holds for this regular expression’s language.
  
  Return type Boolean

first(parent_first=None)
  
  Return type set

last(parent_last=None)
  
  Return type set

linearForm()
  
  Return type dic

mark()
  Make all atoms maked (tag False) :rtype: reex.regex
  
  reversal()
  
  Reversal of regexp
  
  Return type reex.regex

rpn()
5.11 Class disj

class reex.disj(arg1, arg2, sigma=None)
   Bases: reex.connective

Class for disjunction operation on regular expressions.

ewp()
   Whether the empty word property holds for this regular expression’s language.
      Return type  Boolean

first (parent_first=None)
   Return type  set

last (parent_last=None)
   Return type  set

linearForm()
   Return type  dict

mark()
   Conversion to marked atoms :rtype: disj

nfaThompson()
   Returns an NFA (Thompson) that accepts the RE.
      Return type  NFA
reversal()
    Reversal of regexp
    
    Return type  reex.regexp

rpn()
    RPN representation :rtype: str :return: printable RPN representation

tailForm()
    
    Return type  dict

unmark()
    Conversion back to unmarked atoms :rtype: disj

5.12 Class power

class reex.power(arg, n=1, sigma=None)
    Bases: reex.regexp

    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: reex.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

treeLength()
Number of nodes of the regular expression’s syntactical tree.

Return type integer
5.13 Class option

class reex.option(arg, sigma=None)
Bases: reex.regexp

Class for option operation on regular expressions.

reex.RegularExpression → reex.regexp → reex.option

alphabetLength()
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

ewp()
Whether the empty word property holds for this regular expression’s language.

Return type  Boolean

first (parent_first=None)
Return type  set

followListsStar (lists=None)
to be fixed

last (parent_first=None)
Return type  set

linearForm()
Return type  dic

mark()
Make all atoms marked (tag False) :rtype: reex.regexp

nfaThompson()
Returns a NFA that accepts the RE.

Return type  NFA
reversal()
Reversal of regexp

Return type: reex.regex

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
5.14 Class conj (intersection)

class reex.conj(arg1, arg2, sigma=None)  
Bases: reex.connective

Intersection operation of regexps

ewp()  
    Whether the empty word property holds for this regular expression’s language.

    Return type  Boolean

linearForm()  
    Return type  dic

mark()  
    Make all atoms maked (tag False) :rtype: reex.regexp

rpn()  
    RPN representation :rtype: str :return: printable RPN representation

support (side=True)

5.15 Class shuffle

class reex.shuffle(arg1, arg2, sigma=None)  
Bases: reex.connective

Shuffle operation of regexps

ewp()  
    Whether the empty word property holds for this regular expression’s language.

    Return type  Boolean

first (parent_list=None)  
    Parameters parent_list –

    Returns

followListsD (lists=None)  
    in progress

linearForm()  
    Return type  dic

mark()  
    Make all atoms maked (tag False) :rtype: reex.regexp

rpn()  
    RPN representation :rtype: str :return: printable RPN representation

support (side=True)

supportlast (side=True)
5.16 Class atom

class reex.atom(val, sigma=None)
    Bases: reex.regexp

    Simple atom (symbol)

    Variables
    • Sigma – alphabet set of strings
    • val – the actual symbol

Constructor of a regular expression symbol.

    Parameters val – the actual symbol

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

    Returns set of regular expressions

    Return type set

    See also:
    Antimirov, 95

static alphabeticLength()
    Number of occurrences of alphabet symbols in the regular expression.

    Return type integer

    Attention: Doesn’t include the empty word.

derivative(sigma)
    Derivative of the regular expression in relation to the given symbol.

    Parameters sigma – an arbitrary symbol.

    Return type regular expression

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

    See also:
static epsilonLength()
Number of occurrences of the empty word in the regular expression.

Return type integer

first (parent_first=None)
List of possible symbols matching the first symbol of a string in the language of the regular expression.

Parameters parent_first –

Returns list of symbols

followLists (lists=None)
Map of each symbol’s follow list in the regular expression.

Parameters lists –

Returns map of symbols’ follow lists
Return type {symbol: list of symbols}

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

followListsD (lists=None)
Map of each symbol’s follow list in the regular expression.

Parameters lists –

Returns map of symbols’ follow lists
Return type {symbol: list of symbols}

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

See also:

followListsStar (lists=None)
Map of each symbol’s follow list in the regular expression under a star.

Parameters lists –

Returns map of symbols’ follow lists
Return type {symbol: list of symbols}

last (parent_last=None)
List of possible symbols matching the last symbol of a string in the language of the regular expression.

Parameters parent_last –

Returns list of symbols
linearForm()

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

Returns dictionary mapping heads to sets of tails

Return type \{symbol: set([regular expressions])\}

See also:
Antimirov, 95

linearFormC()

Returns

linearP()

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

Return type \text{boolean}

mark()

Return type \text{m_atom}

static measure(from_parent=None)

A list with four measures for regular expressions.

Parameters from_parent –

Return type \[\text{int}, \text{int}, \text{int}, \text{int}\]

[alphabeticLength, treeLength, epsilonLength, starHeight]

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

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

nfaThompson()

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

Return type \text{NFA}

See also:
K. Thompson. Regular Expression Search Algorithm. CACM 11(6), 419-422 (1968)
partialDerivatives(\textit{sigma})
Set of partial derivatives of the regular expression in relation to given symbol.

\textbf{Parameters} \textit{sigma} – symbol in relation to which the derivative will be calculated.

\textbf{Returns} set of regular expressions

See also:
Antimirov, 95

partialDerivativesC(\textit{sigma})

\textbf{Parameters} \textit{sigma} –

\textbf{Returns}

\textbf{reduced}(\textit{hasEpsilon}=\text{False})
Equivalent regular expression with the following cases simplified:

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

\textbf{Parameters} \textit{hasEpsilon} – used internally to indicate that the language of which this term is a subterm has the empty word.

\textbf{Returns} regular expression

\begin{itemize}
\item \textbf{Attention:} Returned structure isn’t strictly a duplicate. Use \texttt{\_\_copy\_\_()} for that purpose.
\end{itemize}

reversal()
Reversal of regexp

\textbf{RPN representation}:return: printable RPN representation

setOfSymbols()
Set of symbols that occur in a regular expression..

\textbf{Returns} set of symbols

\textbf{Return type} set of symbols

snf(\textit{hollowdot}=\text{False})
Star Normal Form (SNF) of the regular expression.

\textbf{Parameters} \textit{hollowdot} –

\textbf{Returns} regular expression in star normal form
static starHeight()
Maximum level of nested regular expressions with a star operation applied.
For instance, starHeight(((a*b)*+b*)*) is 3.
Return type integer

stringLength()
Length of the string representation of the regular expression.
Return type integer

support(side=True)
Support of a regular expression. :param side: if True concatenation of a set on the left if False on the right (prefix support) :return: set of regular expressions :rtype: set
See also:
See also:

supportlast(side=True)
Subset of support such that elements have ewp

static syntacticLength()
Number of nodes of the regular expression’s syntactical tree (sets).
Return type integer

tailForm()
Returns

static treeLength()
Number of nodes of the regular expression’s syntactical tree.
Return type integer

unmarked()
The unmarked form of the regular expression. Each leaf in its syntactical tree becomes a regexp(), the epsilon() or the emptyset().
Return type (general) regular expression

5.17 Class position

class reex.position(val, sigma=None)
Bases: reex.atom
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 epsilon() or the emptyset().

**Return type** (general) regular expression

### 5.18 Class ParseReg

class reex.ParseReg (no_table=1, table='.tableambreg')  
Bases: reex.ParseReg1

A parser for regular expressions with ambiguous rules: not working

### 5.19 Class sconnective (special connective)

class reex.sconnective (arg, sigma=None)  
Bases: reex.regexp

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

associativity of concatenation; identities Sigma^* and Sigma^+. Connectives are: sdisj: disjunction  
sconj: intersection sconcat: concatenation

For parsing use str2sre
 alphabeticLength()  
Returns
 epsilonLength()  
Returns
 first()  
Return type set
 last()  
Return type set
 linearForm()  
Return type dic
 mark()  
Make all atoms maked (tag False) :rtype: reex.regexp
 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
 treeLength()  
Returns

5.20 Class sconcat

class reex.sconcat (arg, sigma=None)  
Bases: reex.sconnective

Class that represents the concatenation operation.
derivative(supa)
  Parameters sigma –
  Returns

ewp()
  Returns

head()
  Returns

head_rev()
  Returns

linearForm()
  Returns

linearFormC()
  Returns

partialDerivatives(supa)
  Parameters sigma –
  Returns

partialDerivativesC(supa)
  Parameters sigma –
  Returns

support(side=True)
  Returns

tail()
  Returns

tail_rev()
  Returns

\section{Class sstar}

class \texttt{reex.sstar}(arg, sigma=None)
  Bases: \texttt{reex.star}
Special regular expressions modulo associativity, commutativity, idempotence of disjunction and intersection; associativity of concatenation; identities $\Sigma^*$ and $\Sigma^+$.  
sstar: Class that represents Kleene star

```python
def derivative(sigma):
    Parameters sigma --
    Returns

def linearForm():
    Returns

def partialDerivatives(sigma):
    Parameters sigma --
    Returns

def partialDerivativesC(sigma):
    Parameters sigma --
    Returns

def support(side=True):
    Returns
```

5.22 Class sdisj

```python
class reex.sdisj(arg, sigma=None):
    Bases: reex.sconnective
    Class that represents the disjunction operation for special regular expressions.
```

cross(ri, s, lists)
    Returns
def derivative(sigma)
Parameters \( \sigma \)
Returns
\texttt{ewp}()
Returns
\texttt{first}()
Returns
\texttt{followLists}(\texttt{lists}=\texttt{None})
Parameters \texttt{lists} –
Returns
\texttt{followListsStar}(\texttt{lists}=\texttt{None})
Parameters \texttt{lists} –
Returns
\texttt{last}()
Returns
\texttt{linearForm}()
Returns
\texttt{linearFormC}()
Returns
\texttt{partialDerivatives}(\texttt{sigma})
Parameters \texttt{sigma} –
Returns
\texttt{partialDerivativesC}(\texttt{sigma})
Parameters \texttt{sigma} –
Returns
\texttt{support}(\texttt{side}=\texttt{True})
Returns

### 5.23 Class \texttt{sconj}

\begin{verbatim}
class \texttt{reex.sconj}(\texttt{arg}, \texttt{sigma}=\texttt{None})
Bases: \texttt{reex.sconnective}

Class that represents the conjunction operation.
\end{verbatim}
derivative (sigma)
  Parameters sigma –
  Returns

ewp()
  Returns

linearForm()
  Returns

partialDerivatives (sigma)
  Parameters sigma –
  Returns

partialDerivativesC (sigma)
  Parameters sigma –
  Returns

support (side=True)
  Returns

5.24 Class snot

class reex.snot (arg, sigma=set([]))
  Bases: reex.regexp

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

alphabeticLength()
  Returns

derivative (sigma)
  :param sigma :return:
epsilonLength()
  Returns

ewp()
  Returns

first()
  Return type set

last()
  Return type set

linearForm()
  Returns

linearFormC()
  Returns

mark()
  Make all atoms maked (tag False) :rtype: reex.regexp

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

treeLength()
  Returns
5.25 Functions

\texttt{reex.str2regexp}(s, parser=\texttt{\emph{reex.ParseRegl}}, no\_table=1, sigma=\texttt{None}, strict=\texttt{False})

Reads a regexp from string.

\begin{itemize}
  \item \texttt{s}(\texttt{string}) – the string representation of the regular expression
  \item \texttt{parser}(\texttt{Yappy}) – a parser generator for regexps
  \item \texttt{no\_table}(\texttt{integer}) – if 0 table is created
  \item \texttt{sigma}(\texttt{list or set of symbols}) – alphabet of the regular expression
  \item \texttt{strict}(\texttt{boolean}) – if True tests if the symbols of the regular expression are included in sigma
\end{itemize}

\textbf{Return type} \texttt{reex.regexp}

\texttt{reex.str2sre}(s, parser=\texttt{\emph{reex.ParseS}}, no\_table=1, sigma=\texttt{None}, strict=\texttt{False})

Reads a sre from string. Arguments as \texttt{str2regexp}.

\textbf{Return type} \texttt{regexp}

\texttt{reex.rpn2regexp}(s, sigma=\texttt{None}, strict=\texttt{False})

Reads a (simple) regexp from a RPN representation

\begin{align*}
R & ::= \texttt{.RR | +RR | \*R | L | @} \\
L & ::= \texttt{[a-z] | [A-Z]}
\end{align*}

\textbf{Parameters} \texttt{s}(\texttt{string}) – RPN representation

\textbf{Return type} \texttt{reex.regexp}

\textbf{Note:} This method uses python stack... thus depth limitations apply
CHAPTER SIX

MODULE: TRANSDUCERS (TRANSDUCERS)

Finite Tranducer Support
Transducer manipulation.
New in version 1.0.

6.1 Class Transducer

class transducers.Transducer
  Bases: fa.NFA
  Base class for Transducers

  setOutput(listOfSymbols)
  Set Output
  Parameters listOfSymbols(set/list) – output symbols

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

6.2 Class SFT (Standard Form Transducers)

class transducers.SFT
  Bases: transducers.GFT
  Standard Form Tranducer
  Variables Output(set) – output alphabet
**addEpsilonLoops()**
Add a loop transition with epsilon input and output to every state in the transducer.

**addTransition(stsrc, symi, symo, sti2)**
Adds a new transition

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

**addTransitionProductQ(src, dest, ddest, sym, out, futQ, pastQ)**
Add transition to the new transducer instance.

**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 \textit{SFT}

\texttt{concat}(other)
Concatenation of transducers

\textbf{Parameters}\ \texttt{other}(\textit{SFT}) – the other operand

\textbf{Return type} \textit{SFT}

\texttt{delTransition}(sti, sym, symo, sti2, \texttt{_no_check}=False)
Remove a transition if existing and perform cleanup on the transition function’s internal data structure.

\textbf{Parameters}

- \texttt{symo} – symbol output
- \texttt{sti1} (\textit{int}) – state index of departure
- \texttt{sti2} (\textit{int}) – state index of arrival
- \texttt{sym} – symbol consumed
- \texttt{_no_check} (\textit{bool}) – dismiss secure code

\texttt{deleteState}(sti)
Remove given state and transitions related with that state.

\textbf{Parameters}\ \texttt{sti}(\textit{int}) – index of the state to be removed

\textbf{Raises} \texttt{DFAstateUnknown} – if state index does not exist

\texttt{deleteStates}\ (\texttt{lstates})
Delete given iterable collection of states from the automaton.

\textbf{Parameters}\ \texttt{lstates}(\textit{set|list}) – collection of int representing states

\texttt{dup}()
Duplicate of itself :rtype: \textit{SFT}

\begin{center}
\textbf{Attention: } only duplicates the initially connected component
\end{center}

\texttt{emptyP}()
Tests if the relation realized the empty transducer

\textbf{Return type} \textit{bool}

\texttt{epsilonOutP}()
Tests if epsilon occurs in transition outputs

\textbf{Return type} \textit{bool}

\texttt{epsilonP}()
Test whether this transducer has input epsilon-transitions

\textbf{Return type} \textit{bool}

\texttt{evalWordP}(wp)
Tests whether the transducer returns the second word using the first one as input

\textbf{Parameters}\ \texttt{wp}(\textit{tuple}) – pair of words

\textbf{Return type} \textit{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:

See also:

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 meaningful state names.

Note: The resulting transducer is not trim.

Parameters other \(\text{DFA}/\text{NFA}\) – 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 meaningful names of states.

Parameters other \(\text{DFA}/\text{NFA}\) – the automata needs to be operated.

Return type SFT

inverse()
Switch the input label with the output label.
No initial or final state changed.

Returns Transducer with transitions switched.

Return type SFT

nonEmptyW()
Witness of non emptyness
**nonFunctionalW**()

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

- **Returns**: witness
- **Return type**: tuple

**outIntersection**(*other*)

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

- **Parameters** *other* (*DFA|NFA*) – the automaton used as a filter of the output
- **Return type**: SFT

**outIntersectionDerived**(*other*)

Naive version of outIntersection

- **Parameters** *other* (*DFA|NFA*) – the automaton used as a filter of the output
- **Return type**: SFT

**outputS**(*s*)

Output label coming out of the state i

- **Parameters** *s* (*int*) – index state
- **Return type**: set

**productInput**(*other*)

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

- **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.

- **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

---

6.2. Class SFT (Standard Form Transducers)
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 Nomal Form Transducer

Return type NFT

toOutNFA ()

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

Returns the NFA

Return type NFA

toSFT ()

Pacifying rule

Return type SFT

trim ()

Remove states that do not lead to a final state, or, inclusively, that can’t be reached from the initial state. Only useful states remain.
Attention: in place transformation

\texttt{union}\,(other)

Union of the two transducers

- Parameters \texttt{other}\,(SFT) – the other operand
- Return type \texttt{SFT}

6.3 Functions
CHAPTER
SEVEN

MODULE: FINITE LANGUAGES (FL)

Finite languages and related automata manipulation
Finite languages manipulation

7.1 Class FL (Finite Language)

class fl.FL(wordsList=None, Sigma=None)
   Bases: object
   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

### 7.2 Class DFCA (Deterministic Finite Cover Automata)

```python
class fl.DFCA
    Bases: fa.DFA

Deterministic Cover Automata class
```

```python
length
    size of the longest word :rtype: int
    Type return
```

### 7.3 Class AFA (Acyclic Finite Automata)

```python
class fl.AFA
    Bases: object

Base class for Acyclic Finite Automata
```

```python
addState(_)
    Return type int

directRank()
    Compute rank function
    Returns ranf map
    Return type dict
```

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

---

**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

---

### 7.4 Class ADFA (Acyclic Deterministic Finite Automata)

**class fl.ADFA**

Bases: *fa.DFA*, *fl.AFA*

Acyclic Deterministic Finite Automata class

![Diagram of ADFA hierarchy](image)

---

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.deep.

**Return type** **ADFA**

forceToDFA()

Conversion to DFA

**Return type** **DFA**

forceToDFCA()

Conversion to DFCA

**Return type** **DFA**

level()

Computes the level for each state

**Returns** levels of states

**Return type** **dict**

New in version 0.9.8.

minDFCA()

Generates a minimal deterministic cover automata from a DFA
Return type **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(sl, s2)
Tests if two states of a ADFA are equivalent

Parameters

- **sl** (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.

### 7.5 Class ANFA (Acyclic Non-deterministic Finite Automata)

**class** *fl.ANFA*
**Bases:** *fa.NFA, fl.AFA*

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

7.6 Class RndWGen (Random Word Generator)

class fl.RndWGen(aut)
   Bases: object
   Word random generator class
   New in version 1.2.
   Parameters aut (ADFA) – automata recognizing the language

next()
   Next word
   Returns a new random word

7.7 Functions

fl.sigmaInitialSegment(Sigma, l, exact=False)
   Generates the ADFA recognizing Sigma*i for i<=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

fl.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

fl.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*

**f1. 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?

:returns: the generated trie automaton

**Return type** *ADFA*

**f1. 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?

:returns: the generated trie automaton

**Return type** *ADFA*

**f1. 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*

**f1. stringToADFA** *(s)*

Converts a canonical string representation of a ADFA to a ADFA:

- :param list s: the string in its canonical order

:returns: the ADFA

**Return type** *ADFA*

**See also:**

MODULE: GRAPHS (GRAPH CREATION AND MANIPULATION)

Graph support
Basic Graph object support and manipulation

```python
class graphs.Graph
    Bases: common.Drawable

    Graph base class

    Variables
    • Vertices (list) – Vertices’ names
    • Edges (set) – set of pairs (always sorted)
```

```python
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

dotFormat (size)
    Some dot representation
    Parameters size (str) – size parameter for dotviz
    Return type 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

class graphs.DiGraph

Bases: graphs.Graph

Directed graph base class

addEdge \((v_1, v_2)\)

Adds an edge

Parameters

• **v1** (int) – vertex 1 index
• **v2** (int) – vertex 2 index

static **dotDrawEdge** \((st_1, st_2, sep='\n')\)

Draw a transition in Dot Format

Parameters

• **st1** (str) – starting state
• **st2** (str) – ending state
• **sep** (str) – separator

Return type str

dotDrawVertex \((sti, sep='\n')\)

Draw a Vertex in Dot Format

Parameters

• **sti** (int) – index of the state
• **sep** (str) – separator

Return type str

dotFormat \((size='20, 20', direction='LR', sep='\n', strict=False, maxLblSz=10)\)

A dot representation

Parameters

• **direction** (str) – direction of drawing
• **size** (str) – size of image
• **sep** (str) – line separator
• **maxLblSz** – max size of labels before getting removed
• **strict** – use limitations of label sizes

**Returns** the dot representation

**Return type** str

New in version 0.9.6.
Changed in version 0.9.8.

inverse()
Inverse of a digraph

class graphs.DiGraphVm
Bases: graphs.DiGraph

Directed graph with marked vertices

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

markVertex(v)
Mark vertex v

**Parameters** v (int) – vertex
Module: Context Free Grammars Manipulation (CFG)

Context Free Grammars Manipulation.
Basic context-free grammars manipulation for building uniform random generators

9.1 Class CFGrammar (Context Free Grammar)

class cfg.CFGrammar(gram)
    Bases: object
    Class for context-free grammars
    Variables
    • Rules – grammar rules
    • Terminals – terminals symbols
    • Nonterminals – nonterminals symbols
    • Start(str) – start symbol
    • ntr – dictionary of rules for each nonterminal

Initialization

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

NULLABLE()
    Determines which nonterminals X -* []

makenonterminals()
    Extracts C{nonterminals} from grammar rules.

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

9.2 Class CNF

class cfg.CNF(gram, mark='A@')
    Bases: cfg.CFGrammar
    No useless nonterminals or epsilon 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.
Chomsky()
Transform to CNF

elim_unitary()
Elimination of unitary rules

9.3 Class cfgGenerator

class cfg.cfgGenerator(cfgr, size)
Bases: object

CFG uniform generator

generate()
Generates a new random object generated from the start symbol

Returns object

Return type string

9.4 Class reStringRGGenerator (Reg Exp Generator)

class cfg.reStringRGGenerator(Sigma=None, size=10, cfgr=None, epsilon=None, empty=None, ident='Ti')
Bases: cfg.cfgGenerator

Uniform random Generator for reStrings
Uniform random generator for regular expressions. Used without arguments generates an uncollapsible re over \{a,b\} with size 10. For generate an arbitary re over an alphabet of 10 symbols of size 100: reStringRGenerator(smallAlphabet(10),100,ReGrammar[“g_regular_base”])

Parameters

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

Note: the grammar can have already this symbols

9.5 Functions

cfg.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

`cfg.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`
CHAPTER
TEN

MODULE: RANDOM DFA GENERATOR (RNDFA)

Random DFA generation
ICDFA Random generation binding
Changed in version 0.9.4: Interface python to the C code
Changed in version 0.9.6: Working with incomplete automata
Changed in version 0.9.8: distinct classes for complete and incomplete ICDFA

10.1 Class ICDFArgen (Generator container)

class rndfa.ICDFArgen
Bases: object

Generic ICDFA random generator class

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

See also:

next ()
Get the next generated DFA

Returns a random generated ICDFA

Return type DFA

10.2 Class ICDFArnd (Complete ICDFA random generator)

class rndfa.ICDFArnd(n, k, seed=0)
Bases: rndfa.ICDFArgen

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

Variables

• n (int) – number of states
• \( k (\text{int}) \) – size of the alphabet
• \( \text{seed} (\text{int}) \) – seed for the random generator (if 0 uses time as seed)

**Attention:** At the moment, if two generators are ‘simultaneous’ used, and seed where provided, the repeatable aspect of the generated sequences is *not* warranted. To achieve that behaviour use one generator at a time.

Changed in version 1.3.4: seed added to the random generator

### 10.3 Class ICDFA\_Rnd\_Incomplete (Incomplete ICDFA generator)

```python
class rndfa.ICDFArndIncomplete(n, k, bias=None, seed=0)
    Bases: rndfa.ICDFArGen
```

Incomplete ICDFA random generator class

**Variables**

• \( n (\text{int}) \) – number of states
• \( k (\text{int}) \) – size of alphabet
• \( \text{bias} (\text{float}) \) – how often must the gost sink state appear (default None)
• \( \text{seed} (\text{int}) \) – 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
CHAPTER ELEVEN

MODULE: RANDOM ADFA GENERATOR (RNDADFA)

Random ADFA generation
ADFA Random generation binding
New in version 1.2.1.

11.1 Class ADFArnd (ADFA random generator)

class rndadfa.ADFArnd(n, k=2, s=1)
Sets a random generator for Adfas by sources. By default, s=1 to be initially connected

Variables
• n (int) – number of states
• k (int) – size of the alphabet
• s (int) – number of sources

Note: For ICDFA s=1

alpha(n, s, k=2)
Number of labeled acyclic initially connected DFA by states and by sources

Parameters
• k (int) – alphabet size
• n (int) – number of states
• s (int) – number of 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 sources

  **Return type** int

  **Note:** uses gamma instead of beta or rndAdfa

**beta** \((n, s, k=2)\)
Number of valid configurations of transitions

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

  **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

**next ()**
Generates a random adfa

  **Returns** an dfa if number of sources is 1; otherwise self.transitions has the transitions of an adfa with s sources

  **Return type** DFA

**rndAdfa** \((n, s)\)
Recursively generates a initially connected adfa

  **Parameters**
• \( n (\text{int}) \) – number of states
• \( s (\text{int}) \) – 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

\textbf{rndNumberSecondSources} (n, s)
Uniformaly random generates the number of secondary sources

Parameters
• \( n (\text{int}) \) – number of states
• \( s (\text{int}) \) – number of sources

Return type \( \text{int} \)

\textbf{rndTransitionsFromSources} (n, s, u)
Generates the transitions from the sources, ensuring that all secondary sources are connected

Parameters
• \( n (\text{int}) \) – number of states
• \( s (\text{int}) \) – number of sources
• \( u (\text{int}) \) – number of secondary sources
CHAPTER
TWELVE

MODULE: COMBO OPERATIONS (COMBOPERATIONS)

Several combined operations for DFAs

Combined operations

\texttt{comboperations.starConcat(fa1, fa2, strict=False)}

Star of concatenation of two languages: \((L_1.L_2)^*\)

\textbf{Parameters}

- \texttt{fa1 (DFA)} – first automaton
- \texttt{fa2 (DFA)} – second automaton
- \texttt{strict (bool)} – should the alphabets be necessary equal?

\textbf{Return type} \texttt{DFA}

\textbf{See also:}

\texttt{comboperations.concatWStar(fa1, fa2, strict=False)}

Concatenation combined with star: \((L_1.L_2^*)\)

\textbf{Parameters}

- \texttt{fa1 (DFA)} – first automaton
- \texttt{fa2 (DFA)} – second automaton
- \texttt{strict (bool)} – should the alphabets be necessary equal?

\textbf{Return type} \texttt{DFA}

\textbf{See also:}

\texttt{comboperations.starWConcat(fa1, fa2, strict=False)}

Star combined with concatenation: \((L_1^*.L_2)\)

\textbf{Parameters}

- \texttt{fa1 (DFA)} – first automaton
- \texttt{fa2 (DFA)} – second automaton
- \texttt{strict (bool)} – should the alphabets be necessary equal?

\textbf{Return type} \texttt{DFA}
See also:

`comboperations.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:

`comboperations.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:

`comboperations.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`

`comboperations.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:


`comboperations.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:

Code theory module
New in version 1.0.

13.1 Class CodeProperty

```python
class codes.CodeProperty(name, alph):
    Bases: object
    See also:

    Variables Sigma – the alphabet
```

13.2 Class TrajProp

```python
class codes.TrajProp(aut, Sigma):
    Bases: codes.IATProp
    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
• \texttt{traj} (\texttt{NFA}) – trajectory language
• \texttt{Sigma} (\texttt{set}) – alphabet

\textbf{Return type} \texttt{SFT}

13.3 Class \texttt{IPTProp}

class \texttt{codes.IPTProp} (\texttt{aut}, \texttt{name}=\texttt{None})
Bases: \texttt{codes.CodeProperty} 

Input Preserving Transducer Property

\begin{verbatim}
codes.CodeProperty -> codes.IPTProp
\end{verbatim}

\textbf{Variables}

• \texttt{Aut} (\texttt{SFT}) – the transducer defining the property
• \texttt{Sigma} (\texttt{set}) – alphabet

\textbf{Constructor}
:param \texttt{SFT aut}: Input preserving transducer

\textbf{addToCode} (\texttt{aut}, \texttt{N}=\texttt{2000})

Returns an NFA and a list \texttt{W} of up to \texttt{N} words of length \texttt{ell}, such that the NFA accepts \texttt{L(aut)} union \texttt{W}, which is an error-detecting language. \texttt{ell} is computed from \texttt{aut}

\textbf{Parameters}

• \texttt{aut} (\texttt{NFA}) – the automaton
• \texttt{N} (\texttt{int}) – the number of words to construct
• \texttt{n} (\texttt{int}) – number of tries when needing a new word

\textbf{Returns} an automaton and a list of strings

\textbf{Return type} \texttt{tuple}

\textbf{makeCode} (\texttt{N}, \texttt{ell}, \texttt{s}=\texttt{2000}, \texttt{ov_free}=\texttt{False})

Returns an NFA and a list \texttt{W} of up to \texttt{N} words of length \texttt{ell}, such that the NFA accepts \texttt{W}, which is an error-detecting language. The alphabet to use is \{0,1,...,s-1\}. where \texttt{s} \leq 10.

\textbf{Parameters}

• \texttt{N} (\texttt{int}) – the number of words to construct
• \texttt{ell} (\texttt{int}) – the codeword length
• \texttt{s} (\texttt{int}) – the alphabet size (must be \leq 10)
• \texttt{n} (\texttt{int}) – number of tries when needing a new word

\textbf{Returns} an automaton and a list of strings

\textbf{Return type} \texttt{tuple}
makeCodeO \( (N, \ell, s, n=2000, \text{end}=\text{None}, \text{ov\_free}=\text{False}) \)

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 \( \{0,1,\ldots,s-1\} \). 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 \( \leq 10 \))
- \( n \) (**int**) – number of tries when needing a new word
- \( \text{end} \) (**Word**) – a Word or None that should much the end of code words
- \( \text{ov\_free} \) (**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 \( (\text{aut}, U=\text{None}) \)

Tests if the language is maximal w.r.t. the property

**Parameters**
- \( \text{aut} \) (**NFA**) – the automaton
- \( U \) (**NFA**) – Universe of permitted words (Sigma\(^*\) as default)

**Return type** bool

notMaxStatW \( (\text{aut}, \ell, n=2000, \text{ov\_free}=\text{False}) \)

Returns a word of length \( \ell \) to add into \( \text{aut} \) or None; simpler version of function nonMaxStatFEpsW

**Parameters**
- \( \text{aut} \) (**NFA**) – the automaton
- \( \ell \) (**int**) – the length of the words in \( \text{aut} \)
- \( n \) (**int**) – number of words to try

**Returns** a string or None

**Return type** str

notMaximalW \( (\text{aut}, U=\text{None}) \)

Tests if the language is maximal w.r.t. the property

**Parameters**
- \( \text{aut} \) (**DFA\mid NFA**) – the automaton
- \( U \) (**DFA\mid NFA**) – Universe of permitted words (Sigma\(^*\) as default)

**Return type** bool

**Raises** PropertyNotSatisfied – if not satisfied

notSatisfiesW \( (\text{aut}) \)

Return a witness of non-satisfaction of the property by the automaton language

**Parameters** \( \text{aut} \) (**DFA\mid 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`

### 13.4 Class IATProp

```python
class codes.IATProp(aut, name=None)
    Bases: codes.IPTProp
    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

### 13.5 Class PrefixProp

```python
class codes.PrefixProp(t)
    Bases: codes.TrajProp, codes.FixedProp
    Prefix Property
```

`satisfiesPrefixP(aut)`
Satisfaction of property by the automaton language: faster than satisfiesP
Parameters `aut` *(DFA/NFA)* – the automaton

Return type `bool`

13.6 Class **ErrDetectProp**

codes.ErrDetectProp

**alias of codes.IPTProp**

13.7 Class **ErrCorrectProp**

class codes.ErrCorrectProp (t)

**Bases: codes.IPTProp**

Error Correcting Property

notMaximalW (aut, U=None)

Tests if the language is maximal w.r.t. the property

Parameters

- `aut` *(DFA/NFA)* – the automaton
- `U` *(DFA/NFA)* – Universe of permitted words (Sigma^* as default)

Return type `bool`

notSatisfiesW (aut)

Satisfaction of the code property by the automaton language

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

Return type `tuple`

satisfiesP (aut)

Satisfaction of the property by the automaton language

See also:


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

Return type `bool`
13.8 Functions

codes.buildTrajPropS(\textit{regex, sigma})
Builds a TrajProp from a string regexp

Parameters

• \textit{regex (str)} – the regular expression

• \textit{sigma (set)} – alphabet

Return type \textit{TrajProp}

codes.buildIATPropF(\textit{fname})
Builds a IATProp from a FAdo SFT file

Parameters \textit{fname (str)} – file name

Return type \textit{IATProp}

codes.buildIPTPropF(\textit{fname})
Builds a IPTProp from a FAdo SFT file

Parameters \textit{fname (str)} – file name

Return type \textit{IPTProp}

codes.buildIATPropS(\textit{s})
Builds a IATProp from a FAdo SFT string

Parameters \textit{s (str)} – string containing SFT

Return type \textit{IATProp}

codes.buildIPTPropS(\textit{s})
Builds a IPTProp from a FAdo SFT string

Parameters \textit{s (str)} – file name

Return type \textit{IPTProp}

codes.buildErrorDetectPropF(\textit{fname})
Builds an Error Detecting Property

Parameters \textit{fname (str)} – file name

Return type \textit{ErrDetectProp}

codes.buildErrorCorrectPropF(\textit{fname})
Builds an Error Correcting Property

Parameters \textit{fname (str)} – file name

Return type \textit{ErrCorrectProp}

codes.buildErrorDetectPropS(\textit{s})
Builds an Error Detecting Property from string

Parameters \textit{s (str)} – transducer string

Return type \textit{ErrDetectProp}

codes.buildErrorCorrectPropS(\textit{s})
Builds an Error Correcting Property from string

Parameters \textit{s (str)} – transducer string
Return type  \textit{ErrCorrectProp}
\begin{verbatim}
codes.buildPrefixProperty(alphabet)
    Builds a Prefix Code Property
    \textbf{Parameters} \textit{alphabet} \textit{(set)} – alphabet
    \textbf{Return type} \textit{PrefixProp}
\end{verbatim}
\begin{verbatim}
codes.buildTrajPropS(regex, sigma)
    Builds a TrajProp from a string regexp
    \textbf{Parameters}
    \begin{itemize}
    \item \textit{regex} \textit{(str)} – the regular expression
    \item \textit{sigma} \textit{(set)} – alphabet
    \end{itemize}
    \textbf{Return type} \textit{TrajProp}
\end{verbatim}
\begin{verbatim}
codes.editDistanceW(auto)
    Compute the edit distance of a given regular language accepted by the NFA via Input-altering transducer.
    \textbf{Attention:} language should have at least two words
    \textbf{See also:}
    \textbf{Parameters} \textit{auto} \textit{(NFA)} – language recogniser
    \textbf{Returns} The edit distance of the given regular language plus a witness pair
    \textbf{Return type} \textit{tuple}
\end{verbatim}
\begin{verbatim}
codes.exponentialDensityP(aut)
    Checks if language density is exponential
    Using breadth first search (BFS)
    \textbf{Attention:} aut should not have Epsilon transitions
    \textbf{Parameters} \textit{aut} \textit{(NFA)} – the representation of the language
    \textbf{Return type} \textit{bool}
\end{verbatim}
\begin{verbatim}
codes.createInputAlteringSIDTrans(n, sigmaSet)
    Create an input-altering SID transducer based
    \textbf{Parameters}
    \begin{itemize}
    \item \textit{n} \textit{(int)} – max number of errors
    \item \textit{sigmaSet} \textit{(set)} – alphabet
    \end{itemize}
    \textbf{Returns} a transducer representing the SID channel
    \textbf{Return type} \textit{SFT}
\end{verbatim}
CHAPTER
FOURTEEN

MODULE: GRAIL COMPATIBILITY (GRAIL)

GRAIL support.
GRAIL formats support. This is an auxiliary module that should be imported by fa.py
New in version 0.9.4.

14.1 Class ParserGrail

class grail.ParserGrail (no_table=1, table='.tableGrail')
   Bases: yappy_parser.Yappy
   A parser form GRAIL standard automata descriptions

14.2 Class Grail

class grail.Grail
   Bases: object
   A class for Grail execution
   Changed in version 0.9.8: tries to initialise execPath from fadorc
   do (cmd, *args)
      Execute Grail command
      Parameters
      • cmd (string) – name of the command
      • args – arguments
      Raises
      • GrailCommandError – if the syntax is not correct an exception is raised
• **FAdoGeneralError** – if Grail fails to execute something

**setExecPath** (*path*)

Sets the path to the Grail executables

**Parameters**

`path` (*str*) – the path to Grail executables

## 14.3 Functions

**grail.exportToGrail** (*fileName, fa*)

Saves a finite automaton definition to a file using Grail format

**Parameters**

- `fileName` (*str*) – file name
- `fa` (*FA*) – the FA

**grail.FAToGrail** ((*f, fa*)

Saves a finite automaton definition to an open file using Grail format

**Parameters**

- `f` (*file*) – opended file
- `fa` (*FA*) – the FA

**grail.importFromGrailFile** (*fileName*)

Imports a finite automaton from a file in GRAIL format

The type of the object returned depends on the transition definition read as well as the number of initial states declared

**Parameters**

- `fileName` (*str*) – file name

**Returns**

the automata read

**Return type** *FA*

**grail.FAFromGrail** (*buffer*)

Imports a finite automaton from a buffer in GRAIL format

The type of the object returned depends on the transition definition read as well as the number of initial states declared

**Parameters**

- `buffer` (*str*) – buffer file

**Returns**

the automata read

**Return type** *FA*
Set Specification Transducer support

New in version 1.4.

15.1 Class PSP

class sst.PSP
    Bases: object

    Relation pair of set specifications
CHAPTER
SIXTEEN

SMALL TUTORIAL
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:

```python
>>> 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.

```python
>>> m3 = DFA()
>>> m3.setSigma(['0','1'])
>>> m3.addState('s1')
>>> m3.addState('s2')
```
It is now possible, for instance, to see the structure of the automaton or to test if a word is accepted by it.

```python
>>> 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:

```python
>>> 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:

```text
@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

```python
>>> 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:

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

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

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

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

```python
>>> 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}}))
```

State names can be renamed in-place using:

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

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

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

```python
>>> u.witness()
'@epsilon'
```

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 epsilon-transitions. But it is possible to test if a NFA has epsilon-transitions and convert between a NFA with epsilon-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 (@epsilon) or the concatenation or the union (+) or the Kleene star (*) of a regular expression. Examples of regular expressions are a+b, (a+b)*, and (@epsilon+ a)(ba+ab+@emptyset).

The class regexp is the base class for regular expressions and is used to represent an alphabet symbol. The classes epsilon 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:
But we can convert a string to a regexp class or subclass, using the method str2regexp.

```python
>>> 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) . a 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.

```python
>>> 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).

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

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.

```python
>>> 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 epsilon transitions is obtained (nfaThompson). Epsilon free NFAs can be obtained by the Glushkov method (Position automata) (nfaPosition,) the partial derivatives method (nfaPD) or by the follow method (nfaFollow). The two last methods usually allows to obtain smaller NFAs.

```python
>>> r.nfaThompson()
NFA(['', '', '', '', '0', '1', '2', '3', '8', '9'], ['a', 'b'], ['8'], ['9'], ['''(@epsilon + 0) (0* (@epsilon + 0)) (1 + @(epsilon + 0)) (0* 1)) (1 (0* @epsilon + 0)) (0* 1)) ((1 + ((@epsilon + 0) (0* 1)) (1 (0* 1)))* (1 + ((@epsilon + 0) (0* 1)) (1 (0* 1)))* (1 + (0 ((1 (0* 1))* 0)))* (0 ((1 (0* 1)))* (1 + (1 (0* ((@epsilon + 0))))))]
```

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

```python
>>> r.nfaThompson()
NFA(['', '', '', '', '0', '1', '2', '3', '8', '9'], ['a', 'b'], ['8'], ['9'], ['''(@epsilon + 0) (0* (@epsilon + 0)) (1 + @(epsilon + 0)) (0* 1)) (1 (0* @epsilon + 0)) (0* 1)) ((1 + ((@epsilon + 0) (0* 1)) (1 (0* 1)))* (1 + ((@epsilon + 0) (0* 1)) (1 (0* 1)))* (1 + (0 ((1 (0* 1))* 0)))* (0 ((1 (0* 1)))* (1 + (1 (0* ((@epsilon + 0))))))]
```

```python
>>> r.nfaThompson()
NFA(['', '', '', '', '0', '1', '2', '3', '8', '9'], ['a', 'b'], ['8'], ['9'], ['''(@epsilon + 0) (0* (@epsilon + 0)) (1 + @(epsilon + 0)) (0* 1)) (1 (0* @epsilon + 0)) (0* 1)) ((1 + ((@epsilon + 0) (0* 1)) (1 (0* 1)))* (1 + ((@epsilon + 0) (0* 1)) (1 (0* 1)))* (1 + (0 ((1 (0* 1))* 0)))* (0 ((1 (0* 1)))* (1 + (1 (0* ((@epsilon + 0))))))]
```
>>> r.nfaPosition()
NFA(('Initial', ('a', 1), ('b', 2), ('a', 3)), ('a', 'b'), ('Initial', [('Initial', 'a', 1), ('a', 3)], ('Initial', ('a', 1), ('b', 2), ('a', 3)), ('Initial', ('a', 1), ('a', 2)), ('Initial', ('a', 1), ('b', 2), ('a', 3)), ('Initial', ('a', 1), ('a', 2)), ('Initial', ('a', 1), ('a', 2)), ('Initial', ('a', 1), ('b', 2)), ('Initial', ('a', 1), ('a', 2)), ('Initial', ('a', 1), ('a', 2)), ('Initial', ('a', 1), ('b', 2))]

>>> r.nfaPD()
NFA(('a (a + (b a))*', 'a (a + (b a))*'), ('a', 'b'), ('(a + (b a))*', ('a + (b a))*'), ('(a + (b a))*', 'b', (concat(regex(b), regex(a)))}, (star(disj(regex(a), concat(regex(b), regex(a)))), 'a'), star(disj(regex(a), concat(regex(b), regex(a)))), (concat(regex(a), star(disj(regex(a), concat(regex(b), regex(a))))), 'a', star(disj(regex(a), concat(regex(b), regex(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
CHAPTER EIGHTEEN

INDICES AND TABLES

- genindex
- modindex
- search
PYTHON MODULE INDEX

C
    cfg, 107
    codes, 121
    comboperations, 117
    common, 47

f
    fa, 5
    fio, 49
    fl, 93

g
    grail, 129
    graphs, 103

r
    reex, 51
    rndadfa, 113
    rndfa, 111

s
    sst, 131

t
    transducers, 85
concatWStar() (fa.GFA method), 42
deterministicP () (fa.DFA static method), 20
deterministicP () (fa.NFA method), 20
detSet() (fa.NFA method), 33
DFA (class in fa), 16
dfaAuPoint() (reex.regexp method), 52
dfaBrzozowski() (reex.regexp method), 52
DFAtoADFA() (in module fl), 101
dfaYMG() (reex.regexp method), 52
DFCA (class in fl), 95
dFS() (fa.GFA method), 41
dfs_visit() (fa.GFA method), 42
diff() (fl.FL method), 93
diGraph (class in graphs), 104
diGraphVm (class in graphs), 105
directRank() (fl.FLA method), 95
disj() (class in reex), 65
disj() (fa.DFA method), 6
disjunction() (fa.DFA method), 6
disjWstar() (in module combooperations), 118
diss() (fl.ADFABFA method), 97
dissMin() (fl.ADFABFA method), 97
dist() (fa.DFA method), 20
distDerivative() (reex.specialConstant method), 56
distMin() (fa.DFA method), 20
distR() (fa.DFA method), 20
distRMin() (fa.DFA method), 20
distTS() (fa.DFA method), 20
do() (grail.Grail method), 129
dotDrawEdge() (graphs.DiGraph static method), 104
dotDrawState() (fa.FA method), 7
dotDrawState() (fa.SemiDFA method), 12
dotDrawTransition() (fa.FA method), 7
dotDrawTransition() (fa.OFA static method), 13
dotDrawTransition() (fa.SemiDFA static method), 12
dotDrawVertex() (graphs.DiGraph method), 104
dotFormat() (fa.FA method), 7
dotFormat() (fa.NFA method), 33
dotFormat() (fa.SemiDFA method), 12
dotFormat() (graphs.DiGraph method), 104
dotFormat() (graphs.Graph method), 103
dump() (fa.OFA method), 13
dup() (fa.DFA method), 20
dup() (fa.GFA method), 42
dup() (fa.NFA method), 33
dup() (fa.OFA method), 13
dup() (fl.ADFABFA method), 97
dup() (transducers.SFT method), 87
ediGraph (class in graphs), 104
distMin() (fa.DFA method), 20
distR() (fa.DFA method), 20
distRMin() (fa.DFA method), 20
distTS() (fa.DFA method), 20
do() (grail.Grail method), 129
dotDrawEdge() (graphs.DiGraph static method), 104
dotDrawState() (fa.FA method), 7
dotDrawState() (fa.SemiDFA method), 12
dotDrawTransition() (fa.FA method), 7
dotDrawTransition() (fa.OFA static method), 13
dotDrawTransition() (fa.SemiDFA static method), 12
dotDrawVertex() (graphs.DiGraph method), 104
dotFormat() (fa.FA method), 7
dotFormat() (fa.NFA method), 33
dotFormat() (fa.SemiDFA method), 12
dotFormat() (graphs.DiGraph method), 104
dotFormat() (graphs.Graph method), 103
dump() (fa.OFA method), 13
dup() (fa.DFA method), 20
dup() (fa.GFA method), 42
dup() (fa.NFA method), 33
dup() (fa.OFA method), 13
dup() (fl.ADFABFA method), 97
dup() (transducers.SFT method), 87
ediGraph (class in graphs), 104
distMin() (fa.DFA method), 20
distR() (fa.DFA method), 20
distRMin() (fa.DFA method), 20
distTS() (fa.DFA method), 20
do() (grail.Grail method), 129
dotDrawEdge() (graphs.DiGraph static method), 104
dotDrawState() (fa.FA method), 7
dotDrawState() (fa.SemiDFA method), 12
dotDrawTransition() (fa.FA method), 7
dotDrawTransition() (fa.OFA static method), 13
dotDrawTransition() (fa.SemiDFA static method), 12
dotDrawVertex() (graphs.DiGraph method), 104
dotFormat() (fa.FA method), 7
dotFormat() (fa.NFA method), 33
dotFormat() (fa.SemiDFA method), 12
dotFormat() (graphs.DiGraph method), 104
dotFormat() (graphs.Graph method), 103
dump() (fa.OFA method), 13
dup() (fa.DFA method), 20
dup() (fa.GFA method), 42
dup() (fa.NFA method), 33
dup() (fa.OFA method), 13
dup() (fl.ADFABFA method), 97
dup() (transducers.SFT method), 87
ediGraph (class in graphs), 104
distMin() (fa.DFA method), 20
distR() (fa.DFA method), 20
distRMin() (fa.DFA method), 20
distTS() (fa.DFA method), 20
do() (grail.Grail method), 129
dotDrawEdge() (graphs.DiGraph static method), 104
dotDrawState() (fa.FA method), 7
dotDrawState() (fa.SemiDFA method), 12
dotDrawTransition() (fa.FA method), 7
dotDrawTransition() (fa.OFA static method), 13
dotDrawTransition() (fa.SemiDFA static method), 12
dotDrawVertex() (graphs.DiGraph method), 104
dotFormat() (fa.FA method), 7
dotFormat() (fa.NFA method), 33
dotFormat() (fa.SemiDFA method), 12
dotFormat() (graphs.DiGraph method), 104
dotFormat() (graphs.Graph method), 103
dump() (fa.OFA method), 13
dup() (fa.DFA method), 20
dup() (fa.GFA method), 42
dup() (fa.NFA method), 33
dup() (fa.OFA method), 13
dup() (fl.ADFABFA method), 97
dup() (transducers.SFT method), 87
ediGraph (class in graphs), 104
distMin() (fa.DFA method), 20
distR() (fa.DFA method), 20
distRMin() (fa.DFA method), 20
distTS() (fa.DFA method), 20
do() (grail.Grail method), 129
dotDrawEdge() (graphs.DiGraph static method), 104
dotDrawState() (fa.FA method), 7
dotDrawState() (fa.SemiDFA method), 12
dotDrawTransition() (fa.FA method), 7
dotDrawTransition() (fa.OFA static method), 13
dotDrawTransition() (fa.SemiDFA static method), 12
dotDrawVertex() (graphs.DiGraph method), 104
dotFormat() (fa.FA method), 7
dotFormat() (fa.NFA method), 33
dotFormat() (fa.SemiDFA method), 12
dotFormat() (graphs.DiGraph method), 104
dotFormat() (graphs.Graph method), 103
dump() (fa.OFA method), 13
dup() (fa.DFA method), 20
dup() (fa.GFA method), 42
dup() (fa.NFA method), 33
dup() (fa.OFA method), 13
dup() (fl.ADFABFA method), 97
dup() (transducers.SFT method), 87
ediGraph (class in graphs), 104
distMin() (fa.DFA method), 20
distR() (fa.DFA method), 20
distRMin() (fa.DFA method), 20
distTS() (fa.DFA method), 20
do() (grail.Grail method), 129
dotDrawEdge() (graphs.DiGraph static method), 104
dotDrawState() (fa.FA method), 7
dotDrawState() (fa.SemiDFA method), 12
dotDrawTransition() (fa.FA method), 7
dotDrawTransition() (fa.OFA static method), 13
dotDrawTransition() (fa.SemiDFA static method), 12
dotDrawVertex() (graphs.DiGraph method), 104
dotFormat() (fa.FA method), 7
dotFormat() (fa.NFA method), 33
dotFormat() (fa.SemiDFA method), 12
dotFormat() (graphs.DiGraph method), 104
dotFormat() (graphs.Graph method), 103
dump() (fa.OFA method), 13
dup() (fa.DFA method), 20
dup() (fa.GFA method), 42
dup() (fa.NFA method), 33
dup() (fa.OFA method), 13
dup() (fl.ADFABFA method), 97
dup() (transducers.SFT method), 87

E
editDistanceW() (in module codes), 127
elim_unitary() (cfg.CNF method), 108
elimEpsilon() (fa.NFA method), 33
Index
followLists() (reex.sdisj method), 80
followLists() (reex.specialConstant method), 56
followListsD() (reex.atom method), 72
followListsD() (reex.shuffle method), 70
followListsD() (reex.specialConstant method), 56
followListsStar() (reex.atom method), 72
followListsStar() (reex.specialConstant method), 56
followListsStar() (reex.sdisj method), 80
followListsStar() (reex.specialConstant static method), 56
forceToDFA() (fl.ADFArnd method), 97
forceToDFCA() (fl.ADFArnd method), 97
functionalP() (transducers.SFT method), 88

G

gamma() (rndadfa.ADFArnd method), 114
generate() (cfg.cfgGenerator method), 108
genRandomTrie() (in module fl), 101
genRndTrieBalanced() (in module fl), 100
genRndTriePrefix() (in module fl), 101
genRndTrieUnbalanced() (in module fl), 100
getLeaves() (fl.AFA method), 96
GFA (class in fa), 41
Grail (class in grail), 129
grail (module), 129
Graph (class in graphs), 103
graphs (module), 103
gRules() (in module cfg), 108

H

half() (fa.NFA method), 35
hasStateIndexP() (fa.FA method), 8
hasTransitionP() (fa.NFA method), 35
hasTrapStateP() (fa.DFA method), 22
head() (reex.sconcat method), 78
head_rev() (reex.sconcat method), 78
HKeqP() (fa.DFA method), 17
HKeqP() (fa.NFA method), 31
homogeneousFinalityP() (fa.NFA method), 35
homogenousP() (fa.NFAr method), 40
hyperMinimal() (fa.DFA method), 22

I

IATProp (class in codes), 124
ICDFArgen (class in rndfa), 111
ICDFArnd (class in rndfa), 111
ICDFArndIncomplete (class in rndfa), 112
images() (fa.FA method), 8
importFromGrailFile() (in module grail), 130
inDegree() (fa.DFA method), 22
indexList() (fa.FA method), 8
infix() (fa.DFA method), 23
inIntersection() (transducers.SFT method), 88
inIntersectionSlow() (transducers.SFT method), 88
Initial (fa.FA attribute), 5
initialComp() (fa.DFA method), 23
initialComp() (fa.GFA method), 42
initialComp() (fa.NFA method), 35
initialComp() (fa.OFA method), 14
initialP() (fa.DFA method), 23
initialP() (fa.FA method), 8
initialSet() (fa.DFA method), 23
initialSet() (fa.FA method), 9
inputS() (fa.FA method), 9
intersection() (fl.FL method), 94
interWStar() (in module comboperations), 119
inverse() (graphs.DiGraph method), 105
inverse() (transducers.SFT method), 88
IPTProp (class in codes), 122

J

joinStates() (fa.DFA method), 23

L

last() (reex.atom method), 72
last() (reex.concat method), 64
last() (reex.connective method), 61
last() (reex.disj method), 65
last() (reex.option method), 68
last() (reex.power method), 67
last() (reex.regexp method), 53
last() (reex.sconnective method), 77
last() (reex.sdisj method), 80
last() (reex.snot method), 82
last() (reex.specialConstant method), 57
last() (reex.star method), 62
length (fl.DFCA attribute), 95
1EquivNFA() (fa.NFA method), 35
level() (fl.ADFArnd method), 97
linearForm() (reex.atom method), 73
linearForm() (reex.concat method), 64
linearForm() (reex.conj method), 70
linearForm() (reex.connective method), 61
linearForm() (reex.disj method), 65
linearForm() (reex.option method), 68
linearForm() (reex.power method), 67
linearForm() (reex.regexp method), 53
linearForm() (reex.sconnective method), 78
linearForm() (reex.snot method), 70
linearForm() (reex.sconnective method), 77
linearForm() (reex.sdisj method), 80
linearForm() (reex.snot method), 82
linearForm() (reex.specialConstant method), 57
option (class in reex), 68
ordered() (fl.AFA method), 96
orderedStrConnComponents() (fa.DFA method),
outIntersection() (transducers.SFT method), 89
outIntersectionDerived() (transducers.SFT method), 89
outputS() (transducers.SFT method), 89

P
pairGraph() (fa.DFA method), 26
ParseReg (class in reex), 76
ParserGrail (class in grail), 129
partialDerivatives() (reex.atom method), 73
partialDerivatives() (reex.epsilon static method), 58
partialDerivatives() (reex.sconcat method), 78
partialDerivatives() (reex.sconj method), 81
partialDerivatives() (reex.sdisj method), 80
partialDerivatives() (reex.sigmaP method), 60
partialDerivatives() (reex.sigmaS method), 61
partialDerivatives() (reex.snot method), 82
partialDerivatives() (reex.sstar method), 79
partialDerivativesC() (reex.atom method), 74
partialDerivativesC() (reex.emptyset method),
59
partialDerivativesC() (reex.epsilon method), 58
partialDerivativesC() (reex.sconcat method),
78
partialDerivativesC() (reex.sconj method), 81
partialDerivativesC() (reex.sdisj method), 80
partialDerivativesC() (reex.sigmaP static method), 60
partialDerivativesC() (reex.sigmaS method), 61
partialDerivativesC() (reex.snot method), 82
partialDerivativesC() (reex.specialConstant method),
57
partialDerivativesC() (reex.sstar method), 79
FD() (reex.atom method), 71
plus() (fa.FA method), 9
position (class in reex), 75
possibleToReverse() (fa.DFA method), 26
possibleToReverse() (fl.ADAFA method), 98
power (class in reex), 66
pref() (fa.DFA method), 26
PrefixProp (class in codes), 124
print_data() (fa.DFA method), 26
product() (fa.DFA method), 26
product() (fa.NFA method), 36
productInput() (transducers.SFT method), 89
productInputSlow() (transducers.SFT method), 89
productSlow() (fa.DFA method), 26
PSP (class in sst), 131

R
re_stateElimination() (fa.OFA method), 15
re_stateElimination_nn() (fa.OFA method), 15
readFromFile() (in module fio), 49
reCG() (fa.OFA method), 14
reCG_nn() (fa.OFA method), 15
reduced() (reex.atom method), 74
reDynamicCycleHeuristic() (fa.OFA method), 15
reex (module), 51
regexp (class in reex), 51
regexp() (fa.DFA method), 26
regexpSE() (fa.OFA method), 15
RegularExpression (class in reex), 51
renameState() (fa.FA method), 9
renameStates() (fa.FA method), 9
renameStatesFromPosition() (fa.NFA method),
36
reorder() (fa.DFA method), 27
reorder() (fa.GFA method), 42
reorder() (fa.NFA method), 37
rEquivNFA() (fa.OFA method), 36
reStaticCycleHeuristic() (fa.OFA method), 15
reStringRGenerator (class in cfg), 108
reversal() (fa.FA method), 10
reversal() (fa.NFA method), 37
reversal() (reex.atom method), 74
reversal() (reex.concat method), 64
reversal() (reex.disj method), 66
reversal() (reex.option method), 69
reversal() (reex.power method), 67
reversal() (reex.specialConstant method), 57
reversal() (reex.star method), 63
reversal() (transducers.SFT method), 89
reverseTransitions() (fa.DFA method), 27
reverseTransitions() (fa.NFA method), 37
reversibleP() (fa.DFA method), 27
rnndadfa (module), 113
rnndAdfa() (rnndadfa.ADFArnd method), 114
rndfa (module), 111
rnndNumberSecondSources() (rnndadfa.ADFArnd method), 115
rnndTransitionsFromSources() (rnndadfa.ADFArnd method), 115
RndWGen (class in fl), 100
rpn() (reex.atom method), 74
rpn() (reex.concat method), 64
rpn() (reex.conj method), 70
rpn() (reex.connective method), 62
rpn() (reex.disj method), 66
rpn() (reex.emptyset method), 59
rpn() (reex.epsilon method), 58
rpn() (reex.option method), 69
rpn() (reex.power method), 67
rpn() (reex.regex method), 55
rpn() (reex.sconnective method), 77
rpn() (reex.shuffle method), 70
rpn() (reex.snot method), 82
rpn() (reex.specialConstant method), 57
rpn() (reex.star method), 63
rpn2regexp() (in module reex), 83
runOnNFA() (transducers.SFT method), 89
runOnWord() (transducers.SFT method), 90
S
same_nullability() (fa.FA method), 10
satisfiesP() (codes.ErrCorrectProp method), 125
satisfiesP() (codes.IPTProp method), 124
satisfiesPrefixP() (codes.PrefixProp method), 124
saveToFile() (in module fio), 50
saveToString() (in module reex), 44
sconcat (in module reex), 77
sconj (in module reex), 80
sconnective (in module reex), 76
sdisj (in module reex), 79
SemiDFA (class in fa), 11
setDeadState() (fa.FA method), 96
setExecPath() (grail.Grail method), 128
setFinal() (fa.FA method), 10
setInitial() (fa.FA method), 10
setInitial() (fa.NFA method), 37
setInitial() (transducers.SFT method), 90
setOfSymbols() (reex.atom method), 74
setOfSymbols() (reex.connective method), 62
setOfSymbols() (reex.option method), 69
setOfSymbols() (reex.position method), 76
setOfSymbols() (reex.power method), 67
setOfSymbols() (reex.regex method), 55
setOfSymbols() (reex.sconnective method), 77
setOfSymbols() (reex.snot method), 82
setOfSymbols() (reex.specialConstant method), 57
setOfSymbols() (reex.star method), 63
setOutput() (transducers.Transducer method), 85
setSigma() (fa.FA method), 10
setSigma() (fl.FL method), 94
setSigma() (reex.regex method), 55
SFT (in class transducers), 85
shuffle (in module reex), 70
shuffle() (fa.DFA method), 27
shuffle() (fa.NFA method), 37
Sigma (fa.FA attribute), 5
Sigma (fa.SemiDFA attribute), 12
sigmaInitialSegment() (in module fl), 100
sigmaP (in module reex), 59
sigmaS (in module reex), 60
simDiff() (fa.DFA method), 27
smallAlphabet() (in module cfg), 109
sMonoid() (fa.DFA method), 27
snf() (reex.atom method), 74
snf() (reex.epsilon method), 58
snot (class in reex), 81
sop() (fa.DFA method), 27
specialConstant (class in reex), 56
SPRegExp() (fa.OFA method), 12
square() (transducers.SFT method), 90
square_fv() (transducers.SFT method), 90
SSemiGroup (class in fa), 43
sSemiGroup() (fa.DFA method), 27
sst (module), 131
sstar (class in reex), 78
star (class in reex), 62
star() (fa.DFA method), 28
star() (fa.NFA method), 37
star() (transducers.SFT method), 90
starConcat() (in module comboperations), 117
starDisj() (in module comboperations), 118
starHeight() (reex.atom static method), 74
starHeight() (reex.connective method), 62
starHeight() (reex.option method), 69
starHeight() (reex.power method), 67
starHeight() (reex.regex method), 55
starHeight() (reex.sconnective method), 77
starHeight() (reex.snot method), 82
starHeight() (reex.star method), 63
starI() (fa.DFA method), 28
starInter() (in module comboperations), 118
starInter0() (in module comboperations), 118
starWConcat() (in module comboperations), 117
stateAlphabet() (fa.FA method), 10
stateChildren() (fa.DFA method), 28
stateChildren() (fa.GFA method), 43
stateChildren() (fa.NFA method), 37
stateChildren() (fa.OFA method), 15
stateIndex() (fa.FA method), 10
stateName() (fa.FA method), 11
statePairEquiv() (fl.AFA method), 98
States (fa.FA attribute), 5
States (fa.SemiDFA attribute), 11
str2regexp() (in module reex), 83
str2sre() (in module reex), 83
stringLength() (reex.atom method), 75
stringToADFA() (in module fl), 101
stringToDFA() (in module fa), 45
stronglyConnectedComponents() (fa.DFA method), 28
stronglyConnectedComponents() (fa.NFA method), 37
subword() (fa.DFA method), 28
subword() (fa.NFA method), 37
succintTransitions() (fa.DFA method), 28
succintTransitions() (fa.FA method), 11
succintTransitions() (fa.GFA method), 43
succintTransitions() (fa.NFA method), 37
succintTransitions() (fa.OFA method), 16
succintTransitions() (transducers.Transducer method), 85
suff() (fa.DFA method), 28
suffixClosedP() (fl.FL method), 94
support() (reex.atom method), 75
support() (reex.conj method), 70
support() (reex.sconcat method), 78
support() (reex.sconj method), 81
support() (reex.shuffle method), 70
support() (reex.sigmaP method), 60
support() (reex.sigmaS method), 61
support() (reex.snot method), 82
support() (reex.specialConstant method), 57
support(last) (reex.atom method), 75
support(last) (reex.shuffle method), 70
support(last) (reex.specialConstant method), 57
syncPower() (fa.DFA method), 28
syncWords() (fa.DFA method), 29
syntacticLength() (reex.atom static method), 75
syntacticLength() (reex.connective method), 77
syntacticLength() (reex.snot method), 82

T

tail() (reex.sconcat method), 78
tail_rev() (reex.sconcat method), 78
tailForm() (reex.atom method), 75
tailForm() (reex.concat method), 65
tailForm() (reex.disj method), 66
tailForm() (reex.regexp method), 55
tailForm() (reex.specialConstant method), 57
tailForm() (reex.star method), 64
toADFA() (fa.DFA method), 29
toANFA() (fl.ADFA method), 98
toDFA() (fa.DFA method), 29
toDFA() (fa.NFA method), 38
toDFA() (fl.FL method), 94
toDFA() (reex.regexp method), 55
toGFA() (fa.DFA method), 29
toGFA() (fa.GFA method), 43
toGFA() (fa.NFA method), 38
toGFA() (fa.OFA method), 16
toInNFA() (transducers.SFT method), 90
toNFA() (fa.DFA method), 29
toNFA() (fa.NFA method), 38
toNFA() (fa.NFAr method), 40
toNFA() (fl.ADFA method), 98
toNFA() (fl.FL method), 94
toNFA() (reex.regexp method), 55
toNFAr() (fa.NFA method), 38
toNFT() (transducers.SFT method), 90
toOutNFA() (transducers.SFT method), 90
topoSort() (fa.OFA method), 16
toSFT() (transducers.SFT method), 90
TrajProp (class in codes), 121
trajToTransducer() (codes.TrajProp static method), 121
Transducer (class in transducers), 85
transducers (module), 85
transitions() (fa.DFA method), 29
transitionsA() (fa.DFA method), 29
treeLength() (reex.atom static method), 75
treeLength() (reex.connective method), 62
treeLength() (reex.option method), 69
treeLength() (reex.power method), 67
treeLength() (reex.regexp method), 55
treeLength() (reex.sconnective method), 77
treeLength() (reex.snot method), 82
treeLength() (reex.star method), 64
trieFA() (fl.FL method), 94
trim() (fa.OFA method), 16
trim() (fl.ADFA method), 98
trim() (transducers.SFT method), 90
trimP() (fa.OFA method), 16

U

union() (fa.FA method), 11
union() (fl.FL method), 94
union() (transducers.SFT method), 91
unionSigma() (reex.regexp method), 55
uniqueRepr() (fa.DFA method), 29
uniqueRepr() (fa.GFA method), 43
uniqueRepr() (fa.NFA method), 38
uniqueRepr() (fa.OFA method), 16
universalP() (fa.DFA method), 29
unlinkSoleIncoming() (fa.NFAr method), 40
unlinkSoleOutgoing() (fa.NFAr method), 41
unmark() (fa.DFA method), 30
unmark() (reex.concat method), 65
unmark() (reex.disj method), 66
unmark() (reex.specialConstant method), 57
unmark() (reex.star method), 64
unmarked() (reex.atom method), 75
unmarked() (reex.position method), 76
unmarked() (reex.specialConstant method), 57
usefulStates() (fa.DFA method), 30
usefulStates() (fa.GFA method), 43
usefulStates() (fa.NFA method), 38
usefulStates() (fa.OFA method), 16

V

vDescription() (fa.DFA static method), 30
vDescription() (fa.NFA static method), 38
vertexIndex() (graphs.Graph method), 103

W

weight() (fa.GFA method), 43
weightWithCycles() (fa.GFA method), 43
witness() (fa.DFA method), 30
witness() (fa.NFA method), 38
witnessDiff() (fa.DFA method), 30
Word (class in common), 47
wordDerivative() (reex.regexp method), 55
wordDerivative() (reex.specialConstant method), 57
wordGenerator() (fl.ADAFA method), 99
WordI() (fa.SSemiGroup method), 44
wordImage() (fa.NFA method), 38
WordPS() (fa.SSemiGroup method), 44
words() (fa.FA method), 44