**CONTENTS**

1 About this package 3
1.1 Install ...................................................... 3
1.2 Licence ..................................................... 3
1.3 Prerequisites ............................................... 3
1.4 Useful links ............................................... 4
1.5 Release history ........................................... 4

2 Finitely generated subgroups of free groups 5
2.1 The class FinitelyGeneratedSubgroup ....................... 5
2.2 Ancillary functions about words ............................ 20
2.3 Ancillary functions about automata ........................ 29
2.4 The folding algorithm ...................................... 39
2.5 Ancillary functions about bases ............................ 42
2.6 About morphisms between free groups: connecting with the train_track package .......... 45
2.7 Ancillary functions about free factors ..................... 48

3 Partial injections 55
3.1 The class PartialInjection .................................. 55
3.2 Ancillary functions about partial injections ............... 58

4 References, Indices and tables 61

Bibliography 63

Python Module Index 65

Index 67
This is the reference manual for Pascal Weil’s stallings_graphs Research Code extension to the Sage mathematical software system. Sage is free open source math software that supports research and teaching in algebra, geometry, number theory, cryptography, and related areas.
stallings_graphs Research Code implements tools to experiment with finitely generated subgroups of infinite groups in Sage, via a set of new Python classes. Many of the modules correspond to research code written for published articles (random generation, decision for various properties, etc). It is meant to be reused and reusable (full documentation including doctests).

Comments are welcome.

1.1 Install

To install this module, you do:

```
sage -pip install stallings_graphs
```

or:

```
sage -pip install http://www.labri.fr/perso/weil/software/stallings_graphs-0.2.tar.gz
```

To use this module, you need to import it:

```
from stallings_graphs import *
```

This reference manual contains many examples that illustrate the usage of stallings_graphs. The examples are all tested with each release of stallings_graphs, and should produce exactly the same output as in this manual, except for line breaks.

1.2 Licence

This work is licensed under a Creative Commons Attribution-Share Alike 3.0 License.

1.3 Prerequisites

Installing stallings_graph requires a working SageMath installation. Depending on the usage (especially to produce LaTeX, pdf, png outputs), it might be necessary to install the optional package dot2tex:

```
sage -i dot2tex
```

as well as the external packages ImageMagick, Graphviz.

On OSX, after installing Homebrew, one may do:
sudo brew install graphviz imagemagick

On Debian or Ubuntu, one may do:

```bash
sudo apt install graphviz imagemagick
```

Note that `graphviz` must be installed *before* `dot2tex`.

The package `stallings_graph` also uses Sébastien Labbé’s `slabbe` package and Thierry Coulbois’s `train_track` package. Both are automatically installed when `stallings_graphs` is installed.

### 1.4 Useful links

- **On PyPI**: https://pypi.org/project/stallings-graphs/
- **Professional homepage link**: https://www.labri.fr/perso/weil/software/
- **The slabbe package**: https://pypi.org/project/slabbe/
- **The train_track package**: https://pypi.org/project/train-track/

### 1.5 Release history

**Version 0.2**, released September 2020 (Pascal Weil)

- the package now works with Python 3
- some bug fixes
- addition of new methods on finitely generated subgroups, including the decision of free factors and the computation of the lattice of algebraic extensions of a subgroup
- definition and usage of morphisms between free groups, by means of a connection to Thierry Coulbois’s `train_track` package

**Version 0.1**, released March 2019 (Pascal Weil)
2.1 The class FinitelyGeneratedSubgroup

The class FinitelyGeneratedSubgroup is meant to represent finitely generated subgroups of free groups. The representation of a FinitelyGeneratedSubgroup is a tuple of partial injections on a set of the form \([0..(n-1)]\) (one for each generator of the ambient free group), which represent the Stallings graph of the subgroup, with base vertex 0.

Methods implemented in this file:

- definition of a FinitelyGeneratedSubgroup from a list of generators (Words)
- definition of a FinitelyGeneratedSubgroup from a DiGraph (by folding and pruning)
- random instance
- ambient_group_rank, to compute the rank of the ambient free group
- stallings_graph_size
- rank, to compute the rank of the subgroup
- stallings_graph, to compute the Stallings graph of the subgroup
- show_Stallings_graph, to visualize the Stallings graph
- is_valid, to check the necessary properties of connectedness and trimness
- eq, to check whether two objects represent the same finitely generated subgroup
- basis
- contains_element, to check whether the subgroup contains a given word
- contains_subgroup, to check whether the subgroup contains a given subgroup
- intersection
- has_index, to compute the index of the subgroup
- conjugated_by, to compute the conjugate of a subgroup by a given word
- is_conjugated_to, to check whether two subgroups are conjugated and, optionally, compute a conjugating word
- is_malnormal, to check whether the subgroup is malnormal and, optionally, compute a witness of its non-malnormality
- is_free_factor_of_ambient, to check whether the subgroup is a free factor of the ambient group and, optionally, to compute a complement
• *is_free_factor_of_*

EXAMPIES:

```python
sage: from stallings_graphs import FinitelyGeneratedSubgroup
sage: gens = ['ab', 'ba']
sage: G = FinitelyGeneratedSubgroup.from_generators(gens, alphabet_type='abc')
sage: G
A subgroup of the free group of rank 2, whose Stallings graph has 3 vertices
```

```python
sage: gens = [[1,2,5,-1,-2,2,1],[-1,-2,2,3],[1,2,3]]
sage: G = FinitelyGeneratedSubgroup.from_generators(gens)
sage: G
A subgroup of the free group of rank 5, whose Stallings graph has 3 vertices
```

```python
sage: from stallings_graphs import FinitelyGeneratedSubgroup
sage: from stallings_graphs.about_words import random_reduced_word
sage: from stallings_graphs.about_automata import bouquet
sage: L = [random_reduced_word(100,2) for _ in range(10)]
sage: G = bouquet(L)
sage: H = FinitelyGeneratedSubgroup.from_digraph(G)
sage: H
# random
A subgroup of the free group of rank 2, whose Stallings graph has 965 vertices
```

```python
sage: H = FinitelyGeneratedSubgroup.random_instance(15)
sage: H
A subgroup of the free group of rank 2, whose Stallings graph has 15 vertices
```

AUTHORS:

• Pascal WEIL (2018-04-26): initial version

CNRS, Univ. Bordeaux, LaBRI <pascal.weil@cnrs.fr>

```python
class stallings_graphs.finitely_generated_subgroup.FinitelyGeneratedSubgroup(partial_injections)
Bases: sage.structure.sage_object.SageObject
```

Define the class *FinitelyGeneratedSubgroup*, which represents subgroups of free groups.

The representation of a finitely generated subgroup is by means of the partial injections (on a set of the form
\([0..n-1]\), one per generator of the ambient free group) which describes its Stallings graph, with base vertex 0.

The Stallings graph of a subgroup is a uniquely defined finite directed graph, whose edges are labeled by positive
letters, rooted in a designated vertex, subject to three conditions: it must be connected, folded (no two edges
with the same label share the same initial (resp. terminal) vertex), and every vertex must have valency 2 (in the
underlying non-directed graph), except possibly for the root (also known as base vertex). That is: a subgroup is
represented by a tuple of partial injections on \([0..n-1]\), up to a relabeling of the elements of \([0..n-1]\) fixing
the base vertex (namely 0).

A *FinitelyGeneratedSubgroup* can be created from:

• a list of objects of the class *PartialInjection*, all of the same size;

or

• a list of *Words* on a symmetrical alphabet: either *aUz / A:Z* (upper case is the inverse of lower case),
so-called *alphabet_type='abc'*, or \([-r..-1,1..r]\), so-called *alphabet_type='123'*. 

or

• a labeled *DiGraph* with vertex set \([0..(n-1)]\) and edge labels in a positive alphabet (*aUz* if
*alphabet_type='abc'*, or \([1..r]\) if *alphabet_type='123'*). The *DiGraph* is considered to be rooted at
vertex 0.
or

• a random instance.

**SW_is_free_factor_of**(other, complement=True, alphabet_type='123')

Return whether self is a free factor of other and, if complement is set to True, gives either a statement about it not being a free factor, or if it is, a basis of a complement of self in other (in numerical or in alphabetic form depending on alphabet_type).

other is expected to be a FinitelyGeneratedSubgroup

**INPUT:**

• other – FinitelyGeneratedSubgroup

• complement – boolean

• alphabet_type – a string which can be either 'abc' or '123'

**OUTPUT:**

• a boolean if complement is False, and a pair of a boolean and either a string or a list of objects of type Word otherwise

**EXAMPLES:**

```python
sage: from stallings_graphs import FinitelyGeneratedSubgroup
sage: LH = [[2,-3,1,3,2,3,-2,-1,2,-3,-1], [3,1,1,3,-1,1,3,2,3,1,-1]]
sage: LK = [[2,-3], [1,1], [1,3,-2,1,2,-3,-1], [3,2], [3,1,-3,1], [1,3,2,1], [1,3,3,-1], [1,3,1,-1]]
sage: H = FinitelyGeneratedSubgroup.from_generators(LH, alphabet_type='123')
sage: K = FinitelyGeneratedSubgroup.from_generators(LK, alphabet_type='123')
sage: H.SW_is_free_factor_of(K, complement=True)
(True, [word: 3,2,3,1,2,-1, word: 32, word: 3,1,3,-1, word: 11])
sage: H.SW_is_free_factor_of(K, complement=False)
True

sage: LH = [[-3,1,-2,-1,-1,-3,2,2,3], [-3,-1,-1,3,1,1,-3,1,3,3], [-3,1,3,-1,-1,3,1,3,-1,-1,1]]
sage: LK = [[1,1,2,-1,3], [1,1,3,-1,1], [-3,1,1,3], [-3,2,3], [1,3,3]]
sage: H = FinitelyGeneratedSubgroup.from_generators(LH, alphabet_type='123')
sage: K = FinitelyGeneratedSubgroup.from_generators(LK, alphabet_type='123')
sage: H.SW_is_free_factor_of(K, complement=True)
(False, 'the 1st argument is not a free factor of the second')
sage: H.SW_is_free_factor_of(K, complement=False)
False

sage: LH = [[1,1,2,-1,3], [1,1,3,-1,1], [-3,1,1,3], [-3,2,3], [1,3,3]]
sage: LK = [[1,1,2,-1,3], [1,1,3,-1,1], [-3,1,1,3], [-3,2,3], [1,3,3]]
sage: H = FinitelyGeneratedSubgroup.from_generators(LH, alphabet_type='123')
sage: K = FinitelyGeneratedSubgroup.from_generators(LK, alphabet_type='123')
sage: H.SW_is_free_factor_of(K, complement=True)
(False, '1st argument not contained in 2nd')

sage: H.SW_is_free_factor_of(K, complement=False)
False

sage: H = FinitelyGeneratedSubgroup.from_generators(['bba', 'bAbA'], alphabet_type='abc')
sage: K = FinitelyGeneratedSubgroup.from_generators(['a', 'bb', 'bAbA'], alphabet_type='abc')
sage: H.SW_is_free_factor_of(K, complement=True, alphabet_type='abc')
(True, [word: BB])
```

2.1. The class FinitelyGeneratedSubgroup
sage: H.SW_is_free_factor_of(K, complement = False, alphabet_type = 'abc')
True

sage: H = FinitelyGeneratedSubgroup.from_generators(['a','b'], alphabet_type='abc')
sage: K = FinitelyGeneratedSubgroup.from_generators(['a','b','d'], alphabet_type='abc')
sage: H.SW_is_free_factor_of(K, complement = True, alphabet_type = 'abc')
(True, [word: d])

sage: H.SW_is_free_factor_of(K, complement = False, alphabet_type = 'abc')
True

ALGORITHM:
The algorithm implemented is from [SW2008]. Be aware that the worst-case complexity is polynomial in the size of the two argument subgroups, but exponential in the difference between their ranks.

\textbf{SW\_is\_free\_factor\_of\_ambient}(complement=True, alphabet_type='123')

Return whether self is a free factor of the ambient group and, if complement is set to True, gives either a statement about it not being a free factor, or if it is, a basis of a complement of self (in numerical or in alphabetic form depending on alphabet_type).

INPUT:
- complement – boolean
- alphabet_type – a string which can be either 'abc' or '123'

OUTPUT:
- a boolean if complement is False and a pair of a boolean and either a string of a list of objects of type Word otherwise

EXAMPLES:

sage: from stallings_graphs import FinitelyGeneratedSubgroup
sage: L1 = ['ac', 'bacd', 'ed']
sage: H1 = FinitelyGeneratedSubgroup.from_generators(L1, alphabet_type='abc')
sage: H1.SW_is_free_factor_of_ambient(complement = True, alphabet_type = 'abc')
(True, [word: baE, word: B])

sage: L2 = ['acac', 'bacd', 'ed']
sage: H2 = FinitelyGeneratedSubgroup.from_generators(L2, alphabet_type='abc')
sage: H2.SW_is_free_factor_of_ambient(complement = True, alphabet_type='abc')
(False, 'the 1st argument is not a free factor of the second')

sage: H = FinitelyGeneratedSubgroup.from_generators(['A','d'], alphabet_type='abc')
sage: H.SW_is_free_factor_of_ambient(complement = True, alphabet_type='abc')
(True, [word: b, word: c])

ALGORITHM:
The algorithm implemented is from [SW2008]. Be aware that the worst-case complexity is polynomial in the size of the argument subgroup but exponential in the rank difference between that subgroup and the ambient group.

\textbf{algebraic\_extensions}()

Return a dictionary listing the algebraic extensions of self. The keys are integers without any particular meaning, except key 0 corresponds to H itself. The entries are lists of an algebraic extension, sets of keys corresponding to parents and children of this extension (not a Hasse diagram of the containment relation, but including such a diagram), and a boolean indicating whether the extension is e-algebraic.
For a definition of algebraic and e-algebraic extensions, see [MVW2007].

INPUT:
- `self` – an object of the class `FinitelyGeneratedSubgroup`.

OUTPUT:
- a dictionary whose keys are integers and whose entries are lists of an object of type `FinitelyGeneratedSubgroup`, two sets of keys, and a boolean

EXAMPLES:

```python
sage: from stallings_graphs import FinitelyGeneratedSubgroup
sage: from stallings_graphs.about_free_factors import compute_algebraic_extensions
sage: testgens = ['aba','bab']
sage: testH = FinitelyGeneratedSubgroup.from_generators(testgens,alphabet_type='abc')
sage: testH.algebraic_extensions()
{0: [A subgroup of the free group of rank 2, whose Stallings graph has 5 vertices, set(), {1}, True],
  1: [A subgroup of the free group of rank 2, whose Stallings graph has 1 vertices, {0}, set(), True]}
```

```python
sage: testgens = ['ab','cd']
sage: testH = FinitelyGeneratedSubgroup.from_generators(testgens,alphabet_type='abc')
sage: testH.algebraic_extensions()
{0: [A subgroup of the free group of rank 4, whose Stallings graph has 3 vertices, set(), set(), True]}
```

```python
sage: testgens = ['ABBaBABa','Baba','Ababba','Ababba','ABabAb']
sage: testH = FinitelyGeneratedSubgroup.from_generators(testgens,alphabet_type='abc')
sage: testH.algebraic_extensions()
{0: [A subgroup of the free group of rank 2, whose Stallings graph has 10 vertices, set(), {3, 6, 11}, True],
  3: [A subgroup of the free group of rank 2, whose Stallings graph has 1 vertices, {0, 6, 11}, set(), True],
  6: [A subgroup of the free group of rank 2, whose Stallings graph has 3 vertices, {0, 11}, {3}, True],
  11: [A subgroup of the free group of rank 2, whose Stallings graph has 8 vertices, {0}, {3, 6}, True]}
```

`ambient_group_rank()`

Return the rank of the ambient free group of this `FinitelyGeneratedSubgroup` object.

Exploits the fact that the rank of the ambient free group is the number of partial injections which specify this `FinitelyGeneratedSubgroup`.

INPUT:
- `self` – a `FinitelyGeneratedSubgroup`

OUTPUT:
• an integer

EXAMPLES:

```python
sage: from stallings_graphs import FinitelyGeneratedSubgroup, PartialInjection
sage: L = [PartialInjection([1,2,None,4,5,3]), PartialInjection([0,3,None,None,None])]
sage: H = FinitelyGeneratedSubgroup(L)
sage: H.ambient_group_rank()
2
```

```python
sage: L = []
    sage: H = FinitelyGeneratedSubgroup.from_generators(L)
    sage: H.ambient_group_rank()
0
```

```python
sage: from stallings_graphs import FinitelyGeneratedSubgroup
sage: L = [[3,1,-2,-1,3],[1,2,-1,2,1,2],[1,2,-3,-3,1]]
    sage: H = FinitelyGeneratedSubgroup.from_generators(L)
    sage: H.basis(alphabet_type = '123')
[word: -3,1,2,-1,-3, word: -2,-1,2,1,-2,-1, word: -1,3,3,-2,-1]
```

```python
sage: H.basis()  
[word: CabAC, word: BAbBA, word: AccBA]
```

```python
sage: H = FinitelyGeneratedSubgroup([])
    sage: H.basis()
[]
```

```python
sage: H = FinitelyGeneratedSubgroup.from_generators(['A'],alphabet_type = 'abc')
    sage: H.basis()
[word: a]
```

```python
conjugated_by(w, alphabet_type='123')
```

Return the conjugate of this subgroup by the given word.

w is expected to be a Word, on a numerical or letter alphabet, depending on the value of alphabet_type. The conjugate of a subgroup $H$ by a word $w$ is the subgroup $w^{-1}Hw$.

INPUT:

• self – a FinitelyGeneratedSubgroup

• w – a Word

• alphabet_type – a string which can be either 'abc' or '123'

OUTPUT:

• a FinitelyGeneratedSubgroup
EXAMPLES

```python
sage: from stallings_graphs import FinitelyGeneratedSubgroup
sage: L = ['ab', 'ba', 'aBaa']
sage: H = FinitelyGeneratedSubgroup.from_generators(L, alphabet_type = 'abc')
```

A subgroup of the free group of rank 2, whose Stallings graph has 4 vertices

```python
sage: H
```

```python
sage: w1 = Word('bA')
```

```python
sage: K1 = H.conjugated_by(w1, alphabet_type='abc')
```

A subgroup of the free group of rank 2, whose Stallings graph has 4 vertices

```python
sage: w2 = Word('bAA')
```

```python
sage: K2 = H.conjugated_by(w2, alphabet_type='abc')
```

A subgroup of the free group of rank 2, whose Stallings graph has 5 vertices

```python
sage: w = Word('abba')
```

```python
sage: K3 = H.conjugated_by(w, alphabet_type='abc')
```

```python
sage: K3 == H
```

```
True
```

```python
contains_element(w, alphabet_type='123')
```

Return whether the subgroup contains the word \( w \).

\( w \) is expected to be a Word on a numerical alphabet (alphabet_type = '123') or on a letter alphabet (alphabet_type = 'abc').

INPUT:

- self – a FinitelyGeneratedSubgroup
- w – a Word
- alphabet_type – a string which is either 'abc' or '123'

OUTPUT:

- a boolean

EXAMPLES

```python
sage: w = Word([1,-2,-2])
```

```python
sage: H.contains_element(w)
```

```
False
```

```python
sage: w = Word('abba')
```

```python
sage: H.contains_element(w, alphabet_type = 'abc')
```

```
True
```

```python
sage: w = Word()  
```

```python
sage: H.contains_element(w)
```

```
True
```

2.1. The class FinitelyGeneratedSubgroup
contains_subgroup(other)

Return whether the subgroup contains another subgroup.

other is expected to be an object of class FinitelyGeneratedSubgroup.

INPUT:
- self – a FinitelyGeneratedSubgroup
- other – a FinitelyGeneratedSubgroup

OUTPUT:
- a boolean

EXAMPLES:

```
sage: from stallings_graphs import FinitelyGeneratedSubgroup
sage: L = ['ab', 'ba', 'aBaa']
sage: H = FinitelyGeneratedSubgroup.from_generators(L, alphabet_type = 'abc')
sage: M = ['ab', 'ba']
sage: K = FinitelyGeneratedSubgroup.from_generators(M, alphabet_type = 'abc')
sage: H.contains_subgroup(K)
True
```

```
sage: LL = ['abba', 'bAbA']
sage: K = FinitelyGeneratedSubgroup.from_generators(LL, alphabet_type = 'abc')
sage: H.contains_subgroup(K)
True
```

```
sage: H = FinitelyGeneratedSubgroup([])
sage: H.contains_subgroup(K)
False
```

```
sage: K.contains_subgroup(H)
True
```

static from_digraph(G)

Return the FinitelyGeneratedSubgroup specified by a DiGraph.

G is expected to be a DiGraph with edge labels in [1..r], whose vertices are a set of non-negative integers including 0 (no verification is made). In particular, the empty graph with no vertices is not admissible. The Stallings graph of the finitely generated subgroup produced is obtained by choosing 0 as the base vertex, folding and pruning G.

INPUT:
- G – DiGraph

OUTPUT:
- an object of the class FinitelyGeneratedSubgroup

EXAMPLES:

```
sage: from stallings_graphs import FinitelyGeneratedSubgroup
sage: L = ['abaBa', 'BaBaB', 'cacBac', 'AbAbb']
sage: from stallings_graphs.about_automata import bouquet
sage: G = bouquet(L, alphabet_type='abc')
```

Warning: No exception will be raised if the input is not of the expected type.

**static from_generators***(generators, alphabet_type='123')**

Return the **FinitelyGeneratedSubgroup** specified by a set of generators.

generators is expected to be a list of valid Word objects, either numerical or alphabetical, in accordance with the value of alphabet_type. The **FinitelyGeneratedSubgroup** produced represents the subgroup generated by these words. It is computed by operating a free group reduction on the elements of generators, computing the bouquet of these words and then creating the **FinitelyGeneratedSubgroup** specified by the bouquet.

**INPUT:**

• generators – a tuple of Word objects

**OUTPUT:**

• an object of the class **FinitelyGeneratedSubgroup**

**EXAMPLES:**

```python
def main():
    from stallings_graphs import FinitelyGeneratedSubgroup
    gens = ['ab', 'ba']
    H = FinitelyGeneratedSubgroup.from_generators(gens, alphabet_type='abc')
    H
    A subgroup of the free group of rank 2, whose Stallings graph has 3 vertices

    gens = [[1, 2, 5, -1, -2, 2, 1], [-1, -2, 2, 3], [1, 2, 3]]
    H = FinitelyGeneratedSubgroup.from_generators(gens)
    H
    A subgroup of the free group of rank 5, whose Stallings graph has 3 vertices
```

2.1. The class **FinitelyGeneratedSubgroup**
sage: L = []
sage: H = FinitelyGeneratedSubgroup.from_generators(L)
sage: H
A subgroup of the free group of rank 0, whose Stallings graph has 1 vertices

sage: L = [[2]]
sage: H = FinitelyGeneratedSubgroup.from_generators(L)
sage: H
A subgroup of the free group of rank 2, whose Stallings graph has 1 vertices

Warning: No exception will be raised if the input is not of the expected type. Also: generators can be an empty list.

has_index()
Return the index of this subgroup if it is finite, +Infinity otherwise.

INPUT:
• self – a FinitelyGeneratedSubgroup

OUTPUT:
• an integer or +Infinity

EXAMPLES:

sage: from stallings_graphs import FinitelyGeneratedSubgroup
sage: L = [[3,1,-2,-1,3],[1,2,-1,-2,1,2],[1,2,-3,-3,1]]
sage: H = FinitelyGeneratedSubgroup.from_generators(L)
sage: H.has_index()
+Infinity

sage: H = FinitelyGeneratedSubgroup([])
sage: H.has_index()
1

sage: HH = FinitelyGeneratedSubgroup.from_generators(['ab', 'ba', 'Abab'], alphabet_type = 'abc')
sage: HH.has_index()
2

intersection(K)
Return the intersection of two subgroups.

Both inputs are expected to be objects of class FinitelyGeneratedSubgroup. We understand both to be subgroups of the rank $r$ free group, where $r$ is the maximum of the ambient group ranks of the input subgroups. The intersection is also understood to be a subgroup of the same rank $r$ free group.

INPUT:
• self – FinitelyGeneratedSubgroup
• other – FinitelyGeneratedSubgroup

OUTPUT:
• FinitelyGeneratedSubgroup

EXAMPLES

Chapter 2. Finitely generated subgroups of free groups
sage: from stallings_graphs import FinitelyGeneratedSubgroup
sage: L = [[2,1,-2,2,1,-2], [2,3,1,-3,3,1,-2]]

sage: H = FinitelyGeneratedSubgroup.from_generators(L)

sage: M = ['ab', 'ba', 'bdaB']

sage: K = FinitelyGeneratedSubgroup.from_generators(M, alphabet_type = 'abc')

sage: H.intersection(K)
A subgroup of the free group of rank 4, whose Stallings graph has 1 vertices

sage: L = ['ab', 'aaBa', 'bbAb']

sage: H = FinitelyGeneratedSubgroup.from_generators(L, alphabet_type = 'abc')

sage: M = ['ab', 'bbbb', 'baba', 'ab']

sage: K = FinitelyGeneratedSubgroup.from_generators(M, alphabet_type = 'abc')

sage: S = H.intersection(K)

sage: S.basis()
[word: baba, word: baBaBB, word: ab, word: AbAA]

is_conjugated_to(other, conjugator=False, alphabet_type='123')
Return whether self and other are conjugated.

If conjugator is set to True, the output will also include a conjugator (None if the two subgroups are not conjugated). A word w is a conjugator of H into K if w⁻¹Hw = K.

INPUT:
  • other – FinitelyGeneratedSubgroup
  • conjugator – boolean
  • alphabet_type – a string which can be either 'abc' or '123'

OUTPUT:
  • a boolean or, if conjugator is True, a tuple consisting of a boolean and a Word or None.

EXAMPLES:

sage: from stallings_graphs import FinitelyGeneratedSubgroup

sage: generators = ['aCA', 'abbaA', 'aCacA', 'abbbCA']

sage: H = FinitelyGeneratedSubgroup.from_generators(generators, alphabet_type='abc')

sage: other_gens = ['ba', 'bbcb', 'bccc', 'bbba']

sage: K = FinitelyGeneratedSubgroup.from_generators(other_gens, alphabet_type='abc')

sage: H.is_conjugated_to(K)
True

sage: b, w = H.is_conjugated_to(K,conjugator=True,alphabet_type = 'abc')

sage: w
word: ac

is_malnormal(alphabet_type='123', witness=False)
Return whether this subgroup is malnormal.

The first argument is assumed to be an object of class FinitelyGeneratedSubgroup. The second argument determines whether words are to be represented numerically or alphabetically. This makes a difference only if witness is set to True. In that case, the output includes witness words s, t such that s belongs to the intersection of H and t⁻¹Ht.

INPUT:
  • self – FinitelyGeneratedSubgroup
  • alphabet_type – a string which is either 'abc' or '123'
  • witness – a boolean

OUTPUT:
• a boolean if witness is set to False; and if witness is set to True, then a tuple of the form (True, None, None) if the subgroup is malnormal, and of the form (False, s, t) if it is not, where s and t are of the class \texttt{Word}.

EXAMPLES

sage: from stallings_graphs import FinitelyGeneratedSubgroup
sage: L = [[2,1,-2,1,2], [2,3,1,3,1,-2]]

sage: H = FinitelyGeneratedSubgroup.from_generators(L)

sage: H.is_malnormal()
False

sage: L = ['ab', 'aaBa', 'bbAb']

sage: H = FinitelyGeneratedSubgroup.from_generators(L, alphabet_type = 'abc')

sage: H.is_malnormal()
False

sage: L = ['baB', 'ababa', 'aababbb']

sage: H = FinitelyGeneratedSubgroup.from_generators(L, alphabet_type = 'abc')

sage: H.is_malnormal()
True

sage: M = ['ab', 'bbbb', 'baba', 'aa']

sage: K = FinitelyGeneratedSubgroup.from_generators(M, alphabet_type = 'abc')

sage: K.is_malnormal()
False

sage: H = FinitelyGeneratedSubgroup.from_generators(['a'], alphabet_type = 'abc')

sage: H.is_malnormal()
True

sage: H = FinitelyGeneratedSubgroup([])

sage: H.is_malnormal()
True

sage: L = ['aba', 'abb', 'a8ababA']

sage: H = FinitelyGeneratedSubgroup.from_generators(L, alphabet_type = 'abc')

sage: H.is_malnormal(alphabet_type = 'abc', witness = True)
(False, word: aba, word: aB)

TODO: The algorithm is quadratic and that is rather inefficient for large instances. One would probably gain significant time if, after verifying non malnormality, one could explore the (non-diagonal) connected components starting with the smaller ones.

\textbf{is_valid(\texttt{verbose=False})}

Return whether this \texttt{FinitelyGeneratedSubgroup} input really defines a subgroup.

If verbose is set to True, indications are given if the input is not valid, on the first reason encountered why it is the case. In order: not all elements of partial_injections are actually partial injections; the graph is not connected; some vertex other than 0 has degree less than 2.

INPUT:

• self – \texttt{FinitelyGeneratedSubgroup}

• verbose – boolean

OUTPUT:

• a boolean if verbose is set to False; a pair of a boolean and a string otherwise

EXAMPLES:
ALGORITHM:
The first verification is whether every element of the input’s constitutive list of partial injections is indeed a valid partial injection. The fact that these partial injections all have the same size was checked when this list was made into a `FinitelyGeneratedSubgroup`. The next steps are to verify whether the graph induced by these partial injections is connected, and that all the vertices except for the base vertex (vertex 0) have degree at least 2.

**Warning:** It is not checked whether the input is of the correct type.

`static random_instance(size, ambient_rank=2, verbose=False)`

Return a randomly chosen `FinitelyGeneratedSubgroup`.

`size` is expected to be at least 1 and `ambient_rank` is expected to be at least 0 (a `ValueError` will be raised otherwise). The `FinitelyGeneratedSubgroup` is picked uniformly at random among those of the given size and with the same ambient free group rank.

If the option `verbose` is set to `True`, also prints the number of attempts in the rejection algorithm.

**INPUT:**
- `size` – integer
- `ambient_rank` – integer, default value 2
- `verbose` – a boolean, default value False

**OUTPUT:**
- an object of the class `FinitelyGeneratedSubgroup` if `verbose = False`, and a tuple of an object of the class `FinitelyGeneratedSubgroup` and an integer otherwise

**EXAMPLES:**

```sage
deffrom stallings_graphs import FinitelyGeneratedSubgroup, PartialInjection
defH = FinitelyGeneratedSubgroup.random_instance(12)
defH
A subgroup of the free group of rank 2, whose Stallings graph has 12 vertices
```

```sage
defH = FinitelyGeneratedSubgroup.random_instance(2, ambient_rank = 0)
defH
A subgroup of the free group of rank 0, whose Stallings graph has 1 vertices
```

```sage
defH,c = FinitelyGeneratedSubgroup.random_instance(12,3,verbose=True)
defH
A subgroup of the free group of rank 3, whose Stallings graph has 12 vertices
```

```sage
c #random
1
```
ALGORITHM:

This uses a rejection algorithm. It consists in drawing uniformly at random a tuple of ambient_rank partial injections, each of size size and testing whether they define a valid FinitelyGeneratedSubgroup. If they do not, the tuple is tossed and another is drawn.

For a justification, see [BNW2008].

rank()

Return the rank of this FinitelyGeneratedSubgroup.

The rank of this FinitelyGeneratedSubgroup is equal to edges - vertices + 1, where vertices and edges refer to the number of vertices and edges of the Stallings graph of the corresponding subgroup. In particular vertices is stallings_graph_size and edges is the sum of the domain sizes of the partial injections.

INPUT:

• self – FinitelyGeneratedSubgroup

OUTPUT:

• an integer

EXAMPLES:

sage: from stallings_graphs import FinitelyGeneratedSubgroup, PartialInjection
sage: L = [PartialInjection([1,2,None,4,5,3]), PartialInjection([0,3,4,None,None,None])]
sage: H = FinitelyGeneratedSubgroup(L)
sage: H.rank()
3

: 
sage: L = [] sage: H = FinitelyGeneratedSubgroup.from_generators(L) sage: H.rank() 0

show_Stallings_graph(alphabet_type='abc', visu_tool='plot')

Show the Stallings graph of this FinitelyGeneratedSubgroup.

Edge labels can be of the form $a_1, ..., a_r$ (alphabet_type='123') or of the form $a, b, c, ..., z$ (alphabet_type='abc'). The visualization tool can be graph.plot (with a color coding for the base vertex) or Sébastien Labbé's TikzPicture method.

INPUT:

• self – a FinitelyGeneratedSubgroup

• alphabet_type – a string which is either 'abc' or '123'

• visu_tool – a string which is either 'plot' or 'tikz'

OUTPUT:

• a visualization of the Stallings graph using graph.plot or using TikzPicture, according to the value of visu_tool. In the 'tikz' case, the output can be saved as a .png, .pdf or .tex file

EXAMPLES

sage: from stallings_graphs import FinitelyGeneratedSubgroup, PartialInjection
sage: L = [PartialInjection([1,2,None,4,5,3]), PartialInjection([0,3,4,None,None,None])]
sage: H = FinitelyGeneratedSubgroup(L)
sage: H.show_Stallings_graph(alphabet_type='abc', visu_tool='plot')
Graphics object consisting of 28 graphics primitives

sage: t = H.show_Stallings_graph(alphabet_type='abc', visu_tool='tikz')
sage: # one can then type t.png, t.tex, t.pdf
**stallings_graph()**

Return the Stallings DiGraph of this FinitelyGeneratedSubgroup.

The Stallings graph of the subgroup of a free group represented by this FinitelyGeneratedSubgroup is an edge-labeled DiGraph. The vertex set is \([0..(n - 1)]\), where \(n\) is the size of the input. The base vertex is 0. If \(r\) is the ambient_group_rank of the input, each of the \(r\) partial injections defining the FinitelyGeneratedSubgroup specifies the edges labeled by that particular letter.

**INPUT:**
- `self` – FinitelyGeneratedSubgroup

**OUTPUT:**
- a DiGraph

**EXAMPLES:**

```python
sage: from stallings_graphs import FinitelyGeneratedSubgroup, PartialInjection
sage: L = [PartialInjection([1,2,None,4,5,3]), PartialInjection([0,3,4,None,None,None])]

sage: H = FinitelyGeneratedSubgroup(L)
sage: G = H.stallings_graph()
sage: G
Looped multi-digraph on 6 vertices
```

```python
sage: L = []
sage: H = FinitelyGeneratedSubgroup.from_generators(L)
sage: H.stallings_graph()
sage: G
Looped multi-digraph on 1 vertex
```

**stallings_graph_size()**

Return the size of this FinitelyGeneratedSubgroup.

The size of the FinitelyGeneratedSubgroup is the number of vertices of the Stallings graph of the subgroup it represents. It is equal to the (common) length of the partial injections defining it.

**INPUT:**
- `self` – FinitelyGeneratedSubgroup

**OUTPUT:**
- an integer

**EXAMPLES:**

```python
sage: from stallings_graphs import FinitelyGeneratedSubgroup, PartialInjection
sage: L = [PartialInjection([1,2,None,4,5,3]), PartialInjection([0,3,4,None,None,None])]

sage: H = FinitelyGeneratedSubgroup(L)
sage: H.stallings_graph_size()
6
```

```python
sage: L = []
sage: H = FinitelyGeneratedSubgroup.from_generators(L)
sage: H.stallings_graph_size()
1
```
2.2 Ancillary functions about words

The methods for the class \texttt{FinitelyGeneratedSubgroup} use a number of ancillary functions. These are the functions which deal with words (actually, objects of class \texttt{Word}) in the context of group theory.

A word is a string of characters from either a numerical or an alphabetical set of letters: \texttt{alphabet_type='123'} or \texttt{'abc'}.

\texttt{alphabet_type='123'}: The positive letters form an interval \([1, r]\). Their inverses (a.k.a. negative letters) are the corresponding negative integers. The symmetrized alphabet is the union of positive and negative letters (zero is NOT a letter). The rank of a word is the maximal absolute value of a letter occurring in the word. When represented in a (say LaTeX) file (.tex, .pdf), the letters are written \(a_i\).

\texttt{alphabet_type='abc'}: positive letters are lower case (at most 26 letters, \(a:z\)) and their inverses are the corresponding upper case letters (\(A:Z\)).

We have functions to:

- translate a word or a list of words from one \texttt{alphabet_type} to the other
- test whether a word of \texttt{alphabet_type '123'} is (freely) reduced or cyclically reduced
- freely reduce a word of \texttt{alphabet_type '123'}
- computes the cyclic reduction of a word of \texttt{alphabet_type '123'}
- produce a random word of \texttt{alphabet_type '123'} of given length on an alphabet of given rank (given a positive integer \(r\)).

**EXAMPLES:**

```python
sage: from stallings_graphs.about_words import group_inverse
sage: w = Word('aBabbaBA')
sage: group_inverse(w, alphabet_type='abc')
word: abABBAbA
```

```python
sage: from stallings_graphs.about_words import free_group_reduction
sage: w = Word([3,1,-2,-2,1,-1,2,5,-3])
sage: free_group_reduction(w)
word: 3,1,5,-3
```

```python
sage: from stallings_graphs.about_words import random_reduced_word
sage: w = random_reduced_word(7,3)  #random
Word([2,-1,3,-1,-1,2,-3])
```

**AUTHOR:**

- Pascal WEIL, CNRS, Univ. Bordeaux, LaBRI <pascal.weil@cnrs.fr> (2018-06-09): initial version.

\texttt{stallings_graphs.about_words.alphabetic_inverse(x)}

Return the inverse of an alphabetic letter.

\(x\) is expected to be a character in \(a:z\) or \(A:Z\). Taking the inverse toggles between upper and lower case letters.

**INPUT:**

- \(x\) – character

**OUTPUT:**

- character

**EXAMPLES:**
sage: \texttt{from stallings_graphs.about_words import alphabetic_inverse}
  sage: alphabetic_inverse('b')
  'B'
  sage: alphabetic_inverse('D')
  'd'

\texttt{stallings\_graphs.about\_words.cyclic\_reduction\_of\_a\_word(}u\texttt{)}

Return the elements of the cyclically reduced decomposition of this word.

\textit{u} is expected to be a \texttt{Word} on a numerical alphabet. The cyclically reduced decomposition of \textit{u} is the pair of \texttt{Words} \textit{(v, w)} such that \textit{v} is cyclically reduced, and \textit{u} = \textit{w}^{-1}\textit{vw}.

**INPUT:**
- \textit{w} – \texttt{Word}
- \texttt{check} – boolean

**OUTPUT:**
- pair of objects of class \texttt{Word}

**EXAMPLES**

sage: \texttt{from stallings_graphs.about_words import cyclic_reduction_of_a_word}
  sage: u = Word([1,-2,-2,-1,1])
  sage: cyclic_reduction_of_a_word(u)
  (word: 1,-2,-2, word: )
  sage: u = Word([1,-2,1,-2,1,2,-1])
  sage: cyclic_reduction_of_a_word(u)
  (word: 1,-2,1, word: 2,-1)
  sage: u = Word([1,-2,1,-1,1])
  sage: cyclic_reduction_of_a_word(u)
  (word: 1,-2,1, word: )
  sage: u = Word([1,2,-2,-1,2,2,1,-1,-2])
  sage: cyclic_reduction_of_a_word(u)
  (word: 2, word: )
  sage: u = Word()
  sage: cyclic_reduction_of_a_word(u)
  (word: , word: )

\texttt{stallings\_graphs.about\_words.free\_group\_reduction(}w, \texttt{check=}False\texttt{)}

Return the reduced word that is equivalent to this word.

\textit{w} is expected to be a \texttt{Word} on a numerical alphabet. The option \texttt{check = True} verifies that this is the case. The reduced word equivalent to a word \textit{w} is obtained from \textit{w} by repeatedly deleting pairs of consecutive letters which are mutually inverse.

**INPUT:**
- \textit{w} – \texttt{Word}
- \texttt{check} – boolean

**OUTPUT:**
- \texttt{Word}

**EXAMPLES**
sage: from stallings_graphs.about_words import free_group_reduction
sage: w = Word([3,1,-2,-2,1,-1,2,5,-3])
sage: free_group_reduction(w)
word: 3,1,5,-3

ALGORITHM:
This method implements the classical algorithm, based on the usage of a pushdown automaton.

stallings_graphs.about_words.group_inverse(w, alphabet_type='123', check=False)
Return the (free group) inverse of a word. 

\(w\) is expected to be a \(\text{Word}\) on a numerical or letter alphabet, depending on the value of \(\text{alphabet_type}\). Its inverse is obtained in reading \(w\) in reverse order and replacing each letter by its inverse. If \(\text{check}\) is set to True, \(\text{is\_valid\_Word}\) is run on \(w\).

INPUT:
• \(w\) – \(\text{Word}\)
• alphabet_type – string, which must be either 'abc' or '123'

OUTPUT:
• Word

EXAMPLES:

sage: from stallings_graphs.about_words import group_inverse
sage: w = Word([3,1,-9,-2,5])
sage: group_inverse(w)
word: -5,2,9,-1,-3
sage: w = Word([-1,1,2,-2])
sage: group_inverse(w)
word: 2,-2,-1,1
sage: w = Word([1])
sage: group_inverse(w)
word: -1
sage: w = Word()
sage: group_inverse(w)
word:

stallings_graphs.about_words.inverse_letter(i)
Return the inverse of this (numerical) letter.

\(i\) is expected to be a non-zero integer.

INPUT:
• \(i\) – integer

OUTPUT:
• integer

EXAMPLES:

sage: from stallings_graphs.about_words import inverse_letter
sage: inverse_letter(3)
-3
```python
sage: inverse_letter(-4)
4
```

**stallings_graphs.about_words.is_cyclically_reduced**(w, check=False)

Return whether this word is cyclically reduced.

*w* is expected to be a *Word* on a numerical alphabet. The option `check` verifies that it is the case. A word is cyclically reduced if it is reduced and its first and last letters are not mutually inverse.

**INPUT:**

- *w* – *Word*
- *check* – boolean

**OUTPUT:**

- boolean

**EXAMPLES**

```python
sage: from stallings_graphs.about_words import is_cyclically_reduced
sage: w = Word([3,1,-2,-2,5,-3])
sage: is_cyclically_reduced(w)
False
```

```python
sage: u = Word([3,1,-2,2,5,-3])
sage: is_cyclically_reduced(u)
True
```

**stallings_graphs.about_words.is_reduced**(w, check=False)

Return whether this word is a reduced.

*w* is expected to be a *Word* on a numerical alphabet. A word *w* is reduced (in the group-theoretic sense) if it does not contain consecutive letters which are mutually inverse. The option `check` verifies whether *w* is a valid Word.

**INPUT:**

- *w* – *Word*
- *check* – boolean

**OUTPUT:**

- boolean

**EXAMPLES**

```python
sage: from stallings_graphs.about_words import is_reduced
sage: w = Word([3,1,-2,-2,5,-3])
sage: is_reduced(w)
True
```

```python
sage: u = Word([3,1,-2,2,5,-3])
sage: is_reduced(u)
False
```

**stallings_graphs.about_words.is_valid_Word**(w, alphabet_type='123')

Return whether a *Word* is valid, in the sense of having a consistent alphabet.

*w* is expected to be a *Word*. It is *valid* if all its letters are non-zero integers if `alphabet_type='123'`; or are in *α:z + A:Z* if `alphabet_type='abc'`.

**INPUT:**

- *w* – *Word*
- *alphabet_type* – ‘123’
• w – Word
  • alphabet_type – string, which must be either 'abc' or '123'

OUTPUT:
  • boolean

EXAMPLES:

```python
sage: from stallings_graphs.about_words import is_valid_Word
sage: w = Word([2, -1, -2, 3, 1, 3])
sage: is_valid_Word(w)
True

sage: is_valid_Word(Word('bABcac'), alphabet_type='abc')
True

sage: is_valid_Word(Word([2, -1, -2, 0, 1, 3]))
False
```

```python
stallings_graphs.about_words.is_valid_list_of_Words(L, alphabet_type='123')
```

Return whether a list of Words is valid, in the sense of is_valid_Word.

$L$ is expected to be a list of objects of class Word. It is valid if all its components satisfy is_valid_Word.

INPUT:
  • L – List of objects of the class Word
  • alphabet_type – string, which must be either 'abc' or '123'

OUTPUT:
  • boolean

EXAMPLES:

```python
sage: from stallings_graphs.about_words import is_valid_list_of_Words
sage: L = [Word([2, -1, -2, 3, 1, 3]), Word([1, 2, -3, 1, -1])]
sage: is_valid_list_of_Words(L)
True

:: sage: L = [Word('bABcac'),'abcBA','baaCB'] sage: is_valid_list_of_Words(L, alphabet_type='abc') True
```

```python
stallings_graphs.about_words.negative_letters(r)
```

Return the set of negative (numerical) letters up to $-r$.

$r$ is expected to be positive.

INPUT:
  • r – integer

OUTPUT:
  • the list of integers from $-1$ to $-r$

EXAMPLES:

```python
sage: from stallings_graphs.about_words import negative_letters
sage: negative_letters(6)
[-1, -2, -3, -4, -5, -6]
```
stallings_graphs.about_words.positive_alphabetic_content(w, check=False)

Return the set of positive letters which occur, or their inverse occurs in \( w \).

\( w \) is expected to be a Word on a numerical alphabet. If \( \text{check} \) is \( \text{True} \), \( \text{is_valid_Word}(w, \text{alphabet_type}='123') \) is run.

INPUT:
- \( w \) – Word
- \( \text{check} \) – Boolean

OUTPUT:
- list of positive integers

EXAMPLES:

```sage
from stallings_graphs.about_words import positive_alphabetic_content
sage: w = Word([3,1,-2,-2,5,-3])
sage: positive_alphabetic_content(w)
{1, 2, 3, 5}
```

```sage
sage: w = Word([])
sage: positive_alphabetic_content(w)
set()
```

stallings_graphs.about_words.positive_letters(r)

Return the list of positive (numerical) letters up to \( r \).

\( r \) is expected to be positive.

INPUT:
- \( r \) – integer

OUTPUT:
- list

EXAMPLES:

```sage
from stallings_graphs.about_words import positive_letters
sage: positive_letters(6)
[1, 2, 3, 4, 5, 6]
```

stallings_graphs.about_words.positive_value(i)

Return the positive value of a (numerical) letter.

\( i \) is expected to be a non-zero integer.

INPUT:
- \( i \) – integer

OUTPUT:
- integer

EXAMPLES:

```sage
from stallings_graphs.about_words import positive_value
sage: positive_value(3)
3
sage: positive_value(-4)
4
```
stallings_graphs.about_words.random_letter(r)
Return a random letter in the symmetric alphabet of this size.

$r$ is expected to be a positive integer. The symmetric alphabet of size $r$ is the set of non-zero integers between $-r$ and $r$. The probability distribution is uniform.

INPUT:
• $r$ – integer

OUTPUT:
• integer

EXAMPLES

```
sage: from stallings_graphs.about_words import random_letter
sage: random_letter(4)  # random
2
```

stallings_graphs.about_words.random_reduced_word(n, r)
Return a random reduced word of length $n$ in the symmetric alphabet of size $r$.

$n$ is expected to be a non-negative integer and $r$ is expected to be a positive integer. A word is reduced if it does not contain consecutive letters which are mutually inverse. The probability distribution is uniform.

INPUT:
• $n$ – integer
• $r$ – integer

OUTPUT:
• Word

EXAMPLES

```
sage: from stallings_graphs.about_words import random_reduced_word
sage: random_reduced_word(4, 5)  # random
Word([2,-1,3,-4,-1])
```

stallings_graphs.about_words.random_word(n, r)
Return a random word of length $n$ in the symmetric alphabet of size $r$.

$n$ is expected to be a non-negative integer and $r$ is expected to be a positive integer. The word produced on the symmetric alphabet of size $r$ is not necessarily reduced. The probability distribution is uniform.

INPUT:
• $n$ – integer
• $r$ – integer

OUTPUT:
• Word

EXAMPLES

```
sage: from stallings_graphs.about_words import random_word
sage: random_word(4, 5)  # random
Word([2,-1,3,-4,-1])
```

stallings_graphs.about_words.rank(w, check=False)
Return the least rank of a free group containing this Word.

Chapter 2. Finitely generated subgroups of free groups
$w$ is expected to be a \texttt{Word} on a numerical alphabet. The least rank of a free group containing $w$ is the max of the positive values of its letters. If check is True, \texttt{is_valid_Word(w,alphabet_type='123')} is run.

**INPUT:**
- $w$ – \texttt{Word}
- check – boolean

**OUTPUT:**
- integer

**EXAMPLES:**

```python
sage: from stallings_graphs.about_words import rank
sage: w = Word([3,1,-2,-2,5,-3])
rank(w)
5
```

```python
sage: w = Word([])
rank(w)
0
```

\texttt{stallings_graphs.about_words.symmetric_alphabet}(r)

Return the full symmetric (numerical) alphabet.

$r$ is expected to be positive.

**INPUT:**
- $r$ – integer

**OUTPUT:**
- the list of integers from 1 to $r$ and from $-1$ to $-r$

**EXAMPLES:**

```python
sage: from stallings_graphs.about_words import symmetric_alphabet
sage: symmetric_alphabet(6)
[1, 2, 3, 4, 5, 6, -1, -2, -3, -4, -5, -6]
```

\texttt{stallings_graphs.about_words.translate_alphabetic_Word_to_numeric}(w)

Return the numerical equivalent of a \texttt{Word} of alphabet \texttt{alphabet_type = 'abc'}.

$w$ is expected to be a \texttt{Word} on alphabet $a:z + A:Z$. The output is a \texttt{Word} on alphabet $\{\pm1, \ldots, \pm26\}$.

**INPUT:**
- $w$ – \texttt{Word}

**OUTPUT:**
- \texttt{Word}

**EXAMPLES:**

```python
sage: from stallings_graphs.about_words import translate_alphabetic_Word_to_numeric
sage: translate_alphabetic_Word_to_numeric(Word('aBBaAc'))
word: 1,-2,-2,1,-1,3
```

```python
sage: translate_alphabetic_Word_to_numeric(Word(''))
word: 
```
sage: translate_alphabetic_Word_to_numeric(Word([]))
word:

stallings_graphs.about_words.translate_character_to_numeric(x)
Return the numeric equivalent of a character in a:z or A:Z.

x is expected to be a character in a:z or A:Z. The numeric equivalent is 1-26 for a:z and the opposite for A:Z.

INPUT:
- x – character

OUTPUT:
- integer

EXAMPLES:

sage: from stallings_graphs.about_words import translate_character_to_numeric
sage: translate_character_to_numeric('b')
2
sage: translate_character_to_numeric('D')
-4

stallings_graphs.about_words.translate_numeric_Word_to_alphabetic(w)
Return the alphabetic equivalent of a numeric word.

w is expected to be a Word on a numerical alphabet \{±1, . . . , ±26\}. The output is a Word on alphabet a:z + A:Z.

INPUT:
- w – Word

OUTPUT:
- Word

EXAMPLES:

sage: from stallings_graphs.about_words import translate_numeric_Word_to_alphabetic
sage: translate_numeric_Word_to_alphabetic(Word([2,-1,-2,3,1,3]))
word: bABcac

stallings_graphs.about_words.translate_numeric_to_character(x)
Return the character equivalent of a numerical letter.

x is expected to be a non-zero integer in the interval \([-26; 26]\). A ValueError is raised otherwise. The numeric equivalent is a lower case character if x > 0 and an upper case character otherwise.

INPUT:
- x – integer

OUTPUT:
- character

EXAMPLES:

sage: from stallings_graphs.about_words import translate_numeric_to_character
sage: translate_numeric_to_character(16)
'p'
The methods for the class FinitelyGeneratedSubgroup use a number of ancillary functions. These are the functions which deal with graphs and automata, in the context of group theory.

A word is a string of characters from either a numerical or an alphabetical set of letters: alphabet_type='123' or 'abc'.

alphabet_type='123': The positive letters form an interval [1, r]. Their inverses (a.k.a. negative letters) are the corresponding negative integers. The symmetrized alphabet is the union of positive and negative letters (zero is NOT a letter). The rank of a word is the maximal absolute value of a letter occurring in the word. When represented in a (say LaTeX) file (.tex, .pdf), the letters are written $a_i$.

alphabet_type='abc': positive letters are lower case (at most 26 letters, $a$:$z$) and their inverses are the corresponding upper case letters (A:Z).

Automata are objects of class DiGraph whose edge labels are positive letters (always numerical). When automata are visualized, the value of alphabet_type determines how these edge labels will appear. In most cases, the vertex set of a DiGraph is a set of integers, usually of the form $[0..n]$.

We have functions to:

• compute the bouquet of a list of Word (of alphabet_type '123' or 'abc')
• extract from a DiGraph the list of its transitions (one for each letter labeling an edge)
• determine whether a DiGraph is deterministic and in that case, compute the transition functions (one for each letter labeling an edge)
• determine whether a DiGraph is folded and in that case, compute the corresponding tuple of partial injections (objects of class PartialInjection)
• relabel vertices
• permute the names of two vertices
• normalize its vertex set (so it is $[0..n]$)
• compute the image of a word in a DiGraph
• prune a DiGraph
• cyclically reduce a DiGraph
• compute the fibered product of two objects of class DiGraph
• prepare a rooted DiGraph to be visualized using TikzPicture with alphabet_type '123' or 'abc'
• show a rooted DiGraph (using graph.plot), with the root in a different color

EXAMPLES:

sage: from stallings_graphs.about_words import random_reduced_word
sage: L = ['aBBBaaaab', 'BBAbbABABBA', 'bbAhAbbaabb']
sage: from stallings_graphs.about_automata import bouquet
sage: G = bouquet(L, alphabet_type='abc')
sage: from stallings_graphs.about_folding import NT_fold
sage: GG = NT_fold(G)
sage: GG
Looped multi-digraph on 23 vertices
Return the list of partial injections (in fact: objects of class PartialInjection) on the set of vertices of this graph. 

G is expected to be a folded DiGraph (a ValueError is raised otherwise) with edge labels in \([1..r]\) and vertex set equal to \([0, n - 1]\) (\(r\) and \(n\) not given). Folded means that the graph is deterministic and co-deterministic or, equivalently, that every edge label defines a partial injection on the vertex set. The list returned has size \(r\), and represents the partial injections (from the class PartialInjection) induced by edge labels 1, 2, ..., \(r\), respectively.

**INPUT:**

- \(G\) – DiGraph

**OUTPUT:**

- List of lists

**EXAMPLES**

```python
sage: from stallings_graphs import PartialInjection
sage: from stallings_graphs.about_automata import DiGraph_to_list_of_PartialInjection
sage: from stallings_graphs.about_folding import NT_fold
sage: L = [[[3,1,-2,-1,-3],[1,2,-1,2,1,2],[3,2,-3,-3,1]],
         [[3,1,-2,-1,-3],[1,2,-1,2,1,2],[3,2,-3,-3,1]]]
sage: G = bouquet(L)
sage: GG = NT_fold(G)
sage: DiGraph_to_list_of_PartialInjection(GG)
[A partial injection of size 10, whose domain has size 4, 
A partial injection of size 10, whose domain has size 5, 
A partial injection of size 10, whose domain has size 3]
```

```python
sage: L = [[[3,1,-2,-1,-3],[1,2,-1,2,1,2],[3,2,-3,-3,1],[5]],
         [[3,1,-2,-1,-3],[1,2,-1,2,1,2],[3,2,-3,-3,1],[5]]]
sage: G = bouquet(L)
sage: GG = NT_fold(G)
sage: DiGraph_to_list_of_PartialInjection(GG)
[A partial injection of size 10, whose domain has size 4, 
A partial injection of size 10, whose domain has size 5, 
A partial injection of size 10, whose domain has size 3, 
A partial injection of size 10, whose domain has size 0, 
A partial injection of size 10, whose domain has size 1]
```

Return whether two folded DiGraph are the Stallings graphs of the same subgroup, possibly with different vertex labels.

The two first arguments are expected to be folded DiGraph. They represent the same subgroup if the corresponding tuples of PartialInjection coincide, up to a relabeling of the vertices which fixes the base vertex (vertex 0). That is: if the partial injections defining the second argument are obtained by conjugating the partial injections defining the first argument by a permutation which fixes 0. In verbose mode: details are given as to why the graphs do not represent the same subgroup or, if they do, which permutation fixing 0 maps one to the other. In certificate mode: if the subgroups are the same, the output is (True, sigma) where sigma is a conjugating permutation; otherwise the output is (False, None).

**INPUT:**

- \(G\) – DiGraph
- \(H\) – DiGraph

**EXAMPLES**

```python
sage: L = [[[3,1,-2,-1,-3],[1,2,-1,2,1,2],[3,2,-3,-3,1]],
         [[3,1,-2,-1,-3],[1,2,-1,2,1,2],[3,2,-3,-3,1]]]
sage: G = bouquet(L)
sage: GG = NT_fold(G)
```

• certificate – boolean
• verbose – boolean

OUTPUT: If certificate is set to False:

• a boolean, if certificate is set to False; and if certificate is set to True, a tuple of the form (False, None) or (True, sigma) where sigma is the conjugating permutation

EXAMPLES

\begin{verbatim}
sage: from stallings_graphs import FinitelyGeneratedSubgroup, PartialInjection
sage: from stallings_graphs.about_automata import are_equal_as_rooted_unlabeled
sage: L1 = [PartialInjection([1,2,4,5,3]), PartialInjection([0,4,5,3,None])]
sage: H1 = FinitelyGeneratedSubgroup(L1)
sage: G1 = H1.stallings_graph()
sage: L2 = [PartialInjection([1,2,5,4,3]), PartialInjection([0,5,4,3,None])]
sage: H2 = FinitelyGeneratedSubgroup(L2)
sage: G2 = H2.stallings_graph()
sage: are_equal_as_rooted_unlabeled(G1,G2)
False
sage: are_equal_as_rooted_unlabeled(G1,G3)
True
sage: (b, tau) = are_equal_as_rooted_unlabeled(G1,G3,certificate=True)
sage: tau
[0, 1, 2, 5, 3, 4]
sage: H = FinitelyGeneratedSubgroup([[]])
sage: G1 = H.stallings_graph()
sage: K = FinitelyGeneratedSubgroup.from_generators([])
sage: G2 = K.stallings_graph()
sage: b,tau = are_equal_as_rooted_unlabeled(G1,G2,certificate=True)
(True, [0])
sage: H1 = FinitelyGeneratedSubgroup.from_generators(['a','b'],alphabet_type='abc')
sage: G1 = H1.stallings_graph()
sage: H2 = FinitelyGeneratedSubgroup.from_generators(['ab','ba','aba'],alphabet_type='abc')
sage: G2 = H2.stallings_graph()
sage: are_equal_as_rooted_unlabeled(G1,G2)
True
\end{verbatim}

ALGORITHM:

One first checks whether both inputs represent subgroups in the same free group (same maximum value of a label) and have the same size (number of vertices). Then whether there is a permutation of the vertices other than the base vertex (vertex 0) which maps each transition (a partial injection) of the first argument to the corresponding partial injection of the second. Since a True output is least likely, the algorithm eliminates the most common and superficial reason for not being the same: different profiles of partial injections (ordered lists of lengths of sequences, resp. cycles, in the two subgroups. Then the algorithm attempts to build a conjugating permutation (unique if it exists). This results in a long code, experimentally much faster to run (on randomly generated subgroups constructed to be conjugated) than the shorter code relying on labeled graph isomorphism.

\begin{verbatim}
stallings_graphs.about_automata.bouquet(list_of_Words, alphabet_type='123', check=False)
\end{verbatim}

Return the bouquet of loops labeled by this list of words.

list_of_Words is expected to be a List of objects of class Word on a symmetric alphabet which is numerical (alphabet_type = '123') or consists of letters (alphabet_type = 'abc'). The option check = True
This function verifies that this argument is a valid list of `Word` over the given `alphabet_type`. The bouquet in question is a `DiGraph` with vertex set of the form `[0..n]`, where every word in `list_of_Words` labels a loop at vertex 0. The edges are labeled by letters from the symmetric alphabet.

**INPUT:**
- `list_of_Words` – List of `Word`
- `alphabet_type` – string, which is either '123' or 'abc'
- `check` – boolean

**OUTPUT:**
- `DiGraph`

**EXAMPLES:**

```python
sage: from stallings_graphs.about_automata import bouquet, transitions
sage: L = [[4, 1, 1, -4], [-4, -2, -1, 2, -1], [4]]
```
```
sage: G = bouquet(L)
sage: G
Looped multi-digraph on 8 vertices
```
```
sage: GG = bouquet([[[-1]]])
sage: GG
Looped multi-digraph on 1 vertex
```

**stallings_graphs.about_automata.cyclic_reduction**

This function return the cyclic reduction of this `DiGraph`. If `trace` is set to True, the output includes, in addition to the cyclic reduction of `G`, the `Word` which labels the shortest path from vertex 0 to a vertex that is preserved in the algorithm.

**INPUT:**
- `G` – `DiGraph`
- `trace` – boolean

**OUTPUT:**
- a `Digraph` if `trace` is set to False; and a pair `(GG, w)` of a `DiGraph` and a `Word` otherwise.

**EXAMPLES:**

```python
sage: from stallings_graphs import FinitelyGeneratedSubgroup
sage: from stallings_graphs.about_automata import cyclic_reduction, pruning, normalize_vertex_names
sage: L = ['ababA', 'aBabbabA', 'aababABAA']
```
```
sage: H = FinitelyGeneratedSubgroup.from_generators(L, alphabet_type='abc')
```
```
sage: G = H.stallings_graph()
sage: G
Looped multi-digraph on 6 vertices
```
```
sage: G2 = cyclic_reduction(G)
sage: G2
Looped multi-digraph on 5 vertices
```
```python
sage: G2,w = cyclic_reduction(G,trace=True)
sage: w
word: 1
```

**stallings_graphs.about_automata.exchange_labels(G, i, j)**

Exchanges the given vertex names in this DiGraph.

G is expected to be a DiGraph with vertices 0, ..., n - 1, and i, j are expected to be vertices of G. Outputs an isomorphic DiGraph, where the names of the vertices i and j have been exchanged.

**INPUT:**
- G – DiGraph
- i – integer
- j – integer

**OUTPUT:**
- DiGraph

**EXAMPLES:**

```python
sage: from stallings_graphs.about_automata import exchange_labels
sage: G = DiGraph([(0,0), (0,1), (0,2), (1,0), (1,2), (2,0), (2,1)], loops=True, multiedges=True)
sage: G = exchange_labels(G,0,3)
sage: G
Looped multi-digraph on 5 vertices
```

**stallings_graphs.about_automata.fibered_product(G1, G2)**

Compute the fibered product (a.k.a. direct product) of two edge-labeled graphs.

G1 and G2 are assumed to be of class DiGraph, with edges labeled by positive integers. Their fibered product is the DiGraph whose vertex set is the Cartesian product of the sets of vertices of G1 and G2 and whose edges are as follows: there is an \(a\)-labeled edge from \((u_1, v_1)\) to \((v_1, v_2)\) if and only if G1 has an \(a\)-labeled edge from \(u_1\) to \(v_1\) and G2 has an \(a\)-labeled edge from \(u_2\) to \(v_2\).

**INPUT:**
- G1 – DiGraph
- G2 – DiGraph

**OUTPUT:**
- DiGraph

**EXAMPLES**

```python
sage: from stallings_graphs.about_automata import fibered_product
sage: V1 = range(3)
sage: E1 = [(i,j,abs(i-j)) for i in V1 for j in V1]
sage: G1 = DiGraph([V1,E1], loops=True, multiedges=True)
sage: V2 = range(3)
sage: E2 = [(i,j,abs(i-j+1)) for i in V2 for j in V2]
sage: G2 = DiGraph([V2,E2], loops=True, multiedges=True)
sage: G12 = fibered_product(G1,G2)
sage: G12.vertices()
[(0, 0), (0, 1), (0, 2), (1, 0), (1, 1), (1, 2), (2, 0), (2, 1), (2, 2)]
sage: len(G12.edges())
26
```
stallings_graphs.about_automata.image_of_word\((G, w, qinitial=0, trace=False)\)

Return the vertex reached after reading this word in this graph (None if it cannot be read).

G is expected to be a DiGraph whose edges are labeled deterministically and codeterministically (folded DiGraph) by a numerical alphabet (typically, G is a Stallings graph) [not verified], w is a Word on a numerical alphabet and qinitial is a vertex of G. If one can read w from qinitial in G, the output is the vertex reached. If one cannot, the output is None. The option trace=True documents the situation if w cannot be read in G: the output is the triple (None, length_read, last_vertex_visited) where length_read is the length of the longest prefix u of w which can be read in G starting at qinitial and last_vertex_visited is the vertex reached after reading u.

**INPUT:**

- G – DiGraph
- w – Word
- qinitial – integer (state of G)
- trace – boolean

**OUTPUT:**

- integer or None if trace=False; and if trace=True, a triple consisting of three integers or None and two integers

**EXAMPLES**

```python
sage: from stallings_graphs import FinitelyGeneratedSubgroup
sage: from stallings_graphs.about_automata import image_of_word
sage: L = ['ab', 'ba', 'aBaa']

sage: H = FinitelyGeneratedSubgroup.from_generators(L, alphabet_type = 'abc')

sage: G = H.stallings_graph()

sage: w = Word([1,-2,1,-1,1])

sage: image_of_word(G,w, qinitial = 0,trace = True)
(2, 5, 2)

sage: image_of_word(G,w)
2

sage: ww = Word([1,2,-1,-2])

sage: image_of_word(G,ww)
0

sage: w = Word([2,2,-1])

sage: image_of_word(G,w, qinitial = 0,trace = True)
(None, 1, 2)

sage: image_of_word(G,w) is None
True

sage: w = Word()

sage: image_of_word(G,w, qinitial = 0,trace = True)
(0, 0, 0)

sage: H = FinitelyGeneratedSubgroup([])

sage: G = H.stallings_graph()

sage: w = Word([2,2,-1])

sage: image_of_word(G,w, qinitial = 0,trace = True)
(None, 0, 0)
```

Chapter 2. Finitely generated subgroups of free groups
sage: w = Word()
sage: image_of_word(G, w, qinitial = 0, trace = True)
(0, 0, 0)

stallings_graphs.about_automata.is_deterministic(digr)
Return whether this DiGraph is deterministic.

digr is expected to be a DiGraph with labeled edges and with vertex set of the form [0..n]. It is said to be
deterministic if for each vertex v and each label a, there is at most one a-labeled edge out of v.

INPUT:
• digr – DiGraph

OUTPUT:
• boolean

EXAMPLES

sage: from stallings_graphs.about_automata import bouquet, is_deterministic
sage: L = [[3,1,1,-3], [-3, -2, -1, 2, -1]]
sage: digr = bouquet(L)
sage: is_deterministic(digr)
False

stallings_graphs.about_automata.is_folded(G)
Return whether this DiGraph is folded (deterministic and co-deterministic).

G is expected to be a DiGraph with labeled edges and with vertex set of the form [0..n]. It is said to be deter-
mindistic if for each vertex v and each label a, there is at most one a-labeled edge out of v; and co-deterministic
if for each vertex v and label a, there is at most one a-labeled edge into v.

INPUT:
• G – DiGraph

OUTPUT:
• boolean

EXAMPLES

sage: from stallings_graphs.about_automata import bouquet, is_folded
sage: L = [[-3,1,2,1], [3,2,1]]
sage: G = bouquet(L)
sage: is_folded(G)
False

stallings_graphs.about_automata.normalize_vertex_names(G)
Rename the vertices of this DiGraph so they are of the form [0..n-1].

G is expected to be a DiGraph with vertex set a set of integers (or at least sortable elements). The output
DiGraph is isomorphic to G, with vertices labeled [0..n-1], where the new vertex names respect the original
order on vertex identifiers.

INPUT:
• G – DiGraph

OUTPUT:
• DiGraph

EXAMPLES:
stallings_graphs Documentation, Release 0.2

```python
sage: from stallings_graphs.about_automata import normalize_vertex_names
sage: G = DiGraph([(1,3,7,10,11), (1,1,1), (1,3,2), (1,11,3), (3,1,2), (3,7,1), (3,7,3), (7,10,2),
(7,10,3), (11,10,1)], format='vertices_and_edges', loops=True, multiedges=True)
```

```python
sage: GG = normalize_vertex_names(G)
sage: GG.vertices()
[0, 1, 2, 3]
```

```python
stallings_graphs.about_automata.prepare4visualization_graph(G, alphabet_type='abc', visu_tool='tikz')
```

Return a DiGraph ready for visualization, with good-looking edge labels.

G is expected to be a DiGraph with numerical edge labels. The value of `alphabet_type` decides whether these labels will appear as \(a_1, \ldots, a_r\) (`alphabet_type='123'`) or as \(a, b, c, \ldots, z\) (`alphabet_type='abc'`). The argument `visu_tool` prepares the usage of the `plot` method for graphs (`visu_tool='plot'`) or of Sébastien Labbé's `TikzPicture` method (`visu_tool='tikz'`).

**INPUT:**
- `G` – DiGraph
- `alphabet_type` – string, which can be either 'abc' or '123'
- `visu_tool` – string, which can be either 'plot' or 'tikz'

**OUTPUT:**
- DiGraph

**EXAMPLES:**

```python
sage: from stallings_graphs.about_automata import prepare4visualization_graph, bouquet, show_rooted_graph
sage: from stallings_graphs.about_folding import NT_fold
sage: L = [[3,1,-2,-1,3],[1,2,-1,-2,1,2],[1,2,-3,-3,1]]
```
```
sage: G = bouquet(L)
sage: GG = NT_fold(G)
sage: GGG = prepare4visualization_graph(GG,alphabet_type='abc',visu_tool='plot')
sage: show_rooted_graph(GGG,0)
```
```
Graphics object consisting of 41 graphics primitives
```
```
sage: GGG = prepare4visualization_graph(GG,alphabet_type='abc',visu_tool='tikz')
sage: from slabbe import TikzPicture
```
```
sage: t = TikzPicture.from_graph(GGG, merge_multiedges=False, edge_labels=True, color_by_label=False,
prog='dot')
sage: t.tex() # not tested
sage: t.pdf() # not tested
sage: t.png() # not tested
```

**TESTS:**

Dear User, we made sure that production of images works:

```python
sage: _ = t.tex()
sage: _ = t.pdf(view=False)
sage: _ = t.png(view=False)
```

```python
stallings_graphs.about_automata.pruning(G)
```

Prune a DiGraph, by iteratively removing degree 1 vertices other than the base vertex (vertex 0).

G is expected to be a DiGraph with numerical edge labels, with vertex set of the form \([0..n]\). The output is another DiGraph, obtained from \(G\) by iteratively removing the degree 1 vertices other than vertex 0 – and relabeling the vertices other than 0 so that the vertex set is of the form \([0..m]\).

**INPUT:**
- `G` – DiGraph

Chapter 2. Finitely generated subgroups of free groups

36
OUT: • DiGraph

EXAMPLES
```
sage: from stallings_graphs.about_automata import bouquet, pruning
sage: from stallings_graphs.about_folding import NT_fold
sage: L = [[2,1,-2,1,2,-2],[2,3,1,-3,3,1,-2]]
sage: G = bouquet(L)
sage: GG = NT_fold(G)
sage: GG
Looped multi-digraph on 5 vertices
sage: GGG = pruning(GG)
sage: GGG
Looped multi-digraph on 3 vertices
```

stallings_graphs.about_automata.relabeling(G, R)
Relabels this DiGraph using the given permutation.

G is expected to be a DiGraph with vertices labeled in [0..n]. R is expected to be a permutation of [0..n]. (No verification is made of that fact.) The output DiGraph is obtained from G by relabeling the vertices of G using the permutation R.

INPUT:
• G – DiGraph
• R – List

OUTPUT:
• DiGraph

EXAMPLES:
```
sage: from stallings_graphs.about_automata import relabeling
sage: G = DiGraph([[0,1,2,3,4],[[0,0,1,0],[0,4,2,2],[1,0,2],[1,2,1],[1,2,3],[2,3,2],[2,3,3],[3,0,1]]], format='vertices_and_edges', loops=True, multiedges=True)
sage: R = [4,1,0,3,2]
sage: GG = relabeling(G,R)
sage: GG
Looped multi-digraph on 5 vertices
```

stallings_graphs.about_automata.show_rooted_graph(G, base_vertex=0)
Show this rooted DiGraph, emphasizing its base vertex, using the graph.plot method.

G is expected to be a DiGraph with at least one vertex, with a distinguished base_vertex. The graph.plot function is used to show G. The declared base_vertex is colored green, the other vertices are colored white.

INPUT:
• G – DiGraph
• base_vertex – an object which is a vertex of G

OUTPUT:
• A graphics object

EXAMPLES
sage: from stallings_graphs.about_automata import prepare4visualization_graph, bouquet, show_rooted_graph
sage: from stallings_graphs.about_folding import NT_fold
sage: L = [[3,1,-2,-1,3],[1,2,-1,-2,1,2],[1,2,-3,-3,1]]
sage: G = bouquet(L)
sage: GG = NT_fold(G)
sage: show_rooted_graph(GG, base_vertex=0)
Graphics object consisting of 41 graphics primitives

stallings_graphs.about_automata.transition_function(digr)

Return a dictionary of the transitions (edges) of this DiGraph.

digr is expected to be a deterministic DiGraph (a ValueError is raised otherwise), with edge labels positive integers. The output dictionary maps each edge label to a list: the image of edge label \(a\) is a list indexed by the vertex set, where the \(v\)-entry is None if one cannot read \(a\) from \(v\), and the result of the \(a\)-transition from \(v\) otherwise.

INPUT:

• digr – DiGraph

OUTPUT:

• dictionary

EXAMPLES:

sage: from stallings_graphs.about_automata import bouquet, transition_function
sage: L = [[3,1,-2,-1,3],[1,2,-1,-2,1,2],[-1,2,-3,-3,1]]
sage: digr = bouquet(L)
sage: transition_function(digr)
{1: [5, 2, None, None, 3, None, None, 6, 9, None, None, 0],
2: [None, None, None, 2, None, 6, None, None, 7, 9, 11, None, None, None],
3: [1, None, None, None, 6, None, None, None, None, None, None, None, 11, 12]}

stallings_graphs.about_automata.transitions(digr)

Return a dictionary of the transitions (edges) of this DiGraph, organized by edge labels.

digr is expected to be a DiGraph, with labeled edges and with vertex set of the form \([0..n]\). The output dictionary maps each edge label to a list of sets: the image of edge label \(a\) is a list indexed by the vertex set, where the \(v\)-entry is the set of end vertices of \(a\)-labeled edges out of \(v\), or None if that set is empty.

INPUT:

• digr – DiGraph

OUTPUT:

• dictionary

EXAMPLES:

sage: from stallings_graphs.about_automata import bouquet, transitions
sage: L = [[4,1,1,-4],[-4, -2, -1, 2, -1]]
sage: G = bouquet(L)
sage: transitions(G)
{1: [{7}, {2}, {3}, None, None, None, {5}, None],
2: [None, None, None, None, None, {4}, {7}, None],
4: [{1, 3}, None, None, None, {0}, None, None, None]}
2.4 The folding algorithm

The methods for the class FinitelyGeneratedSubgroup use a number of ancillary functions. These are the functions which deal with the crucial operation of folding a DiGraph.

The algorithm used here is based on Nicholas Touikan’s article [T2006] and it ought to have time complexity $O(n \log^* n)$ — that is: very efficient. It uses in a crucial way the Union-Find algorithm, implemented in the DisjointSet class.

The DiGraph to be folded is expected to have numerical edge labels and to have a vertex set of the form [0..n].

EXAMPLES:

```python
sage: from stallings_graphs.about_words import random_reduced_word
sage: L = ['aABBBaaaab', 'BBbBABABA', 'bbAbAbaabb']
sage: from stallings_graphs.about_automata import bouquet
sage: G = bouquet(L, alphabet_type='abc')
sage: from stallings_graphs.about_folding import NT_fold
sage: GG = NT_fold(G)
sage: GG
Looped multi-digraph on 23 vertices
```

AUTHOR:

- Pascal WEIL, CNRS, Univ. Bordeaux, LaBRI <pascal.weil@cnrs.fr>: initial version (2018-06-09)

stallings_graphs.about_folding.NT_data_structures_initialization(digr)

Return the necessary data to initiate the folding of a labeled DiGraph.

digr is expected to be a labeled DiGraph, with vertex set of the form [0..n] and edges labeled by integers in [1..r]. In this preliminary step of the folding algorithm, the edges of digr are organized in a dictionary of dictionaries and the vertices of digr are organized in a DisjointSet structure (to later use the union-find algorithm). The dictionary of dictionaries is a variant of the data structure used by Nicholas Touikan in [T2006].

INPUT:

- digr – DiGraph

OUTPUT:

- A tuple consisting of a dictionary of dictionaries and a DisjointSet object

EXAMPLES

```python
sage: from stallings_graphs.about_automata import bouquet
sage: from stallings_graphs.about_folding import NT_data_structures_initialization
sage: L = [[3,1,-2,-1,3], [1,2,-1,-2,1,2], [1,2,-3,-3,1]]
sage: G = bouquet(L)
sage: NT_data_structures_initialization(G)
{(0): [13, 5, 10],
  1: [1, 2],
  2: [1],
  3: [4],
  4: [3],
  5: [0],
  6: [7],
  7: [6],
  8: [9],
  9: [8],
  10: [0],
  11: [set(0), set(0)],
  12: [set(0), set(0)],
  13: [set(0), set(0)],
(continues on next page)
```
stallings_graphs.about_folding.NT_fold(digr)

Returns the folded version of this DiGraph (with base vertex 0).

*digr* is expected to be a DiGraph with vertex set of the form \([0..n]\). The base vertex after folding is still called 0. The set of vertices of the output DiGraph is of the form \([0..m]\): this is not reflecting the name of vertices in the original DiGraph – except for the base vertex 0.

**INPUT:**

- *digr* – DiGraph

**OUTPUT:**

- DiGraph

**EXAMPLE**

```
sage: from stallings_graphs.about_words import translate_alphabetic_Word_to_numeric
sage: from stallings_graphs.about_automata import show_rooted_graph, bouquet
sage: from stallings_graphs.about_folding import NT_fold
sage: L1 = ['bABcac', 'abcBA', 'baaCB', 'abABcaCA']
sage: L2 = [translate_alphabetic_Word_to_numeric(w) for w in L1]
sage: G = bouquet(L2)
sage: GG = NT_fold(G)
sage: show_rooted_graph(GG, base_vertex=0)
Graphics object consisting of 62 graphics primitives
```

stallings_graphs.about_folding.NT_fold_edge(NT_vertices, NT_edge_structure, NT_unfolded, u, v1, v2)

Performs the crucial step of folding two edges.

*NT_vertices*, *NT_edge_structure* are expected to be the data structures (see *NT_data_structures_initialization*) associated with a DiGraph. *NT_unfolded* is the current set of unfolded vertices, *u* sits in that set, *v1* and *v2* are distinct vertices such that, for some letter \(a\) (a key in *NT_edge_structure*), *v1* and *v2* are both in *NT_edge_structure*\(a\)[0][\{0\}] (outgoing edges) or both in
The method returns updated versions of \texttt{NT_vertices}, \texttt{NT_edge_structure}, and \texttt{NT_unfolded} after the \texttt{a}-labeled edges out of \texttt{u} and into \texttt{v1} and \texttt{v2} (resp. into \texttt{u} out of \texttt{v1} and \texttt{v2}) are merged.

**INPUT:**

- \texttt{NT_vertices} – \texttt{DisjointSet}
- \texttt{NT_edge_structure} – dictionary of dictionaries
- \texttt{NT_unfolded} – set
- \texttt{u} – element
- \texttt{v1} – element
- \texttt{v2} – element

**OUTPUT:**

- the input objects \texttt{NT_vertices}, \texttt{NT_edge_structure} and \texttt{NT_unfolded} are modified in place

**EXAMPLES**

```python
sage: from stallings_graphs.about_automata import bouquet
sage: from stallings_graphs.about_folding import NT_data_structures_initialization, NT_initially_unfolded_construction, NT_fold_edge
sage: L = [[3,1,-2,-1,3],[1,2,-1,-2,1,2],[1,2,-3,-3,1]]
```

```python
sage: G = bouquet(L)
```

```python
sage: NT_vertices, NT_edge_structure = NT_data_structures_initialization(G)
```

```python
sage: NT_unfolded = set([0])
```

```python
sage: NT_fold_edge(NT_vertices, NT_edge_structure, NT_unfolded, 0, 5, 10)
```

```python
stallings_graphs.about_folding.NT_initially_unfolded_construction(digr, NT_vertices, NT_edge_structure)
```

Returns the set of unfolded vertices in this DiGraph at the beginning of the folding algorithm.

digr is expected to be a DiGraph, \texttt{NT_vertices} is a \texttt{DisjointSet} structure based on the vertices of \texttt{digr} and \texttt{NT_edge_structure} is a dictionary based on the edges of \texttt{digr}. This method is meant to be used once, when the input defining a subgroup is an NFA with one initial-final state.

**INPUT:**

- \texttt{digr} – DiGraph
- \texttt{NT_vertices} – \texttt{DisjointSet}
- \texttt{NT_edge_structure} – dictionary of dictionaries

**OUTPUT:**

- set

**EXAMPLES**

```python
sage: from stallings_graphs.about_automata import bouquet
sage: from stallings_graphs.about_folding import NT_data_structures_initialization, NT_initially_unfolded_construction, NT_fold_edge
sage: L = [[3,1,-2,-1,3],[1,2,-1,-2,1,2],[1,2,-3,-3,1]]
```

```python
sage: G = bouquet(L)
```

```python
sage: NT_vertices, NT_edge_structure = NT_data_structures_initialization(G)
```

```python
sage: NT_initially_unfolded_construction(G, NT_vertices, NT_edge_structure)
```

```python
{0}
```

```python
stallings_graphs.about_folding.NT_is_vertex_unfolded(v, NT_vertices, NT_edge_structure)
```

Return whether this vertex is unfolded in the given \texttt{DisjointSet} structure.

**2.4. The folding algorithm**
v is expected to be an element of the vertex set $V$ of a graph, $NT\_vertices$ is a DisjointSet object based on $V$ and $NT\_edge\_structure$ is a dictionary of dictionaries. The method detects whether the root of $v$ in $NT\_vertices$ is unfolded, that is, whether for some letter $i$, $NT\_edge\_structure[i][w][0]$ or $NT\_edge\_structure[i][w][1]$ has at least 2 elements — after updating these sets using the $NT\_vertices$. find operator.

INPUT:
- digr – DiGraph
- $NT\_vertices$ – DisjointSet
- $NT\_edge\_structure$ – dictionary of dictionaries

OUTPUT:
- boolean

EXAMPLES

```python
sage: from stallings_graphs.about_automata import bouquet
sage: from stallings_graphs.about_folding import NT_data_structures_initialization, NT_is_vertex_unfolded
sage: L = [[3,1,-2,-1,3],[1,2,-1,-2,1,2],[1,2,-3,-3,1]]
sage: G = bouquet(L)
```

```python
sage: NT_data_structures_initialization(G)
sage: NT_is_vertex_unfolded(0, NT_data_structures_initialization(G))
True
```

```python
sage: NT_is_vertex_unfolded(2, NT_data_structures_initialization(G))
False
```

### 2.5 Ancillary functions about bases

The methods for the class FinitelyGeneratedSubgroup use a number of ancillary functions. These are the functions which deal with bases: determining one, parsing a word along a given basis.

A word is a string of characters from either a numerical or an alphabetical set of letters: `alphabet_type='123'` or `'abc'`.

**alphabet_type='123'**: The positive letters form an interval $[1,r]$. Their inverses (aka negative letters) are the corresponding negative integers. The symmetrized alphabet is the union of positive and negative letters (zero is NOT a letter). The rank of a word is the maximal absolute value of a letter occurring in the word. When represented in a (say LaTeX) file (`.tex`, `.pdf`), the letters are written $a_i$.

**alphabet_type='abc'**: Positive letters are lower case (at most 26 letters, $a:z$) and their inverses are the corresponding upper case letters ($A:Z$).

Automata are objects of class DiGraph whose edge labels are positive letters (always numerical). When automata are visualized, the value of `alphabet_type` determines how these edge labels will appear. In most cases, the vertex set of a DiGraph is a set of integers, usually of the form $[0..n]$.

We have functions to:
- compute a spanning tree
- compute a basis specified by a spanning tree
- express a Word in a basis specified by a spanning tree

**EXAMPLES:**
AUTHOR:

- Pascal WEIL (2018-06-09): initial version CNRS, Univ. Bordeaux, LaBRI <pascal.weil@cnrs.fr>

`stallings_graphs.about_bases.basis_from_spanning_tree(G, T, D, root=0, alphabet_type='abc')`

Return the basis (of the space of loops of $G$ at the root vertex) specified by the spanning tree $T$.

$G$ is expected to be a folded DiGraph with numerical edge labels. $T$ (also a DiGraph) is expected to be a spanning tree of $G$. $D$ is expected to be a dictionary associating with each vertex $v$ of $G$ (and $T$) the word labeling the geodesic path in $T$ from root to $v$. The output basis is a list of objects of class `Word` on a numerical alphabet, one for each edge of $G$ that is not in $T$.

**INPUT:**
- $G$ – DiGraph
- $T$ – DiGraph
- $D$ – dictionary (the keys are the vertices of $G$ and the values are of class `Word`)
- $root$ – a vertex of $G$

**OUTPUT:**
- a list of objects of class `Word` (in numerical or alphabetic form)

**EXAMPLES:**

```python
sage: from stallings_graphs.about_bases import basis_from_spanning_tree
sage: generators = [[3,1,-2,-1,3], [1,2,-1,-2,1,2], [1,2,-3,-3,1]]
```

```python
sage: G = NT_fold(bouquet(generators))
```

```python
sage: T,L,D = spanning_tree_and_paths(G)
```

```python
sage: basis_from_spanning_tree(G,T,D,alphabet_type='123')
[word: -3,1,2,-1,-3, word: -2,-1,2,1,-2,-1, word: -1,3,3,-2,-1]
```

### Ancillary functions about bases

**`stallings_graphs.about_bases.basis_interpreter(L, C, alphabet_type='abc', check=False)`**

Returns the translations of a list of words into words (in numerical form) on basis $C$.

$L$ and $C$ are expected to be lists of words in the same format, alphabetic or numerical, specified by `alphabet_type`''. Each word in $L$ is expected to be in the subgroup generated by $C$. This is verified if check is set to True.

**INPUT:**
- $L$ – list of objects of type `Word`
- $C$ – list of objects of type `Word`
- `alphabet_type` – string, which can be either 'abc' or '123'
- `check` – boolean
OUTPUT:

- list of objects of type `Word` (in numerical form)

EXAMPLES:

```
sage: from stallings_graphs.finitely_generated_subgroup import FinitelyGeneratedSubgroup
sage: from stallings_graphs.about_bases import basis_interpreter
sage: generators = ['abbC','aabCa','aaCBa','cBa']
sage: w = Word('abcCabCbcA')
sage: basis_interpreter([Word([])], generators, alphabet_type = 'abc', check = False)
[word: ]
sage: basis_interpreter([w], generators, alphabet_type = 'abc', check = False)
[word: -3,2,-4,-3]
sage: ww = Word('abcCbabcCA')
sage: basis_interpreter([w,Word([])], generators, alphabet_type = 'abc', check = False)
[word: -3,2,-4,-3, word: ]
```

`stallings_graphs.about_bases.spanning_tree_and_paths(G, root=0)`

Return a spanning tree $T$ of this DiGraph, a list of the leaves of $T$, and shortest paths in $T$, from the root to each vertex.

$G$ is expected to be a DiGraph with numerical edge labels. Computes a spanning tree $T$ (also a DiGraph) by breadth first search starting at vertex root $\rightarrow$, along with a list of the non-root leaves of $T$, and a dictionary associating with each vertex $v$ the word labeling the geodesic path in $T$ from root to $v$.

INPUT:

- $G$ – DiGraph
- $\text{root}$ – a vertex of $G$

OUTPUT:

- a triple consisting of a DiGraph, a list and a dictionary

EXAMPLES:

```
sage: from stallings_graphs import FinitelyGeneratedSubgroup
sage: from stallings_graphs.about_bases import spanning_tree_and_paths
sage: L = [[3,1,-2,-1,3],[1,2,-1,-2,1,2],[1,2,-3,-3,1]]
sage: H = FinitelyGeneratedSubgroup.from_generators(L)
sage: G = H.stallings_graph()
sage: T, list_of_leaves, path_in_tree = spanning_tree_and_paths(G)
sage: T
Multi-digraph on 12 vertices
sage: list_of_leaves
[2, 10, 3, 6, 5]
sage: path_in_tree
{0: word: ,
  1: word: 3,
  2: word: 31,
  3: word: -3,1,
  4: word: -3,
  5: word: 1,2,-1,
  6: word: -2,-1,
  7: word: -2,
  8: word: 1,
  9: word: 12,
(continues on next page)
```
stallings_graphs.about_bases.tree_based_interpreter(w, G, T, root=0, alphabet_type='abc')

Return the expression of the Word $w$ in the basis (of the space of loops of $G$ at the root vertex) specified by the spanning tree $T$.

$w$ is expected to be a Word in alphabetic or numerical form, depending on $alphabet_type$'. In addition, `$w` is expected to label a loop at vertex `root` in the folded DiGraph $G$ (with numerical edge labels). $T$ (also a DiGraph) is expected to be a spanning tree of $G$. The output is a numerical Word which is the translation of $w$ in the alphabet of the basis defined by $T$.

**INPUT:**
- $G$ – DiGraph
- $T$ – DiGraph
- $w$ – Word
- $root$ – a vertex of $G$
- $alphabet_type$ – string, which can be either 'abc' or '123'

**OUTPUT:**
- Word (in numerical form)

**EXAMPLES:**

```python
sage: from stallings_graphs.about_automata import bouquet
sage: from stallings_graphs.about_bases import spanning_tree_and_paths, basis_from_spanning_tree, tree_based_interpreter
sage: from stallings_graphs.about_folding import NT_fold
sage: generators = ['abaa','ababb','ababab']
sage: G = NT_fold(bouquet(generators,alphabet_type = 'abc'))
sage: T,L,D = spanning_tree_and_paths(G)
sage: basis_from_spanning_tree(G,T,D)
[word: abaa, word: Abb, word: Bab]
```

```python
sage: w = Word('AbaabAABABab')
sage: tree_based_interpreter(w,G,T)
word: 2,3,3,-1,3
```

## 2.6 About morphisms between free groups: connecting with the train_track package

The methods for the class FinitelyGeneratedSubgroup use a number of ancillary functions. These are the functions which deal with morphisms between free groups.

More precisely, morphisms and automorphisms are handled by Thierry Coulbois’s train_track package. Here we provide mutual translations between objects of class Word, as used in stallings_graphs, and words as used in the train_track package. Specifically, we stick to words on a numerical alphabet ($alphabet_type='123'$) and to the train_track format $type='x0'$.

The translation is as follows: if $i$ is a positive integer, the corresponding letter is $xj$ with $j = i - 1$; if $i$ is a negative integer, the corresponding letter is $Xj$ with $j = -i - 1$.

We have functions to:
• translate a character, or a word, from one of the formats to the other
• define a FGendomorphism (this is a \`\`FreeGroupMorphism from train_track), by giving the list of images of the ambient free group basis
• compute the image of a word (alphabetic or numeric) by a FGendomorphism

We inherit the methods from train_track, to compose morphisms, to check whether they are invertible and, if so, to compute their inverse.

EXAMPLES:

```python
sage: from stallings_graphs.about_TC_morphisms import FGendomorphism
sage: L = ['ab', 'a']

sage: phi = FGendomorphism(L, alphabet_type='abc')

sage: phi
Morphism from Free Group on generators {x0, x1} to Free Group on generators {x0, x1}: x0->x0*x1,x1->x0
```

```python
sage: from stallings_graphs.about_TC_morphisms import image_of_Word_by_endomorphism
sage: w = 'abAbA'

sage: image_of_Word_by_endomorphism(phi, w, alphabet_type='abc')
word: abaBBA
```

AUTHOR:

• Pascal WEIL, CNRS, Univ. Bordeaux, LaBRI <pascal.weil@cnrs.fr> (2019-04-04): initial version.
EXAMPLES:

```
sage: from stallings_graphs.about_TC_morphisms import image_of_Word_by_endomorphism
sage: from stallings_graphs.about_TC_morphisms import FGendomorphism
sage: phi = FGendomorphism(L,alphabet_type='abc')

sage: w = 'abAbA'
sage: image_of_Word_by_endomorphism(phi, w, alphabet_type='abc')
word: abaBBA
```

```
sage: from stallings_graphs.about_TC_morphisms import translate_numeric_Word_to_x0_list
sage: translate_numeric_Word_to_x0_list([7,1,-2,3,-3])
['x6', 'x0', 'X1', 'x2', 'X2']
```

```
sage: from stallings_graphs.about_TC_morphisms import translate_numeric_to_x0_character
sage: translate_numeric_to_x0_character(7)
'x6'
sage: translate_numeric_to_x0_character(-7)
'X6'
```

```
sage: from stallings_graphs.about_TC_morphisms import translate_x0_character_to_numeric
sage: translate_x0_character_to_numeric('xj')
```

2.6. About morphisms between free groups: connecting with the `train_track` package
sage: from stallings_graphs.about_TC_morphisms import translate_x0_character_to_numeric
sage: translate_x0_character_to_numeric('x100')
101

sage: translate_x0_character_to_numeric('X100')
-101

stallings_graphs.about_TC_morphisms.translate_x0_word_to_numeric_Word(u)

Return the corresponding numeric Word.

u is expected to be a FreeGroup element in the sense of the train_track package, written with letters of the form xj or Xj, where j is a non-negative integer in decimal expansion.

INPUT:

• u – element of type train_track.free_group.FreeGroup_class_with_category.element_class (the free group elements in the train_track package)

OUTPUT:

• Word

EXAMPLES:

sage: from train_track import FreeGroupMorphism
sage: D = {'x0': ['x0', 'x1'], 'x1': ['X0']}
sage: phi = FreeGroupMorphism(D)
sage: print(phi)
x0->x0*x1,x1->x0^-1
sage: w = phi(['X0', 'x1', 'x0'])
sage: from stallings_graphs.about_TC_morphisms import translate_x0_word_to_numeric_Word
sage: translate_x0_word_to_numeric_Word(w)
word: -2,-1,2

2.7 Ancillary functions about free factors

The methods for the class FinitelyGeneratedSubgroup use a number of ancillary functions. These are the functions which deal with free factors: determining whether a subgroup is a free factor of the ambient group of another subgroup, deciding primitivity of a word, computing the lattice of algebraic extensions of a subgroup.

We have the following functions:

• SilvaWeil_free_factor_of_ambient: to decide whether a given subgroup is a free factor of the ambient group and, possibly give a basis of its complement

• SilvaWeil_free_factor_of: to decide whether a given subgroup is a free factor of another and, possibly, give a basis of its complement

The algorithm implemented in SilvaWeil_free_factor_of_ambient and SilvaWeil_free_factor_of is from [SW2008]. The worst-case complexity is polynomial in the size of the subgroups considered but exponential in the rank difference between them.

• set_of_possible_additional_generators: an ancillary function to find the additional generators that will lead to the overgroups of a subgroup obtained by identifying two vertices of the Stallings graph
• `compute_algebraic_extensions`: computes the semilattice of algebraic extensions of a subgroup noting those that are elementary algebraic and some of their inclusion relation (sufficiently many to include a Hasse diagram)

The algorithm implemented in `compute_algebraic_extensions` is from [MVW2007]. It requires verifying whether certain subgroups are free factors of others. This is done using the Silva Weil algorithm.

**EXAMPLES:**

```python
sage: from stallings_graphs import FinitelyGeneratedSubgroup
sage: from stallings_graphs.about_free_factors import SilvaWeil_free_factor_of_ambient, SilvaWeil_free_factor_of

sage: L1 = ['ac', 'bacd', 'ed']
sage: H1 = FinitelyGeneratedSubgroup.from_generators(L1, alphabet_type='abc')
sage: SilvaWeil_free_factor_of_ambient(H1, maxletter=0, complement=True)
(True, [word: 2,1,-5, word: -2])

sage: LH = [[-3,1,-2,-1,-1,-3,2,2,3], [-3,-1,-3,1,1,-3,-1,3,1,3,3], [-3,1,3,-1,-1,-3,1,1,3,-1,-1], [1,1,-3,-1,3,1,3,-1,-1]]
sage: LK = [[1,1,2,-1,3], [1,1,3,-1], [-3,1,3,-1,-1], [-3,1,1,3], [-3,2,3], [1,3,3]]
sage: H = FinitelyGeneratedSubgroup.from_generators(LH, alphabet_type='123')
sage: K = FinitelyGeneratedSubgroup.from_generators(LK, alphabet_type='123')
sage: SilvaWeil_free_factor_of(H, K, complement=True)
(False, '1st argument not contained in 2nd')

sage: LH = [[3,1,-2,-1,-1,-3,2,2,3], [-3,-1,-3,1,1,-3,-1,3,1,3,3], [-3,1,3,-1,-1,-3,1,1,3,-1,-1], [1,1,-3,-1,3,1,3,-1,-1]]
sage: LK = [[1,1,2,-1,3], [1,1,3,-1], [-3,1,3,-1,-1], [-3,1,1,3], [-3,2,3], [1,3,3]]
sage: H = FinitelyGeneratedSubgroup.from_generators(LH, alphabet_type='123')
sage: K = FinitelyGeneratedSubgroup.from_generators(LK, alphabet_type='123')
sage: SilvaWeil_free_factor_of(H, K, complement=True)
(False, '1st argument not contained in 2nd')

sage: H = FinitelyGeneratedSubgroup.from_generators(['bha', 'babaB'], alphabet_type='abc')
sage: K = FinitelyGeneratedSubgroup.from_generators(['a', 'bb', 'babaB'], alphabet_type='abc')
sage: SilvaWeil_free_factor_of(H, K, complement=True)
(True, [word: -2,-2])

```

**AUTHOR:**

• Pascal WEIL (2020-05-11): initial version CNRS, Univ. Bordeaux, LaBRI <pascal.weil@cnrs.fr>

**stallings_graphs.about_free_factors.SilvaWeil_free_factor_of(H, K, complement=True)**

If `complement` is set to `False`, returns whether \( H \) (a boolean). If `complement` is set to `True`, returns a pair of a boolean as above, and a string explaining why \( H \) is not a free factor, or a basis for a complement of \( H \) in \( K \) (in numerical form) if \( H \) is a free factor.

\( H \) and \( K \) are expected to be of type `FinitelyGeneratedSubgroup`; `complement` is expected to be a Boolean.

**INPUT:**

• \( \mathcal{H} \) – `FinitelyGeneratedSubgroup`
• $K \rightarrow \text{FinitelyGeneratedSubgroup}$

• complement $\rightarrow$ boolean

OUTPUT:

• a boolean if complement is set to False, and a pair consisting of a boolean and either a string or a list of Words in numerical form otherwise

ALGORITHM:

The algorithm implemented is from [P. Silva, P. Weil. On an algorithm to decide whether a free group is a free factor of another, Theoretical Informatics and Applications 42 (2008) 395-414]. Be aware that the worst-case complexity is polynomial in the size of $H$ and $K$ but exponential in the rank difference between $H$ and $K$.

EXAMPLES:

```python
sage: from stallings_graphs import FinitelyGeneratedSubgroup
sage: from stallings_graphs.about_free_factors import SilvaWeil_free_factor_of
sage: LH = [[2,-3,1,3,2,3,-2,-1,2,-3,-1], [3,1,1,3,-1], [1,3,-2,-1,2,-1,2], [3,2,3,1,2,-1]]
Sage: LK = [[2,-3], [1,1], [1,3,2,3,2,1,-3,-1], [3,2], [3,1,3,-1,1,3,-1], [1,3,3,-1], [1,3,1,-1]]
Sage: H = FinitelyGeneratedSubgroup.from_generators(LH, alphabet_type='123')
Sage: K = FinitelyGeneratedSubgroup.from_generators(LK, alphabet_type='123')
Sage: SilvaWeil_free_factor_of(H, K, complement = True)
(True, [word: 3,2,3,1,2,-1, word: 32, word: 3,1,3,-1, word: 11])
```

```python
sage: SilvaWeil_free_factor_of(H, K, complement = False)
True
```

```python
sage: LH = [[-3,1,-2,-1,-1,3,2,-2,3], [-3,-1,-1,3,1,1,-3,1,3,1,3,3], [-3,1,3,-1,-1,3,1,1,3,-1,1,3],[-3,1,3,1,3,1,1,3,-1,1,3]]
Sage: LK = [[1,1,2,-1,3], [1,1,3,-1], [-3,1,3,-1,1,3,1,3,-1,1,3]]
Sage: H = FinitelyGeneratedSubgroup.from_generators(LH, alphabet_type='123')
Sage: K = FinitelyGeneratedSubgroup.from_generators(LK, alphabet_type='123')
Sage: SilvaWeil_free_factor_of(H, K, complement = True)
(False, '1st argument is not a free factor of the second')
```

```python
sage: SilvaWeil_free_factor_of(H, K, complement = False)
False
```

```python
sage: LH = [[3,1,-1,-1,3,2,3,2,3], [-3,1,-1,3,1,1,-3,-1,3,1,3,3]]
Sage: LK = [[1,1,2,-1,3], [1,1,3,-1], [-3,1,3,-1,1,3,1,3,-1,1,3]]
Sage: H = FinitelyGeneratedSubgroup.from_generators(LH, alphabet_type='123')
Sage: K = FinitelyGeneratedSubgroup.from_generators(LK, alphabet_type='123')
Sage: SilvaWeil_free_factor_of(H, K, complement = True)
(False, '1st argument not contained in 2nd')
```

```python
sage: SilvaWeil_free_factor_of(H, K, complement = False)
False
```

```python
sage: H = FinitelyGeneratedSubgroup.from_generators(['bba', 'bAbB'], alphabet_type='abc')
Sage: K = FinitelyGeneratedSubgroup.from_generators(['a', 'bb', 'bAbB'], alphabet_type='abc')
Sage: SilvaWeil_free_factor_of(H, K, complement = True)
(True, [word: -2, -2])
```

```python
sage: SilvaWeil_free_factor_of(H, K, complement = False)
True
```
sage: H = FinitelyGeneratedSubgroup.from_generators(['a','b'], alphabet_type='abc')
sage: K = FinitelyGeneratedSubgroup.from_generators(['a', 'b', 'd'], alphabet_type='abc')
sage: SilvaWeil_free_factor_of(H, K, complement = True)
(True, word: 4)

sage: SilvaWeil_free_factor_of(H, K, complement = False)
True

2.7. Ancillary functions about free factors

If complement is set to False, returns whether $H$ is set to True, returns a pair of a boolean as above, and a string explaining why $H$ is not a free factor, or a basis for a complement of $H$ if $H$ is a free factor (in numerical form). In that case, the ambient free group is understood to be of rank the maximal letter occurring in $H$ if maxletter is set to 0, of rank maxletter otherwise.

$H$ is expected to be a FinitelyGeneratedSubgroup; maxletter is expected to be a non-negative integer, equal to 0 or greater than or equal to the maximal letter occurring in $H$; complement is expected to be a Boolean.

INPUT:

- $H$ – FinitelyGeneratedSubgroup
- maxletter – integer
- complement – boolean

OUTPUT:

- a boolean if complement is set to False, and a pair consisting of a boolean and either a string or a list of Words in numerical form otherwise

ALGORITHM:

The algorithm implemented is from [P. Silva, P. Weil. On an algorithm to decide whether a free group is a free factor of another, Theoretical Informatics and Applications 42 (2008) 395-414]. Be aware that the worst-case complexity is polynomial in the size of $H$ but exponential in the rank difference between $H$ and the ambient group.

EXAMPLES:

sage: from stallings_graphs import FinitelyGeneratedSubgroup
sage: from stallings_graphs.about_free_factors import SilvaWeil_free_factor_of_ambient
sage: L1 = ['ac', 'b', 'ed']
sage: H1 = FinitelyGeneratedSubgroup.from_generators(L1, alphabet_type='abc')
sage: SilvaWeil_free_factor_of_ambient(H1, maxletter = 0, complement = True)
(True, word: 2, word: -5)

sage: SilvaWeil_free_factor_of_ambient(H1, maxletter = 0, complement = False)
False

sage: L2 = ['acac', 'b', 'ed']
sage: H2 = FinitelyGeneratedSubgroup.from_generators(L2, alphabet_type='abc')
sage: SilvaWeil_free_factor_of_ambient(H2, maxletter = 0, complement = True)
(False, 'the 1st argument is not a free factor of the second')

sage: SilvaWeil_free_factor_of_ambient(H2, maxletter = 0, complement = False)
False

sage: H = FinitelyGeneratedSubgroup.from_generators(['A', 'd'], alphabet_type='abc')
sage: SilvaWeil_free_factor_of_ambient(H, complement = True)
(True, word: 2, word: 3)
stallings_graphs Documentation, Release 0.2

sage: SilvaWeil_free_factor_of_ambient(H, complement = False)
True

stallings_graphs.about_free_factors.compute_algebraic_extensions(H)

Returns detailed information on the semilattice of algebraic extensions of the subgroup \( H \): a dictionary whose keys are integers (without any particular meaning, except key 0 corresponds to \( H \) itself) and whose entries are a list of information on algebraic extensions: sets of parents and children (not a Hasse diagram of the containment relation, but including such a diagram), list of generators to be added to those of \( H \) to generate that particular extension, a set of words which help compute the immediate overgroups of this extension, and two boolean flags expressing, respectively, that the extension is e-algebraic and that it is not algebraic.

\( H \) is expected to be a FinitelyGeneratedSubgroup

INPUT:

- \( H \) – FinitelyGeneratedSubgroup

OUTPUT:

- a dictionary whose keys are integers and whose entries are lists of two sets of keys, a list of Words, a set of Words and two booleans

EXAMPLES:

```python
sage: from stallings_graphs import FinitelyGeneratedSubgroup
sage: from stallings_graphs.about_free_factors import compute_algebraic_extensions
sage: testgens = ['aba', 'bab']
sage: testH = FinitelyGeneratedSubgroup.from_generators(testgens, alphabet_type='abc')
sage: compute_algebraic_extensions(testH)
{0: [set(), {1}, [], {word: -1, word: -1, word: 1, word: 22}, True, False], 1: [{0}, set(), {word: -1}, set(), True, False]}
```

```python
sage: testgens = ['ab', 'cd']
sage: testH = FinitelyGeneratedSubgroup.from_generators(testgens, alphabet_type='abc')
sage: compute_algebraic_extensions(testH)
{0: [set(), set(), [], {word: -3, word: -1, word: 1, -3}, True, False]}
```

```python
sage: testgens = ['ABBaaBAbba', 'Baba', 'Ahababba', 'AbabbAbA', 'AbAbaba']
sage: testH = FinitelyGeneratedSubgroup.from_generators(testgens, alphabet_type='abc')
sage: compute_algebraic_extensions(testH)
{0: [set(), {3, 6, 11}, [], {word: -2, word: -1, word: 1, word: 2, word: 21, word: 221}, True, False], 3: [{0, 6, 11}, set(), {word: -1, word: -2, word: 1, word: 2, word: 21, word: 221}, True, False], 6: [{0, 11}, {3}, {word: 21}, {word: -1}, True, False],}
```
(continues on next page)
stallings_graphs.about_free_factors.set_of_possible_additional_generators($G$)

$G$ is expected to be the Stallings graph of a finitely generated subgroup of a free group. The function returns a set of Words of the form $u_pu_q^{-1}$, where $u_p$ is a path from the root vertex 0 to vertex $p$).

**INPUT:**
- $G$ – DiGraph

**OUTPUT:**
- a set of objects of type `Word`

**EXAMPLE:**

```python
sage: from stallings_graphs import FinitelyGeneratedSubgroup
sage: from stallings_graphs.about_free_factors import set_of_possible_additional_generators
sage: testgens = ['aba', 'bab']
sage: testH = FinitelyGeneratedSubgroup.from_generators(testgens, alphabet_type='abc')
sage: testG = testH.stallings_graph()
sage: set_of_possible_additional_generators(testG)
{word: -1, word: -1,2, word: 1,-2, word: 11, word: 22}
```
CHAPTER
THREE

PARTIAL INJECTIONS

3.1 The class PartialInjection

The class PartialInjection is meant to represent partial injections on a set of the form $[0..n-1]$. The representation of a PartialInjection is the list of images of $0,\ldots,n-1$, in that order, with None in places where the partial injection is not defined.

Methods implemented in this file:

- definition of a PartialInjection from its list of images
- random instance
- size – the length of the list of images (that is, the integer $n$ mentioned above)
- domain_size – the number of entries different from None
- inverse_partial_injection
- is_permutation
- orbit_decomposition

EXAMPLES:

```python
sage: from stallings_graphs import PartialInjection
sage: L = [0,3,None,2,4,None,5,1]
sage: p = PartialInjection(L)
sage: p
A partial injection of size 8, whose domain has size 6
```

```python
sage: pinj = PartialInjection.random_instance(10)
sage: pinj
# random
A partial injection of size 10, whose domain has size 7
```

AUTHOR:

- Pascal WEIL, CNRS, Univ. Bordeaux, LaBRI <pascal.weil@cnrs.fr>: initial version (2018-11-26)

```python
class stallings_graphs.partial_injections.PartialInjection(list_of_images, check=False)

Bases: sage.structure.sage_object.SageObject

Define the class PartialInjection.

The representation of a PartialInjection is a list of length $n$, whose entries are either elements of $[0..n-1]$ without any repetition, or None (the list of images of the elements of $[0..n-1]$). The integer $n$ is seen as the size of the PartialInjection.

A PartialInjection can be created from
```
• a list (its list of images)
or
• a random instance.

EXAMPLES

```python
sage: from stallings_graphs import PartialInjection
sage: L = [0, 3, None, 2, 4, None]
sage: p = PartialInjection(L)
sage: p
A partial injection of size 6, whose domain has size 4

sage: PartialInjection.random_instance(1000)  # random
A partial injection of size 1000, whose domain has size 969
```

`domain_size()`

Return the size of the domain of this `PartialInjection`.

Computes the size of the domain of this partial injection. If it has size \( n \), its domain size is the number of elements of \([0..n-1]\) with an image, that is, \( n - \ell \), where \( \ell \) is the number of `None`.

INPUT:

• `self` – `PartialInjection`

OUTPUT:

• integer

EXAMPLES:

```python
sage: from stallings_graphs import PartialInjection
sage: L = [0, 3, None, 2, 4, None]
sage: p = PartialInjection(L)
sage: p.domain_size()
4
```

`inverse_partial_injection()`

Return the inverse of a `PartialInjection`.

INPUT:

• `self` – `PartialInjection`

OUTPUT:

• a `PartialInjection`

EXAMPLES:

```python
sage: from stallings_graphs import PartialInjection
sage: p = PartialInjection([6, None, 5, 0, 11, 2, None, 3, 9, 1, 7, 10])
sage: q = p.inverse_partial_injection()
sage: q._list_of_images
[3, 9, 5, 7, None, 2, 0, 10, None, 8, 11, 4]
```

`is_permutation()`

Return whether whether a `PartialInjection` is a permutation.

A partial injection is a permutation if and only if its domain size is equal to its size.

INPUT:

• `self` – `PartialInjection`

56 Chapter 3. Partial injections
OUTPUT:
  • boolean

EXAMPLES:

```python
sage: from stallings_graphs import PartialInjection
sage: p = PartialInjection([6, None, 5, 0, 11, 2, None, 3, 9, 1, 7, 10])
```
```
sage: p.is_permutation()
False
```

```python
sage: p = PartialInjection([6, 4, 5, 0, 11, 2, 8, 3, 9, 1, 7, 10])
```
```
sage: p.is_permutation()
True
```

`orbit_decomposition()`
Return the orbit decomposition of a `PartialInjection`.

A partial injection admits a unique decomposition into its *maximal orbits*: a list of sequences and a list of cycles. The particular case of a permutation is that where each orbit is a cycle.

INPUT:
  • `self` – `PartialInjection`

OUTPUT:
  • List of 2 lists

EXAMPLES:

```python
sage: from stallings_graphs import PartialInjection
sage: p = PartialInjection([6, None, 5, 0, 11, 2, None, 3, 9, 1, 7, 10])
```
```
sage: p.orbit_decomposition()
([4, 11, 10, 7, 3, 0, 6], [8, 9, 1])
```

```python
sage: p = PartialInjection([6, 4, 5, 0, 11, 2, 8, 3, 9, 1, 7, 10])
```
```
sage: p.orbit_decomposition()
([4, 11, 10, 7, 3, 6], [8, 9, 1], [2, 5])
```

`static random_instance(size, statistics=False)`
Returns a randomly chosen `PartialInjection` of given size.

`size` is expected to be a positive integer. If `statistics` is set to `True`, the method also returns the number of orbits of the partial injection that are sequences. This number is expected to be asymptotically equivalent to $\sqrt{n}$, with standard deviation $o(\sqrt{n})$, where $n$ is equal to `size`.

INPUT:
  • `size` – integer
  • `statistics` – boolean

OUTPUT:
  • an object of the class `PartialInjection` if `statistics = False`; and a pair of an integer and an object of class `PartialInjection` otherwise

EXAMPLES:

```python
sage: from stallings_graphs import PartialInjection
sage: rand_inj = PartialInjection.random_instance(10)
```
```
sage: rand_inj._list_of_images  # random
[0, 4, 2, None, 3, 9, 7, 8, 6, None]
```

```python
sage: rand_inj = PartialInjection.random_instance(10)
```
```
sage: rand_inj._list_of_images  # random
[2, 4, 6, 0, 3, None, 9, 5, None, None]
```

3.1. The class PartialInjection
ALGORITHM:

The algorithm implemented here is that in [BNW2008]. It performs in linear time, except for a preprocessing which is cached.

size()

Return the size of this PartialInjection.

The size of a PartialInjection is the length of the list that represents it.

INPUT:

• self – PartialInjection

OUTPUT:

• integer

EXAMPLES:

```
sage: from stallings_graphs import PartialInjection
sage: L = [0,3,None,2,4,None]
sage: p = PartialInjection(L)
sage: p.size()
6
```

3.2 Ancillary functions about partial injections

The methods for the class PartialInjection use a number of ancillary functions.

We have the functions

• is_valid_partial_injection, to check whether a list represents a valid partial injection

• number_of_partial_injections_list, to compute the number of partial injections of a given size.

AUTHOR:

• Pascal WEIL, CNRS, Univ. Bordeaux, LaBRI <pascal.weil@cnrs.fr> (2018-06-09): initial version

```python
stallings_graphs.partial_injections_misc.is_valid_partial_injection(L)
```

Return whether a list represents a PartialInjection.

L is expected to be a list. It properly defines a PartialInjection if its entries are either None or in [0..n − 1], where n is the length of L, and if none of the integer entries is repeated.

INPUT:

• L – List

OUTPUT:

• boolean

EXAMPLES:

```
sage: from stallings_graphs.partial_injections_misc import is_valid_partial_injection
sage: L = [3,1,4,None,2]
sage: is_valid_partial_injection(L)
True

sage: L = [3,1,5,None,None,1]
sage: is_valid_partial_injection(L)
False
```
Warning: This test is performed when a PartialInjection is defined. As a stand-alone function, this is intended to be used when one does not want to attempt to define a PartialInjection if the list is not valid.

stallings_graphs.partial_injections_misc.number_of_partial_injections_list(n)
Return the list of the numbers of partial injections on 0, 1, 2, ..., \( n - 1 \).

The input integer is expected to be positive. A ValueError is raised otherwise.

INPUT:
• \( n \) – integer

OUTPUT:
• a List of length \( n \)

EXAMPLES:

```sage
from stallings_graphs.partial_injections_misc import number_of_partial_injections_list
number_of_partial_injections_list(7)
```

```
[1, 2, 7, 34, 209, 1546, 13327]
```

ALGORITHM:
The algorithm implements a recurrence relation described in [BNW2008].
REFERENCES, INDICES AND TABLES

• genindex
• modindex
• search


S
stallings_graphs.about_automata, 29
stallings_graphs.about_bases, 42
stallings_graphs.about_folding, 39
stallings_graphs.about_free_factors, 48
stallings_graphs.about_Tc_morphisms, 45
stallings_graphs.about_words, 20
stallings_graphs.finitely_generated_subgroup, 5
stallings_graphs.partial_injections, 55
stallings_graphs.partial_injections_misc, 58
INDEX

A

algebraic_extensions() (stallings_graphs.finitely_generated_subgroup.FinitelyGeneratedSubgroup method), 8
alphabetic_inverse() (in module stallings_graphs.about_words), 20
ambient_group_rank() (stallings_graphs.finitely_generated_subgroup.FinitelyGeneratedSubgroup method), 9
are_equal_as_rooted_unlabeled() (in module stallings_graphs.about_automata), 30

B

basis() (stallings_graphs.finitely_generated_subgroup.FinitelyGeneratedSubgroup method), 10
basis_from_spanning_tree() (in module stallings_graphs.about_bases), 43
basis_interpreter() (in module stallings_graphs.about_bases), 43
bouquet() (in module stallings_graphs.about_automata), 31

C

compute_algebraic_extensions() (in module stallings_graphs.about_free_factors), 52
conjugated_by() (stallings_graphs.finitely_generated_subgroup.FinitelyGeneratedSubgroup method), 10
contains_element() (stallings_graphs.finitely_generated_subgroup.FinitelyGeneratedSubgroup method), 11
contains_subgroup() (stallings_graphs.finitely_generated_subgroup.FinitelyGeneratedSubgroup method), 12
cyclic_reduction() (in module stallings_graphs.about_automata), 32
cyclic_reduction_of_a_word() (in module stallings_graphs.about_words), 21

D

DiGraph_to_list_of_PartialInjection() (in module stallings_graphs.about_automata), 30
domain_size() (stallings_graphs.partial_injections.PartialInjection method), 56

E

exchange_labels() (in module stallings_graphs.about_automata), 33

F

FGendomorphism() (in module stallings_graphs.about_TC_morphisms), 46
fibered_product() (in module stallings_graphs.about_automata), 33
FinitelyGeneratedSubgroup (class in stallings_graphs.finitely_generated_subgroup), 6
free_group_reduction() (in module stallings_graphs.about_words), 21
from_digraph() (stallings_graphs.finitely_generated_subgroup.FinitelyGeneratedSubgroup static method), 12
from_generators() (stallings_graphs.finitely_generated_subgroup.FinitelyGeneratedSubgroup static method), 13

G

group_inverse() (in module stallings_graphs.about_words), 22
H
has_index() (stallings_graphs.finitely_generated_subgroup.FinitelyGeneratedSubgroup method), 14

I
image_of_word() (in module stallings_graphs.about_automata), 33
image_of_Word_by_endomorphism() (in module stallings_graphs.about_TC_morphisms), 46
intersection() (stallings_graphs.finitely_generated_subgroup.FinitelyGeneratedSubgroup method), 14
inverse_letter() (in module stallings_graphs.about_words), 22
inverse_partial_injection() (stallings_graphs.partial_injections.PartialInjection method), 56
is_conjugated_to() (stallings_graphs.finitely_generated_subgroup.FinitelyGeneratedSubgroup method), 15
is_cyclically_reduced() (in module stallings_graphs.about_words), 23
is_deterministic() (in module stallings_graphs.about_automata), 35
is_folded() (in module stallings_graphs.about_automata), 35
is_malnormal() (stallings_graphs.finitely_generated_subgroup.FinitelyGeneratedSubgroup method), 15
is_permutation() (stallings_graphs.partial_injections.PartialInjection method), 56
is_reduced() (in module stallings_graphs.about_words), 23
is_valid() (stallings_graphs.finitely_generated_subgroup.FinitelyGeneratedSubgroup method), 16
is_valid_list_of_Words() (in module stallings_graphs.about_words), 24
is_valid_partial_injection() (in module stallings_graphs.partial_injections_misc), 58
is_valid_Word() (in module stallings_graphs.about_words), 23

N
negative_letters() (in module stallings_graphs.about_words), 24
normalize_vertex_names() (in module stallings_graphs.about_automata), 35
NT_data_structures_initialization() (in module stallings_graphs.about_folding), 39
NT_fold() (in module stallings_graphs.about_folding), 40
NT_fold_edge() (in module stallings_graphs.about_folding), 40
NT_initially_unfolded_construction() (in module stallings_graphs.about_folding), 41
NT_is_vertex_unfolded() (in module stallings_graphs.about_folding), 41
number_of_partial_injections_list() (in module stallings_graphs.partial_injections_misc), 59

O
orbit_decomposition() (stallings_graphs.partial_injections.PartialInjection method), 57

P
PartialInjection (class in stallings_graphs.partial_injections), 55
positive_alphabetic_content() (in module stallings_graphs.about_words), 24
positive_letters() (in module stallings_graphs.about_words), 25
positive_value() (in module stallings_graphs.about_words), 25
prepare4visualization_graph() (in module stallings_graphs.about_automata), 36
pruning() (in module stallings_graphs.about_automata), 36

R
random_instance() (stallings_graphs.finitely_generated_subgroup.FinitelyGeneratedSubgroup static method), 17
random_instance() (stallings_graphs.partial_injections.PartialInjection static method), 57
random_letter() (in module stallings_graphs.about_words), 25
random_reduced_word() (in module stallings_graphs.about_words), 26
random_word() (in module stallings_graphs.about_words), 26
rank() (in module stallings_graphs.about_words), 26
rank() (stallings_graphs.finitely_generated_subgroup.FinitelyGeneratedSubgroup method), 18
relabeling() (in module stallings_graphs.about_automata), 37

S
set_of_possible_additional_generators() (in module stallings_graphs.about_free_factors), 53
show_rooted_graph() (in module stallings_graphs.about_automata), 37
show_Stallings_graph() (stallings_graphs.finitely_generated_subgroup.FinitelyGeneratedSubgroup method), 18
SilvaWeil_free_factor_of() (in module stallings_graphs/about_free_factors), 49
SilvaWeil_free_factor_of_ambient() (in module stallings_graphs/about_free_factors), 51
size() (stallings_graphs.partial_injections.PartialInjection method), 58
spanning_tree_and_paths() (in module stallings_graphs/about_bases), 44
stallings_graph() (stallings_graphs.finitely_generated_subgroup.FinitelyGeneratedSubgroup method), 18
stallings_graph_size() (stallings_graphs.finitely_generated_subgroup.FinitelyGeneratedSubgroup method), 19
stallings_graphs.about_automata (module), 29
stallings_graphs.about_bases (module), 42
stallings_graphs.about_folding (module), 39
stallings_graphs.about_free_factors (module), 48
stallings_graphs/about_TC_morphisms (module), 45
stallings_graphs/about_words (module), 20
stallings_graphs.finitely_generated_subgroup (module), 5
stallings_graphs.partial_injections (module), 55
stallings_graphs.partial_injections_misc (module), 58
SW_is_free_factor_of() (stallings_graphs.finitely_generated_subgroup.FinitelyGeneratedSubgroup method), 7
SW_is_free_factor_of_ambient() (stallings_graphs.finitely_generated_subgroup.FinitelyGeneratedSubgroup method), 8
symmetric_alphabet() (in module stallings_graphs/about_words), 27

T
transition_function() (in module stallings_graphs/about_automata), 38
transitions() (in module stallings_graphs/about_automata), 38
translate_alphabetic_Word_to_numeric() (in module stallings_graphs/about_words), 27
translate_character_to_numeric() (in module stallings_graphs/about_words), 28
translate_numeric_to_x0_character() (in module stallings_graphs/about_TC_morphisms), 47
translate_numeric_Word_to_alphabetic() (in module stallings_graphs/about_words), 28
translate_numeric_Word_to_x0_list() (in module stallings_graphs/about_TC_morphisms), 47
translate_x0_character_to_numeric() (in module stallings_graphs/about_TC_morphisms), 47
translate_x0_word_to_numeric_Word() (in module stallings_graphs/about_TC_morphisms), 48
tree_based_interpreter() (in module stallings_graphs/about_bases), 45