stallings_graphs Documentation

Release 0.1

Pascal Weil

Apr 04, 2019

CONTENTS

1 Finitely generated subgroups of free groups **3**
- 1.1 The class `FinitelyGeneratedSubgroup` . . . 3
- 1.2 Ancillary functions about words . . . 15
- 1.3 Ancillary functions about automata . . . 24
- 1.4 The folding algorithm . . . 36
- 1.5 Connecting with the `train_track` package . . . 39

2 Partial injections **43**
- 2.1 The class `PartialInjection` . . . 43
- 2.2 Ancillary functions about partial injections . . . 46

3 Indices and tables **49**

Python Module Index **51**

Index **53**

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.

[BNW02008]

To install this module, you do:

```
sage -pip install http://www.labri.fr/perso/weil/software/stallings_graphs-0.1.tar.gz
```

and eventually:

```
sage -pip install stallings_graphs
```

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.

CHAPTER

ONE

FINITELY GENERATED SUBGROUPS OF FREE GROUPS

1.1 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
- `has_index`, to compute the index of the subgroup
- `basis`
- `intersection`
- `is_malnormal`, to check whether the subgroup is malnormal and, optionally, compute a witness of its non-malnormality
- `contains_element`, to check whether the subgroup contains a given word
- `contains_subgroup`, to check whether the subgroup contains a given 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

EXAMPLES:

```
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

::

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

::

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

::

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>

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 define 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 tuple of partial injections.

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 $a{:}z$ / $A{:}Z$ (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 ($a{:}z$ 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.

ambient_group_rank()

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

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:

```
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.ambient_group_rank()
2

::

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

basis(*alphabet_type='123'*)

Return a basis of this subgroup.

The input semigroup is expected to be an object of the class FinitelyGeneratedSubgroup. The variable alphabet_type determines whether the words in the output are numerical or alphabetic.

INPUT:

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

OUTPUT: A list of objects of the class Word

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.basis()
[word: -3,1,2,-1,-3, word: -2,-1,2,1,-2,-1, word: -1,3,3,-2,-1]

::

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

::

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

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

```
sage: from stallings_graphs import FinitelyGeneratedSubgroup
sage: L = ['ab','ba', 'aBaa']
sage: H = FinitelyGeneratedSubgroup.from_generators(L, alphabet_type = 'abc')
sage: H
A subgroup of the free group of rank 2, whose Stallings graph has 4 vertices
```

```
sage: w = Word('abBA')
sage: K = H.conjugated_by(w, alphabet_type='abc')
sage: H == K
True

sage: w1 = Word('bA')
sage: K1 = H.conjugated_by(w1, alphabet_type='abc')
sage: K1
A subgroup of the free group of rank 2, whose Stallings graph has 4 vertices
```

```
sage: w2 = Word('bAA')
sage: K2 = H.conjugated_by(w2, alphabet_type='abc')
sage: K2
A subgroup of the free group of rank 2, whose Stallings graph has 5 vertices
```

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

```
sage: from stallings_graphs import FinitelyGeneratedSubgroup
sage: L = ['ab','ba', 'aBaa']
sage: H = FinitelyGeneratedSubgroup.from_generators(L, alphabet_type = 'abc')
sage: w = Word([1,-2,-2])
sage: H.contains_element(w)
False
```

```
sage: w = Word('abba')
sage: H.contains_element(w, alphabet_type = 'abc')
True
```

```
sage: w = Word()
sage: H.contains_element(w)
True
```

```
sage: H = FinitelyGeneratedSubgroup([])
sage: w = Word()
sage: H.contains_element(w)
True
```

```
sage: w = Word([1,2,1])
sage: H.contains_element(w)
False
```

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: w = Word('abba')
sage: K = FinitelyGeneratedSubgroup.from_generators(['abb'], alphabet_type = 'abc')
sage: H.contains_subgroup(K)
False
```

```
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')
sage: H = FinitelyGeneratedSubgroup.from_digraph(G)
sage: H
A subgroup of the free group of rank 3, whose Stallings graph has 14 vertices

::

sage: V = [0]
sage: E = []
sage: G = DiGraph([V,E], format='vertices_and_edges', loops=True, multiedges=True)
sage: H = FinitelyGeneratedSubgroup.from_digraph(G)
sage: H
A subgroup of the free group of rank 0, whose Stallings graph has 1 vertices

::

sage: V = [0]
sage: E = [(0,0,1)]
sage: G = DiGraph([V,E], format='vertices_and_edges', loops=True, multiedges=True)
sage: H = FinitelyGeneratedSubgroup.from_digraph(G)
sage: H
A subgroup of the free group of rank 1, whose Stallings graph has 1 vertices

::

sage: V = [0]
sage: E = [(0,0,3)]
sage: G = DiGraph([V,E], format='vertices_and_edges', loops=True, multiedges=True)
sage: H = FinitelyGeneratedSubgroup.from_digraph(G)
sage: H
A subgroup of the free group of rank 3, whose Stallings graph has 1 vertices
```

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 (on alphabet $[1..r]$ and inverse letters $[-r..-1]$) or alphabetical (a:z / A:Z). 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:

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

::

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

::

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. `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

```
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', 'aa']
sage: K = FinitelyGeneratedSubgroup.from_generators(M, alphabet_type = 'abc')
sage: S = H.intersection(K)
sage: S.basis()
[word: 2121, word: 2,1,-2,1,-2,-2, word: 12, word: -1,2,-1,-1]
```

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 $w^{-1}Hw = K$.

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 = ['abCA', 'abbaBA', 'aCacA', 'abbbcA']
sage: H = FinitelyGeneratedSubgroup.from_generators(generators, alphabet_type='abc')
sage: other_gens = ['ba', 'bbcb', 'bbcc', 'bbaBB']
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')
```

(continues on next page)

(continued from previous page)

```
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 $t^{-1}Ht$.

INPUT:

- `self` – `FinitelyGeneratedSubgroup`
- `alphabet_type` – a string which is either `'abc'` or `'123'`
- `witness`– a boolean

OUTPUT: if `witness` is set to `False`

- a boolean

if `witness` is set to `True`

- a tuple of the form `(True, None, None)` if this subgroup is malnormal, and of the form `(False, s,t)` if it is not, where s and t are of the class `Word`.

EXAMPLES

```
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: 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', 'aBababA']
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) connnected components starting with the smaller ones.

is_valid(*verbose=False*)

Return whether this 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 – FinitelyGeneratedSubgroup
- verbose – boolean

OUTPUT:

- boolean

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.is_valid()
True

::

sage: M = [PartialInjection([1,2,None,4,5,3]), PartialInjection([0,2,1,None,4,3])]
sage: K = FinitelyGeneratedSubgroup(M)
sage: K.is_valid()
False
```

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: if `verbose = False`

- an object of the class `FinitelyGeneratedSubgroup`

otherwise

- a tuple of an object of the class `FinitelyGeneratedSubgroup` and an integer

EXAMPLES:

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

::

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

::

sage: H,c = FinitelyGeneratedSubgroup.random_instance(12,3,verbose=True)
sage: H
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] F. Bassino, C. Nicaud, P. Weil. Random generation of finitely generated subgroups of a free group, International Journal of Algebra and Computation 18 (2008) 1-31.

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: if visu_tool is set to 'plot'

- a visualization of the Stallings graph using graph.plot

if visu_tool is set to 'tikz'

- a visualization of the Stallings graph using TikzPicture. In this 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:

```
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

::

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

```
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

::

sage: L = []
sage: H = FinitelyGeneratedSubgroup.from_generators(L)
sage: H.stallings_graph_size()
1
```

1.2 Ancillary functions about words

The methods for the class FinitelyGeneratedSubgroup use a number of ancillary functions. These are the functions which deal with words (actually, objects of class Word) 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 (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).

We have functions to:

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

EXAMPLES:

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

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

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.

`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: from stallings_graphs.about_words import alphabetic_inverse
sage: alphabetic_inverse('b')
'B'
sage: alphabetic_inverse('D')
'd'
```

`stallings_graphs.about_words.`**`cyclic_reduction_of_a_word`**`(`u`)`

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

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

INPUT:

- `w` – `Word`
- `check` – boolean

OUTPUT:

- pair of objects of class `Word`

EXAMPLES

```
sage: 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: )
```

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

Return the reduced word that is equivalent to this word.

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

INPUT:

- `w` – `Word`
- `check` – boolean

OUTPUT:

- `Word`

EXAMPLES

```
sage: from stallings_graphs.about_words import free_group_reduction
sage: w = Word([3,1,-2,-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 `Word` on a numerical or letter alphabet, depending on the value of `alphabet_type`. Its inverse is obtained in reading w in reverse order and replacing each letter by its inverse. If `check` is set to `True`, `is_valid_Word` is run on w.

INPUT:

- `w` – `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
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

```
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

::

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

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

::

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 $a{:}z + A{:}Z$ if `alphabet_type='abc'`.

INPUT:

- `w` – `Word`
- `alphabet_type` – string, which must be either `'abc'` or `'123'`

OUTPUT:

- boolean

EXAMPLES:

```
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

::
```

(continues on next page)

(continued from previous page)

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

::

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

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

Return whether a list of Word``s 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
- alphabet_type – string, which must be either 'abc' or '123'

OUTPUT:

- boolean

EXAMPLES:

```
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
```

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:

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

stallings_graphs.about_words.**positive_letters**(r)

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

r is expected to be positive.

INPUT:

- r – integer

OUTPUT:

- the list range(r)

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`.

w is expected to be a `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`, `is_valid_Word(w,alphabet_type='123')` is run.

INPUT:

- `w` – `Word`
- `alphabet_type` – string, which must be either `'abc'` or `'123'`

OUTPUT:

- integer

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

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:

```
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]
```

stallings_graphs.about_words.**translate_alphabetic_Word_to_numeric**(w)

Return the numerical equivalent of a `Word` of `alphabet_type = 'abc'`.

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

INPUT:

- `w` – Word

OUTPUT:

- Word

EXAMPLES:

```
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
```

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 $\{\pm 1, \ldots, \pm 26\}$. 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'
sage: translate_numeric_to_character(-10)
'J'
```

1.3 Ancillary functions about automata

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 (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 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`
- compute a spanning tree
- compute a basis specified by a spanning tree
- 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 = ['aBABBaaaab', 'BBAbbABABA', '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 (2018-06-09): initial version CNRS, Univ. Bordeaux, LaBRI <pascal.weil@cnrs.fr>

stallings_graphs.about_automata.**DiGraph_to_list_of_PartialInjection**(*G*)

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 $[0..n-1]$ (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 $1, 2, \ldots, r$, respectively.

INPUT:

- G – DiGraph

OUTPUT:

- List of lists

EXAMPLES

```
sage: from stallings_graphs import PartialInjection
sage: from stallings_graphs.about_automata import bouquet, DiGraph_to_list_of_PartialInjection
sage: from stallings_graphs.about_folding import NT_fold
sage: from stallings_graphs import PartialInjection
sage: L = [[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]
```

stallings_graphs.about_automata.**are_equal_as_rooted_unlabeled**(*G*, *H*, *certificate=False*, *verbose=False*)

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
- certificate – boolean
- verbose – boolean

OUTPUT: If certificate is set to False:

- a boolean

If certificate is set to True:

- a tuple of the form (False,None) or (True,sigma) where sigma is the conjugating permutation

EXAMPLES

```
sage: from stallings_graphs import FinitelyGeneratedSubgroup, PartialInjection
sage: from stallings_graphs.about_automata import are_equal_as_rooted_unlabeled
sage: L1 = [PartialInjection([1,2,None,4,5,3]), PartialInjection([0,3,4,None,None,None])]
sage: H1 = FinitelyGeneratedSubgroup(L1)
sage: G1 = H1.stallings_graph()
sage: L2 = [PartialInjection([1,2,None,5,4,3]), PartialInjection([0,3,5,None,None,None])]
sage: H2 = FinitelyGeneratedSubgroup(L2)
sage: G2 = H2.stallings_graph()
sage: L3 = [PartialInjection([1,2,None,4,5,3]), PartialInjection([0,5,3,None,None,None])]
sage: H3 = FinitelyGeneratedSubgroup(L3)
sage: G3 = H3.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)
sage: (b,tau)
(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
```

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.

stallings_graphs.about_automata.**basis_from_spanning_tree**(*G*, *T*, *D*, *root=0*)

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 (numerical)

EXAMPLES:

```
sage: from stallings_graphs.about_automata import bouquet, spanning_tree_and_paths, basis_from_spanning_
↪tree
sage: from stallings_graphs.about_folding import NT_fold
sage: generators = [[3,1,-2,-1,3],[1,2,-1,-2,1,2],[1,2,-3,-3,1]]
sage: G = NT_fold(bouquet(generators))
sage: T,L,D = spanning_tree_and_paths(G)
sage: basis_from_spanning_tree(G,T,D)
[word: -3,1,2,-1,-3, word: -2,-1,2,1,-2,-1, word: -1,3,3,-2,-1]

::

sage: basis_from_spanning_tree(T,T,D)
[]
```

stallings_graphs.about_automata.**bouquet**(*list_of_Words*, *alphabet_type='123'*, *check=False*)

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 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:

```
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
```

stallings_graphs.about_automata.**cyclic_reduction**(*G*, *trace=False*)

Return the cyclic reduction of this DiGraph.

G is expected to be DiGraph with numerical edge labels and vertex set of the form $[0..n]$. The cyclic reduction is obtained by iteratively deleting vertices of degree 1 (including the base vertex – that is the difference with the prune method). Its vertex set is normalized to be of the form $[0..m]$. Note that the vertex labeled 0 in the cyclic reduction need not be the same as in the G (but it will be the same if the base vertex of G belongs to the cyclic reduction). 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: if trace is set to False:

- Digraph

else:

- a pair (GG, w) of a DiGraph and a Word.

EXAMPLES:

```
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

::

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 $0, \ldots, n-1$, 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:

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

stallings_graphs.about_automata.**express_Word_in_basis_from_spanning_tree**(*G*, *T*, *w*, *root=0*, *alphabet_type='abc'*)

Return the valuea word 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. w is expected to be a Word labeling a loop at root in 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

EXAMPLES:

```
sage: from stallings_graphs.about_automata import bouquet, spanning_tree_and_paths, basis_from_spanning_
↪tree, express_Word_in_basis_from_spanning_tree
sage: from stallings_graphs.about_folding import NT_fold
sage: generators = ['abaa','ababb','abababb']
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: 1211, word: -1,2,2, word: -2,1,2]

::

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

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

Compute the fibered product (aka 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, u_2) 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

```
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], format='vertices_and_edges', 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], format='vertices_and_edges', 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: if trace=False

- integer or None

if trace=True, a triple consisting of

- an integer or None
- an integer
- an integer

EXAMPLES

```
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()
```

(continues on next page)

(continued from previous page)

```
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)
```

```
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 $range(n)$. 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 $range(n)$. 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; 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:

```
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)
sage: GG = normalize_vertex_names(G)
sage: GG.vertices()
[0, 1, 2, 3, 4]
```

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 $a_1, ..., a_r$ (alphabet_type='123') or as $a, b, c, ..., z$ (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:

```
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:

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

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

OUTPUT:

- DiGraph

EXAMPLES

```
sage: from stallings_graphs.about_automata import bouquet, pruning
sage: from stallings_graphs.about_folding import NT_fold
sage: L = [[2,1,-2,2,1,-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 a DiGraph with vertices labeled $0, \ldots, n-1$. R is expected to be a permutation of $0, \ldots, n-1$. (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), (0,1,2), (0,4,3), (1,0,2), (1,2,1), (1,2,3), (2,3,2), (2,3,3),␣
↪(4,3,1)]], 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.**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 –, 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
- 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_automata 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,
 10: word: -1,3,
 11: word: -1}
```

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 labels 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, 0, None, None, 0],
 2: [None, None, None, 2, None, 6, None, None, 7, 0, 11, None, None, None],
 3: [1, None, None, None, 0, 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 $range(n)$. The output dictionary maps each edge label to a List of sets: the image of edge label x is a list indexed by the vertex set, where the v-entry is the set of end vertices of x-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]}
```

1.4 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 an article by Nicholas Touikan, Intern. J. Algebra and Computation, vol 16, 2006, pages 1031–1045], 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:

```
sage: from stallings_graphs.about_words import random_reduced_word
sage: L = ['aBABBaaaab', 'BBAbbABABA', '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 $[0..n-1]$ 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 his folding algorithm.

INPUT:

- `digr` – `DiGraph`

OUTPUT:

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

EXAMPLES

```
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]]
```

(continues on next page)

(continued from previous page)

```
sage: G = bouquet(L)
sage: NT_data_structures_initialization(G)
({{0}, {10}, {11}, {12}, {13}, {1}, {2}, {3}, {4}, {5}, {6}, {7}, {8}, {9}},
 {1: {0: [{13}, {5, 10}],
   1: [set(), {2}],
   2: [{1}, set()],
   3: [{4}, set()],
   4: [set(), {3}],
   5: [{0}, set()],
   6: [{7}, set()],
   7: [set(), {6}],
   8: [set(), {9}],
   9: [{8}, set()],
   10: [{0}, set()],
   11: [set(), set()],
   12: [set(), set()],
   13: [set(), {0}]},
  2: {0: [{9}, set()],
   1: [set(), set()],
   2: [{3}, set()],
   3: [set(), {2}],
   4: [set(), set()],
   5: [set(), {6}],
   6: [{5}, set()],
   7: [{8}, set()],
   8: [set(), {7}],
   9: [set(), {0}],
   10: [set(), {11}],
   11: [{10}, set()],
   12: [set(), set()],
   13: [set(), set()]},
  3: {0: [{4}, {1}],
   1: [{0}, set()],
   2: [set(), set()],
   3: [set(), set()],
   4: [set(), {0}],
   5: [set(), set()],
   6: [set(), set()],
   7: [set(), set()],
   8: [set(), set()],
   9: [set(), set()],
   10: [set(), set()],
   11: [{12}, set()],
   12: [{13}, {11}],
   13: [set(), {12}]}})
```

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 $[0..n]$: this is not reflecting the name of vertices in the original DiGraph – except for the base vertex.

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
```

(continues on next page)

(continued from previous page)

```
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][u][0] (outgoing edges) or both in NT_edge_structure[a][u][1] (incoming edges). The method returns updated versions of NT_vertices, NT_edge_structure, NT_unfolded after the a-labeled edges out of u and into v1 and v2 (resp. into u out of v1 and v2) are merged.

INPUT:

- NT_vertices – *DisjointSet*
- NT_edge_structure – dictionary of dictionaries
- NT_unfolded – set
- u – element
- v1 – element
- v2 – element

OUTPUT:

- the input objects NT_vertices, NT_edge_structure and NT_unfolded are modified in place

EXAMPLES

```
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]]
sage: G = bouquet(L)
sage: NT_vertices,NT_edge_structure = NT_data_structures_initialization(G)
sage: NT_unfolded = set([0])
sage: NT_fold_edge(NT_vertices,NT_edge_structure,NT_unfolded,0,5,10)
```

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, NT_vertices is a DisjointSet structure based on the vertices of digr and NT_edge_structure is a dictionary based on the edges of digr. This method is meant to be used once, when the input defining a subgroup is a NFA with one initial-final state. (If the DiGraph is a bouquet of freely reduced words, then NT_initially_unfolded could be set immediately to set([0]).)

INPUT:

- digr – DiGraph
- NT_vertices – DisjointSet

- NT_edge_structure – dictionary of dictionaries

OUTPUT:

- set

EXAMPLES

```
sage: from stallings_graphs.about_automata import bouquet
sage: from stallings_graphs.about_folding import NT_data_structures_initialization, NT_initially_
↪unfolded_construction
sage: L = [[3,1,-2,-1,3],[1,2,-1,-2,1,2],[1,2,-3,-3,1]]
sage: G = bouquet(L)
sage: NT_vertices,NT_edge_structure = NT_data_structures_initialization(G)
sage: NT_initially_unfolded_construction(G,NT_vertices,NT_edge_structure)
{0}
```

stallings_graphs.about_folding.**NT_is_vertex_unfolded**(*v*, *NT_vertices*, *NT_edge_structure*)

Return whether this vertex is unfolded in the given DisjointSet structure.

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

```
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)
sage: NT_vertices,NT_edge_structure = NT_data_structures_initialization(G)
sage: NT_is_vertex_unfolded(0,NT_vertices,NT_edge_structure)
True

::

sage: NT_is_vertex_unfolded(2,NT_vertices,NT_edge_structure)
False
```

1.5 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 ambient 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

EXAMPLES:

```
sage: from stallings_graphs.about_TC_morphisms import translate_numeric_Word_to_x0_word
sage: translate_numeric_Word_to_x0_word([7,1,-2,3,-3])
['x6', 'x0', 'X1', 'x2', 'X2']

sage: from stallings_graphs.about_TC_morphisms import translate_x0_word_to_numeric_Word
sage: translate_x0_word_to_numeric_Word(['x1','X0','x2','X1'])
word: 2,-1,3,-2
```

AUTHOR:

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

stallings_graphs.about_TC_morphisms.**translate_numeric_Word_to_x0_word**(*w*)

Return the corresponding word in Thierry Coulbois's `x0` format.

`w` is expected to be a `Word` on a numerical alphabet.

INPUT:

- `w` – Word

OUTPUT:

- list

EXAMPLES:

```
sage: from stallings_graphs.about_TC_morphisms import translate_numeric_Word_to_x0_word
sage: translate_numeric_Word_to_x0_word([7,1,-2,3,-3])
['x6', 'x0', 'X1', 'x2', 'X2']
```

stallings_graphs.about_TC_morphisms.**translate_numeric_to_x0_character**(*i*)

Return the corresponding character in Thierry Coulbois's `x0` format.

`i` is expected to be a non-zero integer. An exception is raised if that is not the case.

INPUT:

- `i` – integer

OUTPUT:

- string

EXAMPLES:

```
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'
```

stallings_graphs.about_TC_morphisms.**translate_x0_character_to_numeric**(*letter*)

Return the corresponding numeric.

letter is expected to be a string of the form xj or Xj, where j is a non-negative integer in decimal expansion.

INPUT:

- letter – string

OUTPUT:

- integer

EXAMPLES:

```
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 list of strings of the form xj or Xj, where j is a non-negative integer in decimal expansion.

INPUT:

- u – list

OUTPUT:

- Word

EXAMPLES:

```
sage: from stallings_graphs.about_TC_morphisms import translate_x0_word_to_numeric_Word
sage: translate_x0_word_to_numeric_Word(['x1','X0','x2','X1'])
word: 2,-1,3,-2

::

sage: translate_x0_word_to_numeric_Word([])
word:
```

CHAPTER

TWO

PARTIAL INJECTIONS

2.1 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:

```
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

::

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

AUTHOR:

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

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 $range(n)$ without any repetition, or `None` (the list of images of the elements of `range(n)`). 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

```
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 $range(n)$ with an image, that is, $n - \ell$, where ℓ is the number of None.

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.domain_size()
4
```

inverse_partial_injection()

Return the inverse of a PartialInjection.

INPUT:

- self – PartialInjection

OUTPUT:

- a PartialInjection

EXAMPLES:

```
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`

OUTPUT:

- boolean

EXAMPLES:

```
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

::

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 Lists

EXAMPLES:

```
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()
([[8, 9, 1], [4, 11, 10, 7, 3, 0, 6]], [[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: if `statistics = False`:

- an object of the class `PartialInjection`

otherwise:

- a pair of an integer and an object of class `PartialInjection`

EXAMPLES:

```
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]

::

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

ALGORITHM:

Tha algorithm implemented here is that in [Bassino, Nicaud, Weil. Random generation of finitely generated subgroups of a free group, International Journal of Algebra and Computation 18 (2008) 1-31]. 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
```

2.2 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

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 $range(n)$, 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
sage: number_of_partial_injections_list(7)
[1, 2, 7, 34, 209, 1546, 13327]
```

ALGORITHM:

The algorithm implements a recurrence relation described in [BNW2008] F. Bassino, C. Nicaud, P. Weil. Random generation of finitely generated subgroups of a free group, International Journal of Algebra and Computation 18 (2008) 1-31.

CHAPTER

THREE

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

S

stallings_graphs.about_automata, 24
stallings_graphs.about_folding, 36
stallings_graphs.about_TC_morphisms, 39
stallings_graphs.about_words, 15
stallings_graphs.finitely_generated_subgroup, 3
stallings_graphs.partial_injections, 43
stallings_graphs.partial_injections_misc, 46

INDEX

A

alphabetic_inverse() (in module stallings_graphs.about_words), 16
ambient_group_rank() (stallings_graphs.finitely_generated_subgroup.FinitelyGeneratedSubgroup method), 5
are_equal_as_rooted_unlabeled() (in module stallings_graphs.about_automata), 25

B

basis() (stallings_graphs.finitely_generated_subgroup.FinitelyGeneratedSubgroup method), 5
basis_from_spanning_tree() (in module stallings_graphs.about_automata), 27
bouquet() (in module stallings_graphs.about_automata), 27

C

conjugated_by() (stallings_graphs.finitely_generated_subgroup.FinitelyGeneratedSubgroup method), 5
contains_element() (stallings_graphs.finitely_generated_subgroup.FinitelyGeneratedSubgroup method), 6
contains_subgroup() (stallings_graphs.finitely_generated_subgroup.FinitelyGeneratedSubgroup method), 7
cyclic_reduction() (in module stallings_graphs.about_automata), 28
cyclic_reduction_of_a_word() (in module stallings_graphs.about_words), 16

D

DiGraph_to_list_of_PartialInjection() (in module stallings_graphs.about_automata), 25
domain_size() (stallings_graphs.partial_injections.PartialInjection method), 44

E

exchange_labels() (in module stallings_graphs.about_automata), 28
express_Word_in_basis_from_spanning_tree() (in module stallings_graphs.about_automata), 29

F

fibered_product() (in module stallings_graphs.about_automata), 29
FinitelyGeneratedSubgroup (class in stallings_graphs.finitely_generated_subgroup), 4
free_group_reduction() (in module stallings_graphs.about_words), 17
from_digraph() (stallings_graphs.finitely_generated_subgroup.FinitelyGeneratedSubgroup static method), 7
from_generators() (stallings_graphs.finitely_generated_subgroup.FinitelyGeneratedSubgroup static method), 8

G

group_inverse() (in module stallings_graphs.about_words), 17

H

has_index() (stallings_graphs.finitely_generated_subgroup.FinitelyGeneratedSubgroup method), 9

I

image_of_word() (in module stallings_graphs.about_automata), 30
intersection() (stallings_graphs.finitely_generated_subgroup.FinitelyGeneratedSubgroup method), 9
inverse_letter() (in module stallings_graphs.about_words), 18
inverse_partial_injection() (stallings_graphs.partial_injections.PartialInjection method), 44
is_conjugated_to() (stallings_graphs.finitely_generated_subgroup.FinitelyGeneratedSubgroup method), 10
is_cyclically_reduced() (in module stallings_graphs.about_words), 18
is_deterministic() (in module stallings_graphs.about_automata), 31
is_folded() (in module stallings_graphs.about_automata), 31
is_malnormal() (stallings_graphs.finitely_generated_subgroup.FinitelyGeneratedSubgroup method), 11
is_permutation() (stallings_graphs.partial_injections.PartialInjection method), 44
is_reduced() (in module stallings_graphs.about_words), 19
is_valid() (stallings_graphs.finitely_generated_subgroup.FinitelyGeneratedSubgroup method), 12
is_valid_list_of_Words() (in module stallings_graphs.about_words), 20
is_valid_partial_injection() (in module stallings_graphs.partial_injections_misc), 46
is_valid_Word() (in module stallings_graphs.about_words), 19

N

negative_letters() (in module stallings_graphs.about_words), 20
normalize_vertex_names() (in module stallings_graphs.about_automata), 32
NT_data_structures_initialization() (in module stallings_graphs.about_folding), 36
NT_fold() (in module stallings_graphs.about_folding), 37
NT_fold_edge() (in module stallings_graphs.about_folding), 38
NT_initially_unfolded_construction() (in module stallings_graphs.about_folding), 38
NT_is_vertex_unfolded() (in module stallings_graphs.about_folding), 39
number_of_partial_injections_list() (in module stallings_graphs.partial_injections_misc), 47

O

orbit_decomposition() (stallings_graphs.partial_injections.PartialInjection method), 45

P

PartialInjection (class in stallings_graphs.partial_injections), 43
positive_letters() (in module stallings_graphs.about_words), 20
positive_value() (in module stallings_graphs.about_words), 21
prepare4visualization_graph() (in module stallings_graphs.about_automata), 32
pruning() (in module stallings_graphs.about_automata), 33

R

random_instance() (stallings_graphs.finitely_generated_subgroup.FinitelyGeneratedSubgroup static method), 12
random_instance() (stallings_graphs.partial_injections.PartialInjection static method), 45
random_letter() (in module stallings_graphs.about_words), 21
random_reduced_word() (in module stallings_graphs.about_words), 21
random_word() (in module stallings_graphs.about_words), 22
rank() (in module stallings_graphs.about_words), 22
rank() (stallings_graphs.finitely_generated_subgroup.FinitelyGeneratedSubgroup method), 13
relabeling() (in module stallings_graphs.about_automata), 33

S

show_rooted_graph() (in module stallings_graphs.about_automata), 34

show_Stallings_graph() (stallings_graphs.finitely_generated_subgroup.FinitelyGeneratedSubgroup method), 14
size() (stallings_graphs.partial_injections.PartialInjection method), 46
spanning_tree_and_paths() (in module stallings_graphs.about_automata), 34
stallings_graph() (stallings_graphs.finitely_generated_subgroup.FinitelyGeneratedSubgroup method), 14
stallings_graph_size() (stallings_graphs.finitely_generated_subgroup.FinitelyGeneratedSubgroup method), 15
stallings_graphs.about_automata (module), 24
stallings_graphs.about_folding (module), 36
stallings_graphs.about_TC_morphisms (module), 39
stallings_graphs.about_words (module), 15
stallings_graphs.finitely_generated_subgroup (module), 3
stallings_graphs.partial_injections (module), 43
stallings_graphs.partial_injections_misc (module), 46
symmetric_alphabet() (in module stallings_graphs.about_words), 22

T

transition_function() (in module stallings_graphs.about_automata), 35
transitions() (in module stallings_graphs.about_automata), 35
translate_alphabetic_Word_to_numeric() (in module stallings_graphs.about_words), 22
translate_character_to_numeric() (in module stallings_graphs.about_words), 23
translate_numeric_to_character() (in module stallings_graphs.about_words), 23
translate_numeric_to_x0_character() (in module stallings_graphs.about_TC_morphisms), 40
translate_numeric_Word_to_alphabetic() (in module stallings_graphs.about_words), 23
translate_numeric_Word_to_x0_word() (in module stallings_graphs.about_TC_morphisms), 40
translate_x0_character_to_numeric() (in module stallings_graphs.about_TC_morphisms), 40
translate_x0_word_to_numeric_Word() (in module stallings_graphs.about_TC_morphisms), 41