GrinPy Documentation

Release 0.1

David Amos, Randy Davila

Dec 09, 2017

Contents:

GrinPy is a NetworkX extension for calculating graph invariants. This extension imports all of NetworkX into the same interface as GrinPy for easy of use and provides the following extensions:

- extended functional interface for graph properties
- calculation of NP-hard invariants such as: independence number, domination number and zero forcing number
- calculation of several invariants that are known to be related to the NP-hard invariants, such as the residue, the annihilation number and the sub-domination number

Our goal is to provide the most comprehensive list of invariants. We will be continuing to add to this list as time goes on, and we invite others to join us by contributing their own implementations of algorithms for computing new or existing GrinPy invariants.

CHAPTER 1

Audience

We envision GrinPy's primary audience to be professional mathematicians and students of mathematics. Computer scientists, electrical engineers, physicists, biologists, chemists and social scientists may also find GrinPy's extensions to the standard NetworkX package useful.

CHAPTER 2

History

Grinpy was originally created to aid the developers, David Amos and Randy Davila, in creating an ordered tree of graph databases for use in an experimental automated conjecturing program. It quickly became clear that a Python package for calculating graph invariants would be useful. GrinPy was created in November 2017 and is still in its infancy. We look forward to what the future brings!

CHAPTER 3

Free Software

GrinPy is free software; you can redistribute it and/or modify it under the terms of the *3-clause BSD license*, the same license that NetworkX is released under. We greatly appreciate contributions. Please join us on Github.

3.1 Tutorial

This guide can help you start working with GrinPy. We assume basic knowledge of NetworkX. For more information on how to use NetworkX, see the NetworkX Documentation.

3.1.1 Calculating the Independence Number

For this example we will create a cycle of order 5.

```
>>> import grinpy as gp
>>> G = gp.cycle_graph(5)
```

In order to compute the independence number of the cycle, we simply call the *independence_number* method on the graph:

```
>>> gp.independence_number(G)
2
```

It's that simple!

Note: In this release (version 0.1), all methods are defined only for simple graphs. In future releases, we will expand to digraphs and multigraphs.

3.1.2 Get a Maximum Independent Set

If we are interested in finding a maximum independent set in the graph:

```
>>> gp.max_independent_set(G)
[0, 2]
```

3.1.3 Determine if a Given Set is Independent

We may check whether or not a given set is independent:

```
>>> gp.is_independent_set(G, [0, 1])
False
>>> gp.is_independent_set(G, [1, 3])
True
```

3.1.4 General Notes

The vast majority of NP-hard invariants will include three methods corresponding to the above examples. That is, for each invariant, there will be three methods:

- Calculate the invariant
- Get a set of nodes realizing the invariant
- Determine whether or not a given set of nodes meets some necessary condition for the invariant.

3.2 Reference

Release 0.1

Date Dec 09, 2017

3.2.1 Classes

Release 0.1

Date Dec 09, 2017

HavelHakimi

Overview

class grinpy.**HavelHakimi**(*sequence*)

Class for performing and keeping track of the Havel Hakimi process on a sequence of positive integers.

sequence [input sequence] The sequence of integers to initialize the Havel Hakimi process.

Methods

HavelHakimi.__init__(sequence)	
HavelHakimi.depth()	Return the depth of the Havel Hakimi process.
HavelHakimi.get_elimination_sequence()	Return the elimination sequence of the Havel Hakimi process.
HavelHakimi.get_initial_sequence()	Return the initial sequence passed to the Havel Hakimi class for initialization.
HavelHakimi.is_graphic()	Return whether or not the initial sequence is graphic.
HavelHakimi.get_process()	Return the list of sequence produced during the Havel Hakimi process.
HavelHakimi.residue()	Return the residue of the sequence.

grinpy.HavelHakimi.__init__

HavelHakimi.**__init__**(*sequence*)

grinpy.HavelHakimi.depth

HavelHakimi.**depth**()

Return the depth of the Havel Hakimi process.

depth [int] The depth of the Havel Hakimi process.

grinpy.HavelHakimi.get_elimination_sequence

HavelHakimi.**get_elimination_sequence**()

Return the elimination sequence of the Havel Hakimi process.

elimSequence [list] The elimination sequence of the Havel Hakimi process.

grinpy.HavelHakimi.get_initial_sequence

HavelHakimi.**get_initial_sequence**()

Return the initial sequence passed to the Havel Hakimi class for initialization.

initSequence [list] The initial sequence passed to the Havel Hakimi class.

grinpy.HavelHakimi.is_graphic

HavelHakimi.**is_graphic**()

Return whether or not the initial sequence is graphic.

isGraphic [bool] True if the initial sequence is graphic. False otherwise.

grinpy.HavelHakimi.get_process

HavelHakimi.**get_process**()

Return the list of sequence produced during the Havel Hakimi process. The first element in the list is the initial sequence.

process [list] The list of sequences produced by the Havel Hakimi process.

grinpy.HavelHakimi.residue

`HavelHakimi.residue()`

Return the residue of the sequence.

residue [int] The residue of the initial sequence. If the sequence is not graphic, this will be None.

3.2.2 Functions

Release 0.1

Date Dec 09, 2017

Degree

Assorted degree related graph utilities.

degree_sequence(G)	Return the degree sequence of G.
min_degree(G)	Return the minimum degree of G.
max_degree(G)	Return the maximum degree of G.
average_degree(G)	Return the average degree of G.
number_of_nodes_of_degree_k(G, k)	Return the number of nodes of the graph with degree equal to k.
number_of_degree_one_nodes(G)	Return the number of nodes of the graph with degree equal to 1.
number_of_min_degree_nodes(G)	Return the number of nodes of the graph with degree equal to the minimum degree of the graph.
number_of_max_degree_nodes(G)	Return the number of nodes of the graph with degree equal to the maximum degree of the graph.
neighborhood_degree_list(G, nbunch)	Return a list of the unique degrees of all neighbors of nodes in nbunch
closed_neighborhood_degree_list(G, nbunch)	Return a list of the unique degrees of all nodes in the closed neighborhood of the nodes in nbunch.

grinpy.functions.degree.degree_sequence

`grinpy.functions.degree.degree_sequence(G)`

Return the degree sequence of G.

The degree sequence of a graph is the sequence of degrees of the nodes in the graph.

G [graph] A NetworkX graph.

degSeq [list] The degree sequence of the graph.

```
>>> G = nx.path_graph(3) # Path on 3 nodes
>>> nx.degree_sequence(G)
[2, 1, 1]
```

grinpy.functions.degree.min_degree

grinpy.functions.degree.**min_degree**(*G*)

Return the minimum degree of G.

The minimum degree of a graph is the smallest degree of any node in the graph.

G [graph] A NetworkX graph.

minDegree [int] The minimum degree of the graph.

```
>>> G = nx.path_graph(3) # Path on 3 nodes
>>> nx.min_degree(G)
1
```

grinpy.functions.degree.max_degree

grinpy.functions.degree.**max_degree**(*G*)

Return the maximum degree of G.

The maximum degree of a graph is the largest degree of any node in the graph.

G [graph] A NetworkX graph.

maxDegree [int] The maximum degree of the graph.

```
>>> G = nx.path_graph(3) # Path on 3 nodes
>>> nx.min_degree(G)
2
```

grinpy.functions.degree.average_degree

grinpy.functions.degree.**average_degree**(*G*)

Return the average degree of G.

The average degree of a graph is the average of the degrees of all nodes in the graph.

G [graph] A NetworkX graph.

avgDegree [float] The average degree of the graph.

```
>>> G = nx.star_graph(3) # Star on 4 nodes
>>> nx.average_degree(G)
1.5
```

grinpy.functions.degree.number_of_nodes_of_degree_k

grinpy.functions.degree.**number_of_nodes_of_degree_k**(*G*, *k*)

Return the number of nodes of the graph with degree equal to k.

G [graph] A NetworkX graph.

k [int] A positive integer.

numNodes [int] The number of nodes in the graph with degree equal to k.

number_of_leaves, number_of_min_degree_nodes, number_of_max_degree_nodes

```
>>> G = nx.path_graph(3) # Path on 3 nodes
>>> nx.number_of_nodes_of_degree_k(G, 1)
2
```

grinpy.functions.degree.number_of_degree_one_nodes

`grinpy.functions.degree.`**`number_of_degree_one_nodes`**`(G)`

Return the number of nodes of the graph with degree equal to 1.

A vertex with degree equal to 1 is also called a *leaf*.

G [graph] A NetworkX graph.

numNodes [int] The number of nodes in the graph with degree equal to 1.

number_of_nodes_of_degree_k, number_of_min_degree_nodes, number_of_max_degree_nodes

```
>>> G = nx.path_graph(3) # Path on 3 nodes
>>> nx.number_of_leaves(G)
2
```

grinpy.functions.degree.number_of_min_degree_nodes

`grinpy.functions.degree.`**`number_of_min_degree_nodes`**`(G)`

Return the number of nodes of the graph with degree equal to the minimum degree of the graph.

G [graph] A NetworkX graph.

numNodes [int] The number of nodes in the graph with degree equal to the minimum degree.

number_of_nodes_of_degree_k, number_of_leaves, number_of_max_degree_nodes, min_degree

```
>>> G = nx.path_graph(3) # Path on 3 nodes
>>> nx.number_of_min_degree_nodes(G)
2
```

grinpy.functions.degree.number_of_max_degree_nodes

`grinpy.functions.degree.`**`number_of_max_degree_nodes`**`(G)`

Return the number of nodes of the graph with degree equal to the maximum degree of the graph.

G [graph] A NetworkX graph.

numNodes [int] The number of nodes in the graph with degree equal to the maximum degree.

number_of_nodes_of_degree_k, number_of_leaves, number_of_min_degree_nodes, max_degree

```
>>> G = nx.path_graph(3) # Path on 3 nodes
>>> nx.number_of_max_degree_nodes(G)
1
```

grinpy.functions.degree.neighborhood_degree_list

grinpy.functions.degree.**neighborhood_degree_list**(*G*, *nbunch*)

Return a list of the unique degrees of all neighbors of nodes in nbunch

G [graph] A NetworkX graph.

nbunch : a single node or iterable container of nodes

degreeList [list] A list of the degrees of all nodes in the neighborhood of the nodes in nbunch.

closed_neighborhood_degree_list, neighborhood

```
>>> import grinpy as gp
>>> G = gp.path_graph(3) # Path on 3 nodes
>>> gp.neighborhood_degree_list(G, 1)
[1, 2]
```

grinpy.functions.degree.closed_neighborhood_degree_list

grinpy.functions.degree.**closed_neighborhood_degree_list**(*G*, *nbunch*)

Return a list of the unique degrees of all nodes in the closed neighborhood of the nodes in nbunch.

G [graph] A NetworkX graph.

nbunch : a single node or iterable container of nodes

degreeList [list] A list of the degrees of all nodes in the closed neighborhood of the nodes in nbunch.

closed_neighborhood, neighborhood_degree_list

```
>>> import grinpy as gp
>>> G = gp.path_graph(3) # Path on 3 nodes
>>> gp.closed_neighborhood_degree_list(G, 1)
[1, 2, 2]
```

Neighborhoods

Functions for computing neighborhoods of vertices and sets of vertices.

neighborhood(G, nbunch)	Return a list of all neighbors of the nodes in nbunch.
closed_neighborhood(G, nbunch)	Return a list of all neighbors of the nodes in nbunch, including the nodes in nbunch.
are_neighbors(G, v, nbunch)	Returns true if v is adjacent to any of the nodes in nbunch.

grinpy.functions.neighborhoods.neighborhood

grinpy.functions.neighborhoods.**neighborhood**(*G*, *nbunch*)

Return a list of all neighbors of the nodes in nbunch.

G [graph] A NetworkX graph.

nbunch : a single node or iterable container

neighbors [list] A list containing all nodes that are a neighbor of some node in nbunch.

closed_neighborhood

```
>>> G = nx.path_graph(3) # Path on 3 nodes
>>> nx.neighborhood(G, 1)
[0, 2]
```

grinpy.functions.neighborhoods.closed_neighborhood

`grinpy.functions.neighborhoods.`**`closed_neighborhood`**`(`*G*, *nbunch*`)`

Return a list of all neighbors of the nodes in nbunch, including the nodes in nbunch.

G [graph] A NetworkX graph.

nbunch : a single node or iterable container

neighbors [list] A list containing all nodes that are a neighbor of some node in nbunch together with all nodes in nbunch.

neighborhood

```
>>> G = nx.path_graph(3) # Path on 3 nodes
>>> nx.closed_neighborhood(G, 1)
[0, 1, 2]
```

grinpy.functions.neighborhoods.are_neighbors

`grinpy.functions.neighborhoods.`**`are_neighbors`**`(`*G*, *v*, *nbunch*`)`

Returns true if v is adjacent to any of the nodes in nbunch. Otherwise, returns false.

G [graph] A NetworkX graph.

v [node] A node in the graph.

nbunch : a single node or iterable container

isNeighbor [bool] If nbunch in a single node, True if v in a neighbor that node and False otherwise.

If nbunch is an interable, True if v is a neighbor of some node in nbunch and False otherwise.

```
>>> G = nx.star_graph(3) # Star on 4 nodes
>>> nx.are_neighbors(G, 0, 1)
True
>>> nx.are_neighbors(G, 1, 2)
False
>>> nx.are_neighbors(G, 1, [0, 2])
True
```

3.2.3 Invariants

Release 0.1

Date Dec 09, 2017

Disparity

Functions for computing disparity related invariants.

vertex_disparity(G, v)	Return number of distinct degrees of neighbors of v.
closed_vertex_disparity(G, v)	Return number of distinct degrees of nodes in the closed neighborhood of v.
disparity_sequence(G)	Return the sequence of disparities of each node in the graph.
closed_disparity_sequence(G)	Return the sequence of closed disparities of each node in the graph.
CW_disparity(G)	Return the Caro-Wei disparity of the graph.
closed_CW_disparity(G)	Return the closed Caro-Wei disparity of the graph.
inverse_disparity(G)	Return the inverse disparity of the graph.
closed_inverse_disparity(G)	Return the closed inverse disparity of the graph.
average_vertex_disparity(G)	Return the average vertex disparity of the graph.
average_closed_vertex_disparity(G)	Return the average closed vertex disparity of the graph.
k_disparity(G, k)	Return the k-disparity of the graph.
closed_k_disparity(G, k)	Return the closed k-disparity of the graph.
irregularity(G)	Return the irregularity measure of the graph.

grinpy.invariants.disparity.vertex_disparity

grinpy.invariants.disparity.**vertex_disparity**(*G*, *v*)

Return number of distinct degrees of neighbors of v.

G [graph] A Networkx graph.

v : a node in G

disparity [int] The number of distinct degrees of neighbors of v.

closed_vertex_disparity

grinpy.invariants.disparity.closed_vertex_disparity

grinpy.invariants.disparity.**closed_vertex_disparity**(*G*, *v*)

Return number of distinct degrees of nodes in the closed neighborhood of v.

G [graph] A Networkx graph.

v : a node in G

closedDisparity [int] The number of distinct degrees of nodes in the closed neighborhood of v.

vertex_disparity

grinpy.invariants.disparity.disparity_sequence

grinpy.invariants.disparity.**disparity_sequence**(*G*)

Return the sequence of disparities of each node in the graph.

G [graph] A Networkx graph.

disparitySequence [list] The sequence of disparities of each node in the graph.

closed_disparity_sequence, vertex_disparity

grinpy.invariants.disparity.closed_disparity_sequence

`grinpy.invariants.disparity.`**`closed_disparity_sequence`**`(G)`

Return the sequence of closed disparities of each node in the graph.

G [graph] A Networkx graph.

disparitySequence [list] The sequence of closed disparities of each node in the graph.

closed_vertex_disparity, disparity_sequence

grinpy.invariants.disparity.CW_disparity

`grinpy.invariants.disparity.`**`CW_disparity`**`(G)`

Return the Caro-Wei disparity of the graph.

The *Caro-Wei disparity* of a graph is defined as:

$$\sum_{v \in V(G)}$$

rac{1}{1 + disp(v)}

where *V(G)* is the set of nodes of *G* and *disp(v)* is the disparity of the vertex v.

This invariant is inspired by the Caro-Wei bound for the independence number of a graph, hence the name.

G [graph] A Networkx graph.

cwDisparity [float] The Caro-Wei disparity of the graph.

closed_CW_disparity, closed_inverse_disparity, inverse_disparity

grinpy.invariants.disparity.closed_CW_disparity

`grinpy.invariants.disparity.`**`closed_CW_disparity`**`(G)`

Return the closed Caro-Wei disparity of the graph.

The *closed Caro-Wei disparity* of a graph is defined as:

$$\sum_{v \in V(G)}$$

rac{1}{1 + cdisp(v)}

where *V(G)* is the set of nodes of *G* and *cdisp(v)* is the closed disparity of the vertex v.

This invariant is inspired by the Caro-Wei bound for the independence number of a graph, hence the name.

G [graph] A Networkx graph.

closedCWDisparity [float] The closed Caro-Wei disparity of the graph.

CW_disparity, closed_inverse_disparity, inverse_disparity

grinpy.invariants.disparity.inverse_disparity

`grinpy.invariants.disparity.`**`inverse_disparity`**`(G)`

Return the inverse disparity of the graph.

The *inverse disparity* of a graph is defined as:

$$\sum_{v \in V(G)}$$

rac{1}{disp(v)}

where *V(G)* is the set of nodes of *G* and *disp(v)* is the disparity of the vertex v.

G [graph] A Networkx graph.

inverseDisparity [float] The inverse disparity of the graph.

CW_disparity, closed_CW_disparity, closed_inverse_disparity

grinpy.invariants.disparity.closed_inverse_disparity

`grinpy.invariants.disparity.`**`closed_inverse_disparity`**`(G)`

Return the closed inverse disparity of the graph.

The *closed inverse disparity* of a graph is defined as:

$$\sum_{v \in V(G)}$$

rac{1}{cdisp(v)}

where *V(G)* is the set of nodes of *G* and *cdisp(v)* is the closed disparity of the vertex v.

G [graph] A Networkx graph.

closedInverseDisparity [float] The closed inverse disparity of the graph.

CW_disparity, closed_CW_disparity, inverse_disparity

grinpy.invariants.disparity.average_vertex_disparity

`grinpy.invariants.disparity.`**`average_vertex_disparity`**`(G)`

Return the average vertex disparity of the graph.

G [graph] A Networkx graph.

avgDisparity [int] The average vertex disparity of the graph.

average_closed_vertex_disparity, vertex_disparity

grinpy.invariants.disparity.average_closed_vertex_disparity

grinpy.invariants.disparity.**average_closed_vertex_disparity**(*G*)

Return the average closed vertex disparity of the graph.

G [graph] A Networkx graph.

avgClosedDisparity [int] The average closed vertex disparity of the graph.

average_vertex_disparity, closed_vertex_disparity

grinpy.invariants.disparity.k_disparity

grinpy.invariants.disparity.**k_disparity**(*G*, *k*)

Return the k-disparity of the graph.

The *k-disparity* of a graph is defined as:

rac{2}{k(k+1)}sum_{i=0}^{k-i}(k-i)f(i)

where *k* is a positive integer and *f(i)* is the frequency of i in the disparity sequence.

G [graph] A Networkx graph.

kDisparity [float] The k-disparity of the graph.

closed_k_disparity

grinpy.invariants.disparity.closed_k_disparity

grinpy.invariants.disparity.**closed_k_disparity**(*G*, *k*)

Return the closed k-disparity of the graph.

The *closed k-disparity* of a graph is defined as:

rac{2}{k(k+1)}sum_{i=0}^{k-1}(k-i)d_i

where *k* is a positive integer and *d_i* is the frequency of i in the closed disparity sequence.

G [graph] A Networkx graph.

closedKDisparity [float] The closed k-disparity of the graph.

k_disparity

grinpy.invariants.disparity.irregularity

grinpy.invariants.disparity.**irregularity**(*G*)

Return the irregularity measure of the graph.

The *irregularity* of an *n*-vertex graph is defined as:

rac{2}{n(n+1)}sum_{i=0}^{n-i}(n-i)f(i)

where *f(i)* is the frequency of i in the closed disparity sequence.

G [graph] A Networkx graph.

irregularity [float] The irregularity of the graph.

k_disparity

Domination

Functions for computing dominating sets in a graph.

is_k_dominating_set(G, nbunch, k)	Return whether or not the nodes in nbunch comprise a k-dominating set.
is_total_dominating_set(G, nbunch)	Return whether or not the nodes in nbunch comprise a total dominating set.
min_k_dominating_set(G, k)	Return a smallest k-dominating set in the graph.
min_dominating_set(G)	Return a smallest dominating set in the graph.
min_total_dominating_set(G)	Return a smallest total dominating set in the graph.
domination_number(G)	Return the domination number the graph.
k_domination_number(G, k)	Return the k-domination number the graph.
total_domination_number(G)	Return the total domination number the graph.

grinpy.invariants.domination.is_k_dominating_set

grinpy.invariants.domination.**is_k_dominating_set**(*G*, *nbunch*, *k*)

Return whether or not the nodes in nbunch comprise a k-dominating set.

A *k-dominating set* is a set of nodes with the property that every node in the graph is either in the set or adjacent at least 1 and at most k nodes in the set.

This is a generalization of the well known concept of a dominating set (take k = 1).

G [graph] A Networkx graph.

nbunch: a single node or iterable container or nodes

k [int] A positive integer.

isKDominating [bool] True if the nodes in nbunch comprise a k-dominating set, and False otherwise.

grinpy.invariants.domination.is_total_dominating_set

grinpy.invariants.domination.**is_total_dominating_set**(*G*, *nbunch*)

Return whether or not the nodes in nbunch comprise a total dominating set.

A * total dominating set* is a set of nodes with the property that every node in the graph is adjacent to some node in the set.

G [graph] A Networkx graph.

nbunch: a single node or iterable container or nodes

isTotalDominating [bool] True if the nodes in nbunch comprise a k-dominating set, and False otherwise.

grinpy.invariants.domination.min_k_dominating_set

`grinpy.invariants.domination.min_k_dominating_set(G, k)`

Return a smallest k-dominating set in the graph.

The method to compute the set is brute force except that the subsets searched begin with those whose cardinality is equal to the sub-k-domination number of the graph, which was defined by Amos et al. and shown to be a tractable lower bound for the k-domination number.

G [graph] A Networkx graph.

k [int] A positive integer.

minKDominatingSet [list] A smallest k-dominating set in the graph.

D. Amos, J. Asplund, and R. Davila, The sub-k-domination number of a graph with applications to k-domination, *arXiv preprint arXiv:1611.02379*, (2016)

grinpy.invariants.domination.min_dominating_set

`grinpy.invariants.domination.min_dominating_set(G)`

Return a smallest dominating set in the graph.

The method to compute the set is brute force except that the subsets searched begin with those whose cardinality is equal to the sub-domination number of the graph, which was defined by Amos et al. and shown to be a tractable lower bound for the k-domination number.

G [graph] A Networkx graph.

k [int] A positive integer.

minDominatingSet [list] A smallest dominating set in the graph.

min_k_dominating_set

D. Amos, J. Asplund, B. Brimkov and R. Davila, The sub-k-domination number of a graph with applications to k-domination, *arXiv preprint arXiv:1611.02379*, (2016)

grinpy.invariants.domination.min_total_dominating_set

`grinpy.invariants.domination.min_total_dominating_set(G)`

Return a smallest total dominating set in the graph.

The method to compute the set is brute force except that the subsets searched begin with those whose cardinality is equal to the sub-total-domination number of the graph, which was defined by Davila and shown to be a tractable lower bound for the k-domination number.

G [graph] A Networkx graph.

minTotalDominatingSet [list] A smallest total dominating set in the graph.

R. Davila, A note on sub-total domination in graphs. *arXiv preprint arXiv:1701.07811*, (2017)

grinpy.invariants.domination.domination_number

`grinpy.invariants.domination.`**`domination_number`**`(`*G*`)`

Return the domination number the graph.

The *domination number* of a graph is the cardinality of a smallest dominating set of nodes in the graph.

The method to compute this number modified brute force.

G [graph] A Networkx graph.

dominationNumber [int] The domination number of the graph.

min_dominating_set, k_domination_number

grinpy.invariants.domination.k_domination_number

`grinpy.invariants.domination.`**`k_domination_number`**`(`*G*, *k*`)`

Return the k-domination number the graph.

The *k-domination number* of a graph is the cardinality of a smallest k-dominating set of nodes in the graph.

The method to compute this number is modified brute force.

G [graph] A Networkx graph.

kDominationNumber [int] The k-domination number of the graph.

min_k_dominating_set, domination_number

grinpy.invariants.domination.total_domination_number

`grinpy.invariants.domination.`**`total_domination_number`**`(`*G*`)`

Return the total domination number the graph.

The *total domination number* of a graph is the cardinality of a smallest total dominating set of nodes in the graph.

The method to compute this number is modified brute force.

G [graph] A Networkx graph.

totalDominationNumber [int] The total domination number of the graph.

DSI

Functions for computing DSI style invariants.

sub_k_domination_number(G, k)	Return the sub-k-domination number of the graph.
slater(G)	Return the Slater invariant for the graph.
sub_total_domination_number(G)	Return the sub-total domination number of the graph.
annihilation_number(G)	Return the annihilation number of the graph.

grinpy.invariants.dsi.sub_k_domination_number

`grinpy.invariants.dsi.sub_k_domination_number(G, k)`

Return the sub-k-domination number of the graph.

The *sub-k-domination number* of a graph G with n nodes is defined as the smallest positive integer t such that the following relation holds:

$$t+$$

rac{1}{k}sum_{i=0}^t d_i geq n

where

$$d_1 \geq d_2 \geq \cdots \geq \underset{\cdot}{n}$$

is the degree sequence of the graph.

G [graph] A Networkx graph.

k [int] A positive integer.

sub [int] The sub-k-domination number of a graph.

slater

```
>>> G = nx.cycle_graph(4)
>>> nx.sub_k_domination_number(G, 1)
True
```

D. Amos, J. Asplund, B. Brimkov and R. Davila, The sub-k-domination number of a graph with applications to k-domination, *arXiv preprint arXiv:1611.02379*, (2016)

grinpy.invariants.dsi.slater

`grinpy.invariants.dsi.slater(G)`

Return the Slater invariant for the graph.

The Slater invariant of a graph G is a lower bound for the domination number of a graph defined by:

$$sl(G) = \min t : t + \sum_{i=0}^{t} d_i \geq n$$

where

$$d_1 \geq d_2 \geq \cdots \geq \underset{\cdot}{n}$$

is the degree sequence of the graph ordered in non-increasing order and n is the order of G.

Amos et al. rediscovered this invariant and generalized it into what is now known as the sub-domination number.

G [graph] A Networkx graph.

slater [int] The Slater invariant for the graph.

sub_k_domination_number

D. Amos, J. Asplund, B. Brimkov and R. Davila, The sub-k-domination number of a graph with applications to k-domination, *arXiv preprint arXiv:1611.02379*, (2016)

P.J. Slater, Locating dominating sets and locating-dominating set, *Graph Theory, Combinatorics and Applications: Proceedings of the 7th Quadrennial International Conference on the Theory and Applications of Graphs*, 2: 2073-1079 (1995)

grinpy.invariants.dsi.sub_total_domination_number

`grinpy.invariants.dsi.`**`sub_total_domination_number`**`(G)`

Return the sub-total domination number of the graph.

The sub-total domination number is defined as:

$$sub_t(G) = \min t : \sum_{i=0}^{t} d_i \geq n$$

where

$$d_1 \geq d_2 \geq \cdots \geq n$$

is the degree sequence of the graph ordered in non-increasing order and *n* is the order of the graph.

This invariant was defined and investigated by Randy Davila.

G [graph] A Networkx graph.

subTotalDominationNumber [int] The sub-total domination number of the graph.

R. Davila, A note on sub-total domination in graphs. *arXiv preprint arXiv:1701.07811*, (2017)

grinpy.invariants.dsi.annihilation_number

`grinpy.invariants.dsi.`**`annihilation_number`**`(G)`

Return the annihilation number of the graph.

The annihilation number of a graph G is defined as:

$$a(G) = \max t : \sum_{i=0}^{t} d_i \leq m$$

where

$$d_1 \leq d_2 \leq \cdots \leq n$$

is the degree sequence of the graph ordered in non-decreasing order and m is the number of edges in G.

G [graph] A Networkx graph.

annihilationNumber [int] The annihilation number of the graph.

Independence

Functions for computing independence related invariants for a graph.

is_independent_set(G, nbunch)	Return whether or not the nodes in nbunch comprise an independent set.
is_k_independent_set(G, nbunch, k)	Return whether or not the nodes in nbunch comprise an a k-independent set.
max_k_independent_set(G, k)	Return a largest k-independent set of nodes in *G*.
max_independent_set(G)	Return a largest independent set of nodes in *G*.
independence_number(G)	Return a the independence number of G.
k_independence_number(G, k)	Return a the k-independence number of G.

grinpy.invariants.independence.is_independent_set

grinpy.invariants.independence.**is_independent_set**(*G*, *nbunch*)

Return whether or not the nodes in nbunch comprise an independent set.

An set *S* of nodes in *G* is called an *independent set* if no two nodes in S are neighbors of one another.

G [graph] A Networkx graph.

nbunch : a single node or iterable container of nodes.

isIndependent [bool] True if the nodes in nbunch comprise an independent set, False otherwise.

is_k_independent_set

grinpy.invariants.independence.is_k_independent_set

grinpy.invariants.independence.**is_k_independent_set**(*G*, *nbunch*, *k*)

Return whether or not the nodes in nbunch comprise an a k-independent set.

A set *S* of nodes in *G* is called a *k-independent set* it every node in S has at most *k*-1 neighbors in S. Notice that a 1-independent set is equivalent to an independent set.

G [graph] A Networkx graph.

nbunch : a single node or iterable container of nodes.

k [int] A positive integer.

isKIndependent [bool] True if the nodes in nbunch comprise a k-independent set, False otherwise.

is_independent_set

grinpy.invariants.independence.max_k_independent_set

grinpy.invariants.independence.**max_k_independent_set**(*G*, *k*)

Return a largest k-independent set of nodes in *G*.

The method used is brute force, except when *k*=1. In this case, the search starts with subsets of *G* with cardinality equal to the annihilation number of *G*, which was shown by Pepper to be an upper bound for the independence number of a graph, and then continues checking smaller subsets until a maximum independent set is found.

G [graph] A Networkx graph.

k [int] A positive integer.

maxKIndependentSet [list] A list of nodes comprising a largest k-independent set in *G*.

max_independent_set

grinpy.invariants.independence.max_independent_set

`grinpy.invariants.independence.`**`max_independent_set`**`(G)`

Return a largest independent set of nodes in *G*.

The method used is a modified brute force search. The search starts with subsets of *G* with cardinality equal to the annihilation number of *G*, which was shown by Pepper to be an upper bound for the independence number of a graph, and then continues checking smaller subsets until a maximum independent set is found.

G [graph] A Networkx graph.

maxIndependentSet [list] A list of nodes comprising a largest independent set in *G*.

max_independent_set

grinpy.invariants.independence.independence_number

`grinpy.invariants.independence.`**`independence_number`**`(G)`

Return a the independence number of G.

The *independence number* of a graph is the cardinality of a largest independent set of nodes in the graph.

G [graph] A Networkx graph.

independenceNumber [int] The independence number of *G*.

k_independence_number

grinpy.invariants.independence.k_independence_number

`grinpy.invariants.independence.`**`k_independence_number`**`(G, k)`

Return a the k-independence number of G.

The *k-independence number* of a graph is the cardinality of a largest k-independent set of nodes in the graph.

G [graph] A Networkx graph.

k [int] A positive integer.

kIndependenceNumber [int] The k-independence number of *G*.

independence_number

Power Domination

Functions for computing power domination related invariants of a graph.

is_power_dominating_set(G, nbunch)	Return whether or not the nodes in nbunch comprise a power dominating set.
min_power_dominating_set(G)	Return a smallest power dominating set of nodes in *G*.
power_domination_number(G)	Return the power domination number of *G*.

grinpy.invariants.power_domination.is_power_dominating_set

grinpy.invariants.power_domination.**is_power_dominating_set**(*G*, *nbunch*)

Return whether or not the nodes in nbunch comprise a power dominating set.

G [graph] A Networkx graph.

nbunch : a single node or iterable container of nodes.

isPowerDominating [bool] True if the nodes in nbunch comprise a power dominating set, False otherwise.

grinpy.invariants.power_domination.min_power_dominating_set

grinpy.invariants.power_domination.**min_power_dominating_set**(*G*)

Return a smallest power dominating set of nodes in *G*.

The method used to compute the set is brute force.

G [graph] A Networkx graph.

minPowerDominatingSet [list] A smallest power dominating set in *G*.

grinpy.invariants.power_domination.power_domination_number

grinpy.invariants.power_domination.**power_domination_number**(*G*)

Return the power domination number of *G*.

G [graph] A Networkx graph.

powerDominationNumber [int] The power domination number of *G*.

Residue

Functions for computing the residue and related invariants.

residue(G)	Return the *residue* of *G*.
k_residue(G, k)	Return the *k-residue* of *G*.

grinpy.invariants.residue.residue

grinpy.invariants.residue.**residue**(*G*)

Return the *residue* of *G*.

The *residue* of a graph *G* is the number of zeros obtained in final sequence of the Havel Hakimi process.

G [graph] A Networkx graph.

residue [int] The residue of *G*.

k_residue, havel_hakimi_process

grinpy.invariants.residue.k_residue

`grinpy.invariants.residue.`**`k_residue`**`(`*G*, *k*`)`

Return the *k-residue* of *G*.

The *k-residue* of a graph *G* is defined as follows:

rac{1}{k}sum_{i=0}^{k-1}(k - i)f(i)

where *f(i)* is the frequency of *i* in the elmination sequence of the graph. The elimination sequence is the sequence of deletions made during the Havel Hakimi process together with the zeros obtained in the final step.

G [graph] A Networkx graph.

kResidue [float] The k-residue of *G*.

residue, havel_hakimi_process, elimination_sequence

Zero Forcing

Functions for computing zero forcing related invariants of a graph.

is_k_forcing_vertex(G, v, nbunch, k)	Return whether or not *v* can *k*-force relative to the set of nodes in nbunch.
is_k_forcing_active_set(G, nbunch, k)	Return whether or not at least one node in nbunch can *k*-force.
is_k_forcing_set(G, nbunch, k)	Return whether or not the nodes in nbunch comprise a *k*-forcing set in *G*.
min_k_forcing_set(G, k)	Return a smallest *k*-forcing set in *G*.
k_forcing_number(G, k)	Return the *k*-forcing number of *G*.
is_zero_forcing_vertex(G, v, nbunch)	Return whether or not *v* can force relative to the set of nodes in nbunch.
is_zero_forcing_active_set(G, nbunch)	Return whether or not at least one node in nbunch can force.
is_zero_forcing_set(G, S)	Return whether or not the nodes in nbunch comprise a zero forcing set in *G*.
min_zero_forcing_set(G)	Return a smallest zero forcing set in *G*.
zero_forcing_number(G)	Return the zero forcing number of *G*.

grinpy.invariants.zero_forcing.is_k_forcing_vertex

`grinpy.invariants.zero_forcing.`**`is_k_forcing_vertex`**`(`*G*, *v*, *nbunch*, *k*`)`

Return whether or not *v* can *k*-force relative to the set of nodes in nbunch.

G [graph] A Networkx graph.

v : a single node in *G*

nbunch: a single node or iterable container of nodes in *G*.

k [int] A positive integer.

isForcing [bool] True if *v* can *k*-force relative to the nodes in nbunch. False otherwise.

grinpy.invariants.zero_forcing.is_k_forcing_active_set

`grinpy.invariants.zero_forcing.`**`is_k_forcing_active_set`**(*G*, *nbunch*, *k*)

Return whether or not at least one node in nbunch can *k*-force.

G [graph] A Networkx graph.

nbunch: a single node or iterable container of nodes in *G*

k [int] A positive integer.

isActive [bool] True if at least one of the nodes in nbunch can *k*-force. False otherwise.

grinpy.invariants.zero_forcing.is_k_forcing_set

`grinpy.invariants.zero_forcing.`**`is_k_forcing_set`**(*G*, *nbunch*, *k*)

Return whether or not the nodes in nbunch comprise a *k*-forcing set in *G*.

G [graph] A Networkx graph.

nbunch: a single node or iterable container of nodes in *G*.

k [int] A positive integer.

isForcingSet [bool] True if the nodes in nbunch comprise a *k*-forcing set in *G*. False otherwise.

grinpy.invariants.zero_forcing.min_k_forcing_set

`grinpy.invariants.zero_forcing.`**`min_k_forcing_set`**(*G*, *k*)

Return a smallest *k*-forcing set in *G*.

The method used to compute the set is brute force.

G [graph] A Networkx graph.

k [int] A positive integer.

minForcingSet [list] A smallest *k*-forcing set in *G*.

grinpy.invariants.zero_forcing.k_forcing_number

`grinpy.invariants.zero_forcing.`**`k_forcing_number`**(*G*, *k*)

Return the *k*-forcing number of *G*.

The *k*-forcing number of a graph is the cardinality of a smallest *k*-forcing set in the graph.

G [graph] A Networkx graph.

k [int] A positive integer.

kForcingNum [int] The *k*-forcing number of *G*.

grinpy.invariants.zero_forcing.is_zero_forcing_vertex

grinpy.invariants.zero_forcing.**is_zero_forcing_vertex**(*G*, *v*, *nbunch*)

Return whether or not *v* can force relative to the set of nodes in nbunch.

G [graph] A Networkx graph.

v: a single node in *G*

nbunch: a single node or iterable container of nodes in *G*.

isForcing [bool] True if *v* can force relative to the nodes in nbunch. False otherwise.

grinpy.invariants.zero_forcing.is_zero_forcing_active_set

grinpy.invariants.zero_forcing.**is_zero_forcing_active_set**(*G*, *nbunch*)

Return whether or not at least one node in nbunch can force.

G [graph] A Networkx graph.

nbunch: a single node or iterable container of nodes in *G*

isActive [bool] True if at least one of the nodes in nbunch can force. False otherwise.

grinpy.invariants.zero_forcing.is_zero_forcing_set

grinpy.invariants.zero_forcing.**is_zero_forcing_set**(*G*, *S*)

Return whether or not the nodes in nbunch comprise a zero forcing set in *G*.

G [graph] A Networkx graph.

nbunch: a single node or iterable container of nodes in *G*.

isForcingSet [bool] True if the nodes in nbunch comprise a zero forcing set in *G*. False otherwise.

grinpy.invariants.zero_forcing.min_zero_forcing_set

grinpy.invariants.zero_forcing.**min_zero_forcing_set**(*G*)

Return a smallest zero forcing set in *G*.

The method used to compute the set is brute force.

G [graph] A Networkx graph.

minForcingSet [list] A smallest zero forcing set in *G*.

grinpy.invariants.zero_forcing.zero_forcing_number

grinpy.invariants.zero_forcing.**zero_forcing_number**(*G*)

Return the zero forcing number of *G*.

The zero forcing number of a graph is the cardinality of a smallest zero forcing set in the graph.

G [graph] A Networkx graph.

zeroForcingNum [int] The zero forcing number of *G*.

3.3 License

GrinPy is distributed with the 3-clause BSD license. As an extension of the NetworkX package, we list the pertinent copyright information as requested by the NetworkX authors.

CHAPTER 4

Indices and tables

- genindex
- modindex
- search

Python Module Index

g

Index

Symbols

A

C

D

G

H

I

K

M

N

P

R

S

T

V

Z