
DWaveNetworkX Documentation

Release 0.8.8

D-Wave Systems Inc

Dec 02, 2020

Contents

1 Installation	3
2 License	5
Bibliography	59
Python Module Index	61
Index	63

D-Wave NetworkX is an extension of [NetworkX](#)—a Python language package for exploration and analysis of networks and network algorithms—for users of D-Wave Systems. It provides tools for working with Chimera graphs and implementations of graph-theory algorithms on the D-Wave system and other binary quadratic model samplers.

The example below generates a graph for a Chimera unit cell (eight nodes in a 4-by-2 bipartite architecture).

```
>>> import dwave_networkx as dnx
>>> graph = dnx.chimera_graph(1, 1, 4)
```

See the documentation for more examples.

CHAPTER 1

Installation

Installation from PyPi:

```
pip install dwave_networkx
```

Installation from source:

```
pip install -r requirements.txt  
python setup.py install
```


Released under the Apache License 2.0.

2.1 Documentation

Note: This documentation is for the latest version of [dwave-networkx](#). Documentation for the version currently installed by [dwave-ocean-sdk](#) is here: [dwave-networkx](#).

2.1.1 Introduction

D-Wave NetworkX provides tools for working with [Chimera](#) and [Pegasus](#) graphs and implementations of graph-theory algorithms on the D-Wave system and other binary quadratic model [samplers](#); for example, functions such as `draw_chimera()` provide easy visualization for Chimera graphs; functions such as `maximum_cut()` or `min_vertex_cover()` provide graph algorithms useful to optimization problems that fit well with the D-Wave system.

Like the D-Wave system, all other supported samplers must have `sample_qubo` and `sample_ising` methods for solving [Ising](#) and [QUBO](#) models and return an iterable of samples in order of increasing energy. You can set a default sampler using the `set_default_sampler()` function.

- For an introduction to quantum processing unit (QPU) topologies such as the Chimera and Pegasus graphs, see [Topology](#).
- For an introduction to binary quadratic models (BQMs), see [Binary Quadratic Models](#).
- For an introduction to samplers, see [Samplers and Composites](#).

Example

Below you can see how to create Chimera graphs implemented in the D-Wave 2X and D-Wave 2000Q systems:

```
import dwave_networkx as dnx

# D-Wave 2X
C = dnx.chimera_graph(12, 12, 4)

# D-Wave 2000Q
C = dnx.chimera_graph(16, 16, 4)
```

2.1.2 Reference Documentation

Release 0.8.8

Date Dec 02, 2020

Algorithms

Implementations of graph-theory algorithms on the D-Wave system and other binary quadratic model samplers.

Canonicalization

<code>canonical_chimera_labeling(G[, t])</code>	Returns a mapping from the labels of G to chimera-indexed labeling.
---	---

`dwave_networkx.canonical_chimera_labeling`

canonical_chimera_labeling (*G*, *t=None*)

Returns a mapping from the labels of G to chimera-indexed labeling.

Parameters

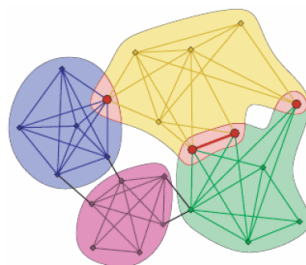
- **G** (*NetworkX graph*) – A Chimera-structured graph.
- **t** (*int (optional, default 4)*) – Size of the shore within each Chimera tile.

Returns `chimera_indices` – A mapping from the current labels to a 4-tuple of Chimera indices.

Return type `dict`

Clique

A clique in an undirected graph $G = (V, E)$ is a subset of the vertex set such that for every two vertices in C there exists an edge connecting the two.



<code>maximum_clique(G[, sampler, lagrange])</code>	Returns an approximate maximum clique.
<code>clique_number(G[, sampler, lagrange])</code>	Returns the number of vertices in the maximum clique of a graph.
<code>is_clique(G, clique_nodes)</code>	Determines whether the given nodes form a clique.

dwave_networkx.maximum_clique

maximum_clique (*G*, *sampler=None*, *lagrange=2.0*, ***sampler_args*)

Returns an approximate maximum clique. A clique in an undirected graph, $G = (V, E)$, is a subset of the vertex set $C \subseteq V$ such that for every two vertices in C there exists an edge connecting the two. This is equivalent to saying that the subgraph induced by C is complete (in some cases, the term clique may also refer to the subgraph). A maximum clique is a clique of the largest possible size in a given graph.

This function works by finding the maximum independent set of the compliment graph of the given graph G which is equivalent to finding maximum clique. It defines a QUBO with ground states corresponding to a maximum weighted independent set and uses the sampler to sample from it.

Parameters

- **G** (*NetworkX graph*) – The graph on which to find a maximum clique.
- **sampler** – A binary quadratic model sampler. A sampler is a process that samples from low energy states in models defined by an Ising equation or a Quadratic Unconstrained Binary Optimization Problem (QUBO). A sampler is expected to have a ‘sample_qubo’ and ‘sample_ising’ method. A sampler is expected to return an iterable of samples, in order of increasing energy. If no sampler is provided, one must be provided using the `set_default_sampler` function.
- **lagrange** (*optional (default 2)*) – Lagrange parameter to weight constraints (no edges within set) versus objective (largest set possible).
- **sampler_args** – Additional keyword parameters are passed to the sampler.

Returns clique_nodes – List of nodes that form a maximum clique, as determined by the given sampler.

Return type `list`

Notes

Samplers by their nature may not return the optimal solution. This function does not attempt to confirm the quality of the returned sample.

References

[Maximum Clique on Wikipedia](#)

[Independent Set on Wikipedia](#)

[QUBO on Wikipedia](#)

dwave_networkx.clique_number

clique_number (*G*, *sampler=None*, *lagrange=2.0*, ***sampler_args*)

Returns the number of vertices in the maximum clique of a graph. A maximum clique is a clique of the largest possible size in a given graph. The clique number $\omega(G)$ of a graph *G* is the number of vertices in a maximum clique in *G*. The intersection number of *G* is the smallest number of cliques that together cover all edges of *G*.

This function works by finding the maximum independent set of the compliment graph of the given graph *G* which is equivalent to finding maximum clique. It defines a QUBO with ground states corresponding to a maximum weighted independent set and uses the sampler to sample from it.

Parameters

- **G** (*NetworkX graph*) – The graph on which to find a maximum clique.
- **sampler** – A binary quadratic model sampler. A sampler is a process that samples from low energy states in models defined by an Ising equation or a Quadratic Unconstrained Binary Optimization Problem (QUBO). A sampler is expected to have a ‘sample_qubo’ and ‘sample_ising’ method. A sampler is expected to return an iterable of samples, in order of increasing energy. If no sampler is provided, one must be provided using the *set_default_sampler* function.
- **lagrange** (*optional (default 2)*) – Lagrange parameter to weight constraints (no edges within set) versus objective (largest set possible).
- **sampler_args** – Additional keyword parameters are passed to the sampler.

Returns clique_nodes – List of nodes that form a maximum clique, as determined by the given sampler.

Return type *list*

Notes

Samplers by their nature may not return the optimal solution. This function does not attempt to confirm the quality of the returned sample.

References

[Maximum Clique on Wikipedia](#)

dwave_networkx.is_clique

is_clique (*G*, *clique_nodes*)

Determines whether the given nodes form a clique.

A clique is a subset of nodes of an undirected graph such that every two distinct nodes in the clique are adjacent.

Parameters

- **G** (*NetworkX graph*) – The graph on which to check the clique nodes.
- **clique_nodes** (*list*) – List of nodes that form a clique, as determined by the given sampler.

Returns is_clique – True if *clique_nodes* forms a clique.

Return type bool

Example

This example checks two sets of nodes, both derived from a single Chimera unit cell, for an independent set. The first set is the horizontal tile's nodes; the second has nodes from the horizontal and vertical tiles.

```
>>> import dwave_networkx as dnx
>>> G = dnx.chimera_graph(1, 1, 4)
>>> dnx.is_clique(G, [0, 1, 2, 3])
False
>>> dnx.is_clique(G, [0, 4])
True
```

Coloring

Graph coloring is the problem of assigning a color to the vertices of a graph in a way that no adjacent vertices have the same color.

Example

The map-coloring problem is to assign a color to each region of a map (represented by a vertex on a graph) such that any two regions sharing a border (represented by an edge of the graph) have different colors.



Fig. 1: Coloring a map of Canada with four colors.

<code>is_vertex_coloring(G, coloring)</code>	Determines whether the given coloring is a vertex coloring of graph G.
<code>min_vertex_color(G[, sampler, chromatic_lb, ...])</code>	Returns an approximate minimum vertex coloring.
<code>min_vertex_color_qubo(G[, chromatic_lb, ...])</code>	Return a QUBO with ground states corresponding to a minimum vertex coloring.
<code>vertex_color(G, colors[, sampler])</code>	Returns an approximate vertex coloring.
<code>vertex_color_qubo(G, colors)</code>	Return the QUBO with ground states corresponding to a vertex coloring.

dwave_networkx.algorithms.coloring.is_vertex_coloring

is_vertex_coloring (*G*, *coloring*)

Determines whether the given coloring is a vertex coloring of graph *G*.

Parameters

- **G** (*NetworkX graph*) – The graph on which the vertex coloring is applied.
- **coloring** (*dict*) – A coloring of the nodes of *G*. Should be a dict of the form {node: color, ...}.

Returns is_vertex_coloring – True if the given coloring defines a vertex coloring; that is, no two adjacent vertices share a color.

Return type `bool`

Example

This example checks two colorings for a graph, *G*, of a single Chimera unit cell. The first uses one color (0) for the four horizontal qubits and another (1) for the four vertical qubits, in which case there are no adjacencies; the second coloring swaps the color of one node.

```
>>> G = dnx.chimera_graph(1,1,4)
>>> colors = {0: 0, 1: 0, 2: 0, 3: 0, 4: 1, 5: 1, 6: 1, 7: 1}
>>> dnx.is_vertex_coloring(G, colors)
True
>>> colors[4]=0
>>> dnx.is_vertex_coloring(G, colors)
False
```

dwave_networkx.algorithms.coloring.min_vertex_color

min_vertex_color (*G*, *sampler=None*, *chromatic_lb=None*, *chromatic_ub=None*, ***sampler_args*)

Returns an approximate minimum vertex coloring.

Vertex coloring is the problem of assigning a color to the vertices of a graph in a way that no adjacent vertices have the same color. A minimum vertex coloring is the problem of solving the vertex coloring problem using the smallest number of colors.

Defines a QUBO [DWMP] with ground states corresponding to minimum vertex colorings and uses the sampler to sample from it.

Parameters

- **G** (*NetworkX graph*) – The graph on which to find a minimum vertex coloring.
- **sampler** – A binary quadratic model sampler. A sampler is a process that samples from low energy states in models defined by an Ising equation or a Quadratic Unconstrained Binary Optimization Problem (QUBO). A sampler is expected to have a ‘sample_qubo’ and ‘sample_ising’ method. A sampler is expected to return an iterable of samples, in order of increasing energy. If no sampler is provided, one must be provided using the `set_default_sampler` function.
- **chromatic_lb** (*int*, *optional*) – A lower bound on the chromatic number. If one is not provided, a bound is calculated.

- **chromatic_ub**(*int*, *optional*) – An upper bound on the chromatic number. If one is not provided, a bound is calculated.
- **sampler_args** – Additional keyword parameters are passed to the sampler.

Returns coloring – A coloring for each vertex in G such that no adjacent nodes share the same color. A dict of the form {node: color, ... }

Return type dict

References

Notes

Samplers by their nature may not return the optimal solution. This function does not attempt to confirm the quality of the returned sample.

dwave_networkx.algorithms.coloring.min_vertex_color_qubo

min_vertex_color_qubo(G , *chromatic_lb=None*, *chromatic_ub=None*)

Return a QUBO with ground states corresponding to a minimum vertex coloring.

Vertex coloring is the problem of assigning a color to the vertices of a graph in a way that no adjacent vertices have the same color. A minimum vertex coloring is the problem of solving the vertex coloring problem using the smallest number of colors.

Defines a QUBO [DWMP] with ground states corresponding to minimum vertex colorings and uses the sampler to sample from it.

Parameters

- **G** (*NetworkX graph*) – The graph on which to find a minimum vertex coloring.
- **chromatic_lb**(*int*, *optional*) – A lower bound on the chromatic number. If one is not provided, a bound is calculated.
- **chromatic_ub**(*int*, *optional*) – An upper bound on the chromatic number. If one is not provided, a bound is calculated.
- **sampler_args** – Additional keyword parameters are passed to the sampler.

Returns QUBO – The QUBO with ground states corresponding to minimum colorings of the graph. The QUBO variables are labelled (v, c) where v is a node in G and c is a color. In the ground state of the QUBO, a variable (v, c) has value 1 if v should be colored c in a valid coloring.

Return type dict

dwave_networkx.algorithms.coloring.vertex_color

vertex_color(G , *colors*, *sampler=None*, ***sampler_args*)

Returns an approximate vertex coloring.

Vertex coloring is the problem of assigning a color to the vertices of a graph in a way that no adjacent vertices have the same color.

Defines a QUBO [DWMP] with ground states corresponding to valid vertex colorings and uses the sampler to sample from it.

Parameters

- **G** (*NetworkX graph*) – The graph on which to find a minimum vertex coloring.
- **colors** (*int/sequence*) – The colors. If an int, the colors are labelled $[0, n)$. The number of colors must be greater or equal to the chromatic number of the graph.
- **sampler** – A binary quadratic model sampler. A sampler is a process that samples from low energy states in models defined by an Ising equation or a Quadratic Unconstrained Binary Optimization Problem (QUBO). A sampler is expected to have a ‘sample_qubo’ and ‘sample_ising’ method. A sampler is expected to return an iterable of samples, in order of increasing energy. If no sampler is provided, one must be provided using the `set_default_sampler` function.
- **sampler_args** – Additional keyword parameters are passed to the sampler.

Returns coloring – A coloring for each vertex in G such that no adjacent nodes share the same color. A dict of the form `{node: color, ...}`

Return type `dict`

References

Notes

Samplers by their nature may not return the optimal solution. This function does not attempt to confirm the quality of the returned sample.

`dwave_networkx.algorithms.coloring.vertex_color_qubo`

vertex_color_qubo (G , *colors*)

Return the QUBO with ground states corresponding to a vertex coloring.

If V is the set of nodes, E is the set of edges and C is the set of colors the resulting qubo will have:

- $|V| * |C|$ variables/nodes
- $|V| * |C| * (|C| - 1)/2 + |E| * |C|$ interactions/edges

The QUBO has ground energy $-|V|$ and an infeasible gap of 1.

Parameters

- **G** (*NetworkX graph*) – The graph on which to find a minimum vertex coloring.
- **colors** (*int/sequence*) – The colors. If an int, the colors are labelled $[0, n)$. The number of colors must be greater or equal to the chromatic number of the graph.

Returns QUBO – The QUBO with ground states corresponding to valid colorings of the graph. The QUBO variables are labelled (v, c) where v is a node in G and c is a color. In the ground state of the QUBO, a variable (v, c) has value 1 if v should be colored c in a valid coloring.

Return type `dict`

Cover

Vertex covering is the problem of finding a set of vertices such that all the edges of the graph are incident to at least one of the vertices in the set.

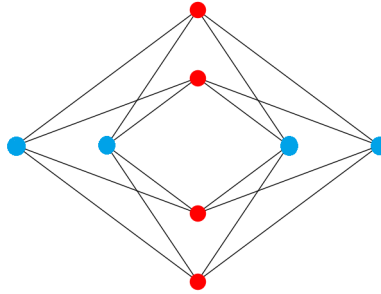


Fig. 2: Cover for a Chimera unit cell: the nodes of both the blue set of vertices (the horizontal tile of the Chimera unit cell) and the red set (vertical tile) connect to all 16 edges of the graph.

<code>min_weighted_vertex_cover(G[, weight, ...])</code>	Returns an approximate minimum weighted vertex cover.
<code>min_vertex_cover(G[, sampler, lagrange])</code>	Returns an approximate minimum vertex cover.
<code>is_vertex_cover(G, vertex_cover)</code>	Determines whether the given set of vertices is a vertex cover of graph G.

`dwave_networkx.algorithms.cover.min_weighted_vertex_cover`

min_weighted_vertex_cover (*G*, *weight=None*, *sampler=None*, *lagrange=2.0*, ***sampler_args*)

Returns an approximate minimum weighted vertex cover.

Defines a QUBO with ground states corresponding to a minimum weighted vertex cover and uses the sampler to sample from it.

A vertex cover is a set of vertices such that each edge of the graph is incident with at least one vertex in the set. A minimum weighted vertex cover is the vertex cover of minimum total node weight.

Parameters

- **G** (*NetworkX graph*) –
- **weight** (*string, optional (default None)*) – If None, every node has equal weight. If a string, use this node attribute as the node weight. A node without this attribute is assumed to have max weight.
- **sampler** – A binary quadratic model sampler. A sampler is a process that samples from low energy states in models defined by an Ising equation or a Quadratic Unconstrained Binary Optimization Problem (QUBO). A sampler is expected to have a ‘sample_qubo’ and ‘sample_ising’ method. A sampler is expected to return an iterable of samples, in order of increasing energy. If no sampler is provided, one must be provided using the `set_default_sampler` function.
- **lagrange** (*optional (default 2)*) – Lagrange parameter to weight constraints versus objective.
- **sampler_args** – Additional keyword parameters are passed to the sampler.

Returns `vertex_cover` – List of nodes that form the minimum weighted vertex cover, as determined by the given sampler.

Return type `list`

Notes

2.1. Samplers by their nature may not return the optimal solution. This function does not attempt to confirm the quality of the returned sample.

https://en.wikipedia.org/wiki/Vertex_cover

https://en.wikipedia.org/wiki/Quadratic_unconstrained_binary_optimization

dwave_networkx.algorithms.cover.min_vertex_cover

min_vertex_cover (*G*, *sampler=None*, *lagrange=2.0*, ***sampler_args*)

Returns an approximate minimum vertex cover.

Defines a QUBO with ground states corresponding to a minimum vertex cover and uses the sampler to sample from it.

A vertex cover is a set of vertices such that each edge of the graph is incident with at least one vertex in the set. A minimum vertex cover is the vertex cover of smallest size.

Parameters

- **G** (*NetworkX graph*) – The graph on which to find a minimum vertex cover.
- **sampler** – A binary quadratic model sampler. A sampler is a process that samples from low energy states in models defined by an Ising equation or a Quadratic Unconstrained Binary Optimization Problem (QUBO). A sampler is expected to have a ‘sample_qubo’ and ‘sample_ising’ method. A sampler is expected to return an iterable of samples, in order of increasing energy. If no sampler is provided, one must be provided using the *set_default_sampler* function.
- **lagrange** (*optional (default 2)*) – Lagrange parameter to weight constraints versus objective.
- **sampler_args** – Additional keyword parameters are passed to the sampler.

Returns **vertex_cover** – List of nodes that form a minimum vertex cover, as determined by the given sampler.

Return type *list*

Examples

This example uses a sampler from `dimod` to find a minimum vertex cover for a Chimera unit cell. Both the horizontal (vertices 0,1,2,3) and vertical (vertices 4,5,6,7) tiles connect to all 16 edges, so repeated executions can return either set.

```
>>> import dwave_networkx as dnx
>>> import dimod
>>> sampler = dimod.ExactSolver() # small testing sampler
>>> G = dnx.chimera_graph(1, 1, 4)
>>> G.remove_node(7) # to give a unique solution
>>> dnx.min_vertex_cover(G, sampler, lagrange=2.0)
[4, 5, 6]
```

Notes

Samplers by their nature may not return the optimal solution. This function does not attempt to confirm the quality of the returned sample.

References

https://en.wikipedia.org/wiki/Vertex_cover

https://en.wikipedia.org/wiki/Quadratic_unconstrained_binary_optimization

dwave_networkx.algorithms.cover.is_vertex_cover

is_vertex_cover (*G*, *vertex_cover*)

Determines whether the given set of vertices is a vertex cover of graph *G*.

A vertex cover is a set of vertices such that each edge of the graph is incident with at least one vertex in the set.

Parameters

- **G** (*NetworkX graph*) – The graph on which to check the vertex cover.
- **vertex_cover** – Iterable of nodes.

Returns **is_cover** – True if the given iterable forms a vertex cover.

Return type `bool`

Examples

This example checks two covers for a graph, *G*, of a single Chimera unit cell. The first uses the set of the four horizontal qubits, which do constitute a cover; the second set removes one node.

```
>>> import dwave_networkx as dnx
>>> G = dnx.chimera_graph(1, 1, 4)
>>> cover = [0, 1, 2, 3]
>>> dnx.is_vertex_cover(G, cover)
True
>>> cover = [0, 1, 2]
>>> dnx.is_vertex_cover(G, cover)
False
```

Elimination Ordering

Many algorithms for NP-hard problems are exponential in treewidth. However, finding a lower bound on treewidth is in itself NP-complete. [GD] describes a branch-and-bound algorithm for computing the treewidth of an undirected graph by searching in the space of *perfect elimination ordering* of vertices of the graph.

A *clique* of a graph is a fully-connected subset of vertices; that is, every pair of vertices in the clique share an edge. A *simplicial* vertex is one whose neighborhood induces a clique. A perfect elimination ordering is an ordering of vertices $1..n$ such that any vertex i is simplicial for the subset of vertices $i..n$.

<code>chimera_elimination_order(m[, n, t])</code>	Provides a variable elimination order for a Chimera graph.
<code>elimination_order_width(G, order)</code>	Calculates the width of the tree decomposition induced by a variable elimination order.
<code>is_almost_simplicial(G, n)</code>	Determines whether a node <i>n</i> in <i>G</i> is almost simplicial.
<code>is_simplicial(G, n)</code>	Determines whether a node <i>n</i> in <i>G</i> is simplicial.
<code>max_cardinality_heuristic(G)</code>	Computes an upper bound on the treewidth of graph <i>G</i> based on the max-cardinality heuristic for the elimination ordering.
<code>minor_min_width(G)</code>	Computes a lower bound for the treewidth of graph <i>G</i> .
<code>min_fill_heuristic(G)</code>	Computes an upper bound on the treewidth of graph <i>G</i> based on the min-fill heuristic for the elimination ordering.

Continued on next page

Table 5 – continued from previous page

<code>min_width_heuristic(G)</code>	Computes an upper bound on the treewidth of graph G based on the min-width heuristic for the elimination ordering.
<code>pegasus_elimination_order(n[, coordinates])</code>	Provides a variable elimination order for the Pegasus graph.
<code>treewidth_branch_and_bound(G[, ...])</code>	Computes the treewidth of graph G and a corresponding perfect elimination ordering.

`dwave_networkx.algorithms.elimination_ordering.chimera_elimination_order`

chimera_elimination_order ($m, n=None, t=None$)

Provides a variable elimination order for a Chimera graph.

A graph defined by `chimera_graph(m,n,t)` has treewidth $\max(m, n) * t$. This function outputs a variable elimination order inducing a tree decomposition of that width.

Parameters

- **m** (*int*) – Number of rows in the Chimera lattice.
- **n** (*int optional, default m*) – Number of columns in the Chimera lattice.
- **t** (*int optional, default 4*) – Size of the shore within each Chimera tile.

Returns order – An elimination order that induces the treewidth of `chimera_graph(m,n,t)`.

Return type `list`

Examples

```
>>> G = dnx.chimera_elimination_order(1, 1, 4) # a single Chimera tile
```

`dwave_networkx.algorithms.elimination_ordering.elimination_order_width`

elimination_order_width ($G, order$)

Calculates the width of the tree decomposition induced by a variable elimination order.

Parameters

- **G** (*NetworkX graph*) – The graph on which to compute the width of the tree decomposition.
- **order** (*list*) – The elimination order. Must be a list of all of the variables in G .

Returns treewidth – The width of the tree decomposition induced by order.

Return type `int`

Examples

This example computes the width of the tree decomposition for the K_4 complete graph induced by an elimination order found through the min-width heuristic.

```

>>> K_4 = nx.complete_graph(4)
>>> tw, order = dnx.min_width_heuristic(K_4)
>>> print(tw)
3
>>> dnx.elimination_order_width(K_4, order)
3

```

dwave_networkx.algorithms.elimination_ordering.is_almost_simplicial

is_almost_simplicial(*G*, *n*)

Determines whether a node *n* in *G* is almost simplicial.

Parameters

- **G** (*NetworkX graph*) – The graph on which to check whether node *n* is almost simplicial.
- **n** (*node*) – A node in graph *G*.

Returns **is_almost_simplicial** – True if all but one of its neighbors induce a clique

Return type `bool`

Examples

This example checks whether node 0 is simplicial or almost simplicial for a K_5 complete graph with one edge removed.

```

>>> K_5 = nx.complete_graph(5)
>>> K_5.remove_edge(1, 3)
>>> dnx.is_simplicial(K_5, 0)
False
>>> dnx.is_almost_simplicial(K_5, 0)
True

```

dwave_networkx.algorithms.elimination_ordering.is_simplicial

is_simplicial(*G*, *n*)

Determines whether a node *n* in *G* is simplicial.

Parameters

- **G** (*NetworkX graph*) – The graph on which to check whether node *n* is simplicial.
- **n** (*node*) – A node in graph *G*.

Returns **is_simplicial** – True if its neighbors form a clique.

Return type `bool`

Examples

This example checks whether node 0 is simplicial for two graphs: *G*, a single Chimera unit cell, which is bipartite, and K_5 , the K_5 complete graph.

```
>>> G = dnx.chimera_graph(1, 1, 4)
>>> K_5 = nx.complete_graph(5)
>>> dnx.is_simplicial(G, 0)
False
>>> dnx.is_simplicial(K_5, 0)
True
```

dwave_networkx.algorithms.elimination_ordering.max_cardinality_heuristic

max_cardinality_heuristic(*G*)

Computes an upper bound on the treewidth of graph *G* based on the max-cardinality heuristic for the elimination ordering.

Parameters *G* (*NetworkX graph*) – The graph on which to compute an upper bound for the treewidth.

Returns

- **treewidth_upper_bound** (*int*) – An upper bound on the treewidth of the graph *G*.
- **order** (*list*) – An elimination order that induces the treewidth.

Examples

This example computes an upper bound for the treewidth of the K_4 complete graph.

```
>>> K_4 = nx.complete_graph(4)
>>> tw, order = dnx.max_cardinality_heuristic(K_4)
```

References

Based on the algorithm presented in [GD]

dwave_networkx.algorithms.elimination_ordering.minor_min_width

minor_min_width(*G*)

Computes a lower bound for the treewidth of graph *G*.

Parameters *G* (*NetworkX graph*) – The graph on which to compute a lower bound on the treewidth.

Returns *lb* – A lower bound on the treewidth.

Return type *int*

Examples

This example computes a lower bound for the treewidth of the K_7 complete graph.

```
>>> K_7 = nx.complete_graph(7)
>>> dnx.minor_min_width(K_7)
6
```

References

Based on the algorithm presented in [GD]

`dwave_networkx.algorithms.elimination_ordering.min_fill_heuristic`

`min_fill_heuristic(G)`

Computes an upper bound on the treewidth of graph G based on the min-fill heuristic for the elimination ordering.

Parameters G (*NetworkX graph*) – The graph on which to compute an upper bound for the treewidth.

Returns

- **treewidth_upper_bound** (*int*) – An upper bound on the treewidth of the graph G .
- **order** (*list*) – An elimination order that induces the treewidth.

Examples

This example computes an upper bound for the treewidth of the K_4 complete graph.

```
>>> K_4 = nx.complete_graph(4)
>>> tw, order = dnx.min_fill_heuristic(K_4)
```

References

Based on the algorithm presented in [GD]

`dwave_networkx.algorithms.elimination_ordering.min_width_heuristic`

`min_width_heuristic(G)`

Computes an upper bound on the treewidth of graph G based on the min-width heuristic for the elimination ordering.

Parameters G (*NetworkX graph*) – The graph on which to compute an upper bound for the treewidth.

Returns

- **treewidth_upper_bound** (*int*) – An upper bound on the treewidth of the graph G .
- **order** (*list*) – An elimination order that induces the treewidth.

Examples

This example computes an upper bound for the treewidth of the K_4 complete graph.

```
>>> K_4 = nx.complete_graph(4)
>>> tw, order = dnx.min_width_heuristic(K_4)
```

References

Based on the algorithm presented in [GD]

`dwave_networkx.algorithms.elimination_ordering.pegasus_elimination_order`

pegasus_elimination_order (*n*, *coordinates=False*)

Provides a variable elimination order for the Pegasus graph.

The treewidth of a Pegasus graph $P(n)$ is lower-bounded by $12n-11$ and upper bounded by $12-4$ [brrr] .

Simple pegasus variable elimination order rules:

- eliminate vertical qubits, one column at a time
- eliminate horizontal qubits in each column once their adjacent vertical qubits have been eliminated

Parameters

- **n** (*int*) – The size parameter for the Pegasus lattice.
- **coordinates** (*bool*, *optional* (*default False*)) – If True, the elimination order is given in terms of 4-term Pegasus coordinates, otherwise given in linear indices.

Returns **order** – An elimination order that provides an upper bound on the treewidth.

Return type *list*

`dwave_networkx.algorithms.elimination_ordering.treewidth_branch_and_bound`

treewidth_branch_and_bound (*G*, *elimination_order=None*, *treewidth_upperbound=None*)

Computes the treewidth of graph G and a corresponding perfect elimination ordering.

Algorithm based on [GD].

Parameters

- **G** (*NetworkX graph*) – The graph on which to compute the treewidth and perfect elimination ordering.
- **elimination_order** (*list* (*optional*, *Default None*)) – An elimination order used as an initial best-known order. If a good order is provided, it may speed up computation. If not provided, the initial order is generated using the min-fill heuristic.
- **treewidth_upperbound** (*int* (*optional*, *Default None*)) – An upper bound on the treewidth. Note that using this parameter can result in no returned order.

Returns

- **treewidth** (*int*) – The treewidth of graph G.
- **order** (*list*) – An elimination order that induces the treewidth.

Examples

This example computes the treewidth for the K_7 complete graph using an optionally provided elimination order (a sequential ordering of the nodes, arbitrarily chosen).


```
>>> K_7 = nx.complete_graph(7)
>>> dnx.treewidth_branch_and_bound(K_7, [0, 1, 2, 3, 4, 5, 6])
(6, [0, 1, 2, 3, 4, 5, 6])
```

References

References

Markov Networks

<code>sample_markov_network(MN[, sampler, ...])</code>	Samples from a markov network using the provided sampler.
<code>markov_network_bqm(MN)</code>	Construct a binary quadratic model for a markov network.

dwave_networkx.algorithms.markov.sample_markov_network

sample_markov_network (*MN*, *sampler=None*, *fixed_variables=None*, *return_sampleset=False*, ***sampler_args*)

Samples from a markov network using the provided sampler.

Parameters

- **G** (*NetworkX graph*) – A Markov Network as returned by `markov_network()`
- **sampler** – A binary quadratic model sampler. A sampler is a process that samples from low energy states in models defined by an Ising equation or a Quadratic Unconstrained Binary Optimization Problem (QUBO). A sampler is expected to have a ‘sample_qubo’ and ‘sample_ising’ method. A sampler is expected to return an iterable of samples, in order of increasing energy. If no sampler is provided, one must be provided using the `set_default_sampler` function.
- **fixed_variables** (*dict*) – A dictionary of variable assignments to be fixed in the markov network.
- **return_sampleset** (*bool (optional, default=False)*) – If True, returns a `dimod.SampleSet` rather than a list of samples.
- ****sampler_args** – Additional keyword parameters are passed to the sampler.

Returns `samples` – A list of samples ordered from low-to-high energy or a sample set.

Return type `list[dict]/dimod.SampleSet`

Examples

```
>>> import dimod
...
>>> potentials = {('a', 'b'): {(0, 0): -1,
...                             (0, 1): .5,
...                             (1, 0): .5,
...                             (1, 1): 2}}
...
>>> MN = dnx.markov_network(potentials)
```

(continues on next page)

(continued from previous page)

```
>>> sampler = dimod.ExactSolver()
>>> samples = dnx.sample_markov_network(MN, sampler)
>>> samples[0]      # doctest: +SKIP
{'a': 0, 'b': 0}
```

```
>>> import dimod
...
>>> potentials = {('a', 'b'): {(0, 0): -1,
...                             (0, 1): .5,
...                             (1, 0): .5,
...                             (1, 1): 2}}
>>> MN = dnx.markov_network(potentials)
>>> sampler = dimod.ExactSolver()
>>> samples = dnx.sample_markov_network(MN, sampler, return_sampleset=True)
>>> samples.first      # doctest: +SKIP
Sample(sample={'a': 0, 'b': 0}, energy=-1.0, num_occurrences=1)
```

```
>>> import dimod
...
>>> potentials = {('a', 'b'): {(0, 0): -1,
...                             (0, 1): .5,
...                             (1, 0): .5,
...                             (1, 1): 2},
...               ('b', 'c'): {(0, 0): -9,
...                             (0, 1): 1.2,
...                             (1, 0): 7.2,
...                             (1, 1): 5}}
>>> MN = dnx.markov_network(potentials)
>>> sampler = dimod.ExactSolver()
>>> samples = dnx.sample_markov_network(MN, sampler, fixed_variables={'b': 0})
>>> samples[0]      # doctest: +SKIP
{'a': 0, 'c': 0, 'b': 0}
```

Notes

Samplers by their nature may not return the optimal solution. This function does not attempt to confirm the quality of the returned sample.

dwave_networkx.algorithms.markov.markov_network_bqm

markov_network_bqm (MN)

Construct a binary quadratic model for a markov network.

Parameters *G* (*NetworkX graph*) – A Markov Network as returned by *markov_network()*

Returns *bqm* – A binary quadratic model.

Return type `dimod.BinaryQuadraticModel`

Matching

A matching is a subset of graph edges in which no vertex occurs more than once.

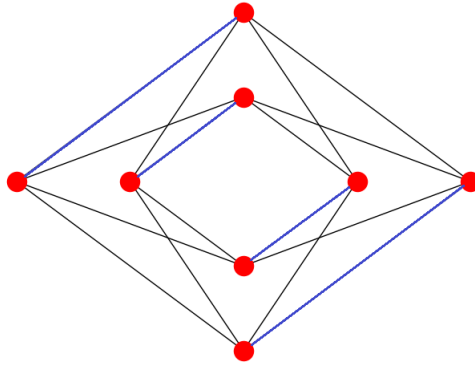


Fig. 3: A matching for a Chimera unit cell: no vertex is incident to more than one edge in the set of blue edges

<code>min_maximal_matching(G[, sampler])</code>	Returns an approximate minimum maximal matching.
<code>is_matching(edges)</code>	Determines whether the given set of edges is a matching.
<code>is_maximal_matching(G, matching)</code>	Determines whether the given set of edges is a maximal matching.

`dwave_networkx.algorithms.matching.min_maximal_matching`

`min_maximal_matching` (*G*, *sampler=None*, ***sampler_args*)

Returns an approximate minimum maximal matching.

Defines a QUBO with ground states corresponding to a minimum maximal matching and uses the sampler to sample from it.

A matching is a subset of edges in which no node occurs more than once. A maximal matching is one in which no edges from *G* can be added without violating the matching rule. A minimum maximal matching is the smallest maximal matching for *G*.

Parameters

- **G** (*NetworkX graph*) – The graph on which to find a minimum maximal matching.
- **sampler** – A binary quadratic model sampler. A sampler is a process that samples from low energy states in models defined by an Ising equation or a Quadratic Unconstrained Binary Optimization Problem (QUBO). A sampler is expected to have a ‘sample_qubo’ and ‘sample_ising’ method. A sampler is expected to return an iterable of samples, in order of increasing energy. If no sampler is provided, one must be provided using the `set_default_sampler` function.
- **sampler_args** – Additional keyword parameters are passed to the sampler.

Returns `matching` – A minimum maximal matching of the graph.

Return type `set`

Example

This example uses a sampler from `dimod` to find a minimum maximal matching for a Chimera unit cell.

```
>>> import dimod
>>> sampler = dimod.ExactSolver()
>>> G = dnx.chimera_graph(1, 1, 4)
>>> matching = dnx.min_maximal_matching(G, sampler)
```

dwave_networkx.algorithms.matching.is_matching

is_matching (*edges*)

Determines whether the given set of edges is a matching.

A matching is a subset of edges in which no node occurs more than once.

Parameters **edges** (*iterable*) – A iterable of edges.

Returns **is_matching** – True if the given edges are a matching.

Return type `bool`

Example

This example checks two sets of edges, both derived from a single Chimera unit cell, for a matching. Because every node in a Chimera unit cell connects to four other nodes in the cell, the first set, which contains all the edges, repeats each node 4 times; the second is a subset of those edges found using the `min_maximal_matching()` function.

```
>>> import dwave_networkx as dnx
>>> G = dnx.chimera_graph(1, 1, 4)
>>> dnx.is_matching(G.edges())
False
>>> dnx.is_matching({(0, 4), (1, 5), (2, 7), (3, 6)})
True
```

dwave_networkx.algorithms.matching.is_maximal_matching

is_maximal_matching (*G*, *matching*)

Determines whether the given set of edges is a maximal matching.

A matching is a subset of edges in which no node occurs more than once. The cardinality of a matching is the number of matched edges. A maximal matching is one where one cannot add any more edges without violating the matching rule.

Parameters

- **G** (*NetworkX graph*) – The graph on which to check the maximal matching.
- **edges** (*iterable*) – A iterable of edges.

Returns **is_matching** – True if the given edges are a maximal matching.

Return type `bool`

Example

This example checks two sets of edges, both derived from a single Chimera unit cell, for a matching. The first set (a matching) is a subset of the second, which was found using the `min_maximal_matching()` function.

```
>>> import dwave_networkx as dnx
>>> G = dnx.chimera_graph(1, 1, 4)
>>> dnx.is_matching({(0, 4), (2, 7)})
True
>>> dnx.is_maximal_matching(G, {(0, 4), (2, 7)})
```

(continues on next page)

(continued from previous page)

```
False
>>> dnx.is_maximal_matching(G, {(0, 4), (1, 5), (2, 7), (3, 6)})
True
```

Maximum Cut

A maximum cut is a subset of a graph's vertices such that the number of edges between this subset and the remaining vertices is as large as possible.

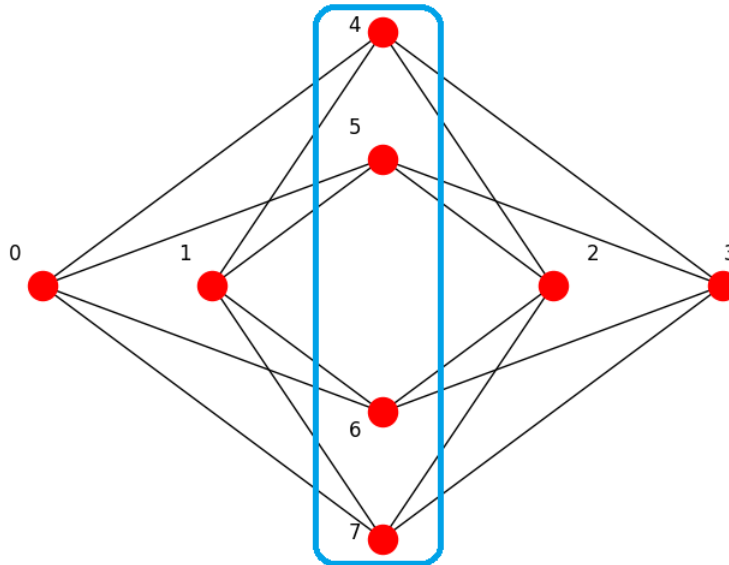


Fig. 4: Maximum cut for a Chimera unit cell: the blue line around the subset of nodes {4, 5, 6, 7} cuts 16 edges; adding or removing a node decreases the number of edges between the two complementary subsets of the graph.

<code>maximum_cut(G[, sampler])</code>	Returns an approximate maximum cut.
<code>weighted_maximum_cut(G[, sampler])</code>	Returns an approximate weighted maximum cut.

`dwave_networkx.algorithms.max_cut.maximum_cut`

maximum_cut (*G*, *sampler=None*, ***sampler_args*)

Returns an approximate maximum cut.

Defines an Ising problem with ground states corresponding to a maximum cut and uses the sampler to sample from it.

A maximum cut is a subset *S* of the vertices of *G* such that the number of edges between *S* and the complementary subset is as large as possible.

Parameters

- **G** (*NetworkX graph*) – The graph on which to find a maximum cut.
- **sampler** – A binary quadratic model sampler. A sampler is a process that samples from low energy states in models defined by an Ising equation or a Quadratic Unconstrained Binary Optimization Problem (QUBO). A sampler is expected to have a ‘sample_qubo’

and ‘sample_ising’ method. A sampler is expected to return an iterable of samples, in order of increasing energy. If no sampler is provided, one must be provided using the `set_default_sampler` function.

- **sampler_args** – Additional keyword parameters are passed to the sampler.

Returns **S** – A maximum cut of G.

Return type `set`

Example

This example uses a sampler from `dimod` to find a maximum cut for a graph of a Chimera unit cell created using the `chimera_graph()` function.

```
>>> import dimod
...
>>> sampler = dimod.SimulatedAnnealingSampler()
>>> G = dnx.chimera_graph(1, 1, 4)
>>> cut = dnx.maximum_cut(G, sampler)
```

Notes

Samplers by their nature may not return the optimal solution. This function does not attempt to confirm the quality of the returned sample.

`dwave_networkx.algorithms.max_cut.weighted_maximum_cut`

weighted_maximum_cut (*G*, *sampler=None*, ***sampler_args*)

Returns an approximate weighted maximum cut.

Defines an Ising problem with ground states corresponding to a weighted maximum cut and uses the sampler to sample from it.

A weighted maximum cut is a subset *S* of the vertices of *G* that maximizes the sum of the edge weights between *S* and its complementary subset.

Parameters

- **G** (*NetworkX graph*) – The graph on which to find a weighted maximum cut. Each edge in *G* should have a numeric *weight* attribute.
- **sampler** – A binary quadratic model sampler. A sampler is a process that samples from low energy states in models defined by an Ising equation or a Quadratic Unconstrained Binary Optimization Problem (QUBO). A sampler is expected to have a ‘sample_qubo’ and ‘sample_ising’ method. A sampler is expected to return an iterable of samples, in order of increasing energy. If no sampler is provided, one must be provided using the `set_default_sampler` function.
- **sampler_args** – Additional keyword parameters are passed to the sampler.

Returns **S** – A maximum cut of G.

Return type `set`

Notes

Samplers by their nature may not return the optimal solution. This function does not attempt to confirm the quality of the returned sample.

Independent Set

An independent set is a set of a graph's vertices with no edge connecting any of its member pairs.

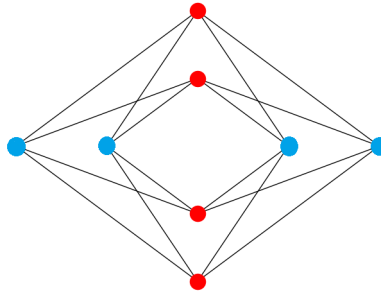


Fig. 5: Independent sets for a Chimera unit cell: the nodes of both the blue set of vertices (the horizontal tile of the Chimera unit cell) and the red set (vertical tile) are independent sets of the graph, with no blue node adjacent to another blue node and likewise for red nodes.

<code>maximum_weighted_independent_set(G[, ...])</code>	Returns an approximate maximum weighted independent set.
<code>maximum_independent_set(G[, sampler, lagrange])</code>	Returns an approximate maximum independent set.
<code>is_independent_set(G, indep_nodes)</code>	Determines whether the given nodes form an independent set.

`dwave_networkx.maximum_weighted_independent_set`

`maximum_weighted_independent_set(G, weight=None, sampler=None, lagrange=2.0, **sampler_args)`

Returns an approximate maximum weighted independent set.

Defines a QUBO with ground states corresponding to a maximum weighted independent set and uses the sampler to sample from it.

An independent set is a set of nodes such that the subgraph of G induced by these nodes contains no edges. A maximum independent set is an independent set of maximum total node weight.

Parameters

- **G** (*NetworkX graph*) – The graph on which to find a maximum cut weighted independent set.
- **weight** (*string, optional (default None)*) – If `None`, every node has equal weight. If a string, use this node attribute as the node weight. A node without this attribute is assumed to have max weight.
- **sampler** – A binary quadratic model sampler. A sampler is a process that samples from low energy states in models defined by an Ising equation or a Quadratic Unconstrained

Binary Optimization Problem (QUBO). A sampler is expected to have a ‘sample_qubo’ and ‘sample_ising’ method. A sampler is expected to return an iterable of samples, in order of increasing energy. If no sampler is provided, one must be provided using the `set_default_sampler` function.

- **lagrange** (*optional (default 2)*) – Lagrange parameter to weight constraints (no edges within set) versus objective (largest set possible).
- **sampler_args** – Additional keyword parameters are passed to the sampler.

Returns indep_nodes – List of nodes that form a maximum weighted independent set, as determined by the given sampler.

Return type `list`

Notes

Samplers by their nature may not return the optimal solution. This function does not attempt to confirm the quality of the returned sample.

References

[Independent Set on Wikipedia](#)

[QUBO on Wikipedia](#)

`dwave_networkx.maximum_independent_set`

maximum_independent_set (*G, sampler=None, lagrange=2.0, **sampler_args*)

Returns an approximate maximum independent set.

Defines a QUBO with ground states corresponding to a maximum independent set and uses the sampler to sample from it.

An independent set is a set of nodes such that the subgraph of *G* induced by these nodes contains no edges. A maximum independent set is an independent set of largest possible size.

Parameters

- **G** (*NetworkX graph*) – The graph on which to find a maximum cut independent set.
- **sampler** – A binary quadratic model sampler. A sampler is a process that samples from low energy states in models defined by an Ising equation or a Quadratic Unconstrained Binary Optimization Problem (QUBO). A sampler is expected to have a ‘sample_qubo’ and ‘sample_ising’ method. A sampler is expected to return an iterable of samples, in order of increasing energy. If no sampler is provided, one must be provided using the `set_default_sampler` function.
- **lagrange** (*optional (default 2)*) – Lagrange parameter to weight constraints (no edges within set) versus objective (largest set possible).
- **sampler_args** – Additional keyword parameters are passed to the sampler.

Returns indep_nodes – List of nodes that form a maximum independent set, as determined by the given sampler.

Return type `list`

Example

This example uses a sampler from `dimod` to find a maximum independent set for a graph of a Chimera unit cell created using the `chimera_graph()` function.

```
>>> import dimod
>>> sampler = dimod.SimulatedAnnealingSampler()
>>> G = dnx.chimera_graph(1, 1, 4)
>>> indep_nodes = dnx.maximum_independent_set(G, sampler)
```

Notes

Samplers by their nature may not return the optimal solution. This function does not attempt to confirm the quality of the returned sample.

References

[Independent Set on Wikipedia](#)

[QUBO on Wikipedia](#)

`dwave_networkx.is_independent_set`

`is_independent_set` (*G*, *indep_nodes*)

Determines whether the given nodes form an independent set.

An independent set is a set of nodes such that the subgraph of *G* induced by these nodes contains no edges.

Parameters

- **G** (*NetworkX graph*) – The graph on which to check the independent set.
- **indep_nodes** (*list*) – List of nodes that form a maximum independent set, as determined by the given sampler.

Returns `is_independent` – True if `indep_nodes` form an independent set.

Return type `bool`

Example

This example checks two sets of nodes, both derived from a single Chimera unit cell, for an independent set. The first set is the horizontal tile's nodes; the second has nodes from the horizontal and vertical tiles.

```
>>> import dwave_networkx as dnx
>>> G = dnx.chimera_graph(1, 1, 4)
>>> dnx.is_independent_set(G, [0, 1, 2, 3])
True
>>> dnx.is_independent_set(G, [0, 4])
False
```

Helper Functions

<code>maximum_weighted_independent_set_qubo(G</code>	Return the QUBO with ground states corresponding to a maximum weighted independent set.
--	---

`dwave_networkx.algorithms.independent_set.maximum_weighted_independent_set_qubo`

maximum_weighted_independent_set_qubo (*G*, *weight=None*, *lagrange=2.0*)

Return the QUBO with ground states corresponding to a maximum weighted independent set.

Parameters

- **G** (*NetworkX graph*) –
- **weight** (*string, optional (default None)*) – If None, every node has equal weight. If a string, use this node attribute as the node weight. A node without this attribute is assumed to have max weight.
- **lagrange** (*optional (default 2)*) – Lagrange parameter to weight constraints (no edges within set) versus objective (largest set possible).

Returns QUBO – The QUBO with ground states corresponding to a maximum weighted independent set.

Return type `dict`

Examples

```

>>> from dwave_networkx.algorithms.independent_set import maximum_weighted_
      ↪ independent_set_qubo
...
>>> G = nx.path_graph(3)
>>> Q = maximum_weighted_independent_set_qubo(G, weight='weight', lagrange=2.0)
>>> Q[(0, 0)]
-1.0
>>> Q[(1, 1)]
-1.0
>>> Q[(0, 1)]
2.0

```

Social

A signed social network graph is a graph whose signed edges represent friendly/hostile interactions between vertices.

<code>structural_imbalance(S[, sampler])</code>	Returns an approximate set of frustrated edges and a bi-coloring.
<code>structural_imbalance_ising(S)</code>	Construct the Ising problem to calculate the structural imbalance of a signed social network.

`dwave_networkx.algorithms.social.structural_imbalance`

structural_imbalance (*S*, *sampler=None*, ***sampler_args*)

Returns an approximate set of frustrated edges and a bicoloring.

A signed social network graph is a graph whose signed edges represent friendly/hostile interactions between

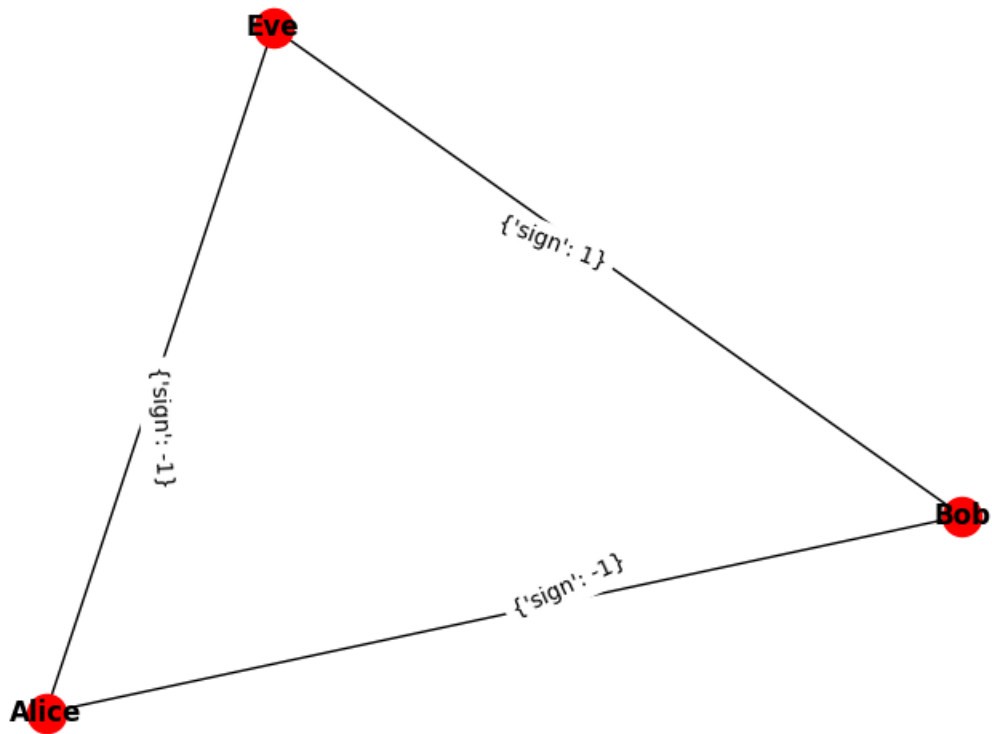


Fig. 6: A signed social graph for three nodes, where Eve and Bob are friendly with each other and hostile to Alice. This network is balanced because it can be cleanly divided into two subsets, {Bob, Eve} and {Alice}, with friendly relations within each subset and only hostile relations between the subsets.

nodes. A signed social network is considered balanced if it can be cleanly divided into two factions, where all relations within a faction are friendly, and all relations between factions are hostile. The measure of imbalance or frustration is the minimum number of edges that violate this rule.

Parameters

- **S** (*NetworkX graph*) – A social graph on which each edge has a ‘sign’ attribute with a numeric value.
- **sampler** – A binary quadratic model sampler. A sampler is a process that samples from low energy states in models defined by an Ising equation or a Quadratic Unconstrained Binary Optimization Problem (QUBO). A sampler is expected to have a ‘sample_qubo’ and ‘sample_ising’ method. A sampler is expected to return an iterable of samples, in order of increasing energy. If no sampler is provided, one must be provided using the `set_default_sampler` function.
- **sampler_args** – Additional keyword parameters are passed to the sampler.

Returns

- **frustrated_edges** (*dict*) – A dictionary of the edges that violate the edge sign. The imbalance of the network is the length of `frustrated_edges`.
- **colors** (*dict*) – A bicoloring of the nodes into two factions.

Raises `ValueError` – If any edge does not have a ‘sign’ attribute.

Examples

```
>>> import dimod
>>> sampler = dimod.ExactSolver()
>>> S = nx.Graph()
>>> S.add_edge('Alice', 'Bob', sign=1) # Alice and Bob are friendly
>>> S.add_edge('Alice', 'Eve', sign=-1) # Alice and Eve are hostile
>>> S.add_edge('Bob', 'Eve', sign=-1) # Bob and Eve are hostile
>>> frustrated_edges, colors = dnx.structural_imbalance(S, sampler)
>>> print(frustrated_edges)
{}
>>> print(colors) # doctest: +SKIP
{'Alice': 0, 'Bob': 0, 'Eve': 1}
>>> S.add_edge('Ted', 'Bob', sign=1) # Ted is friendly with all
>>> S.add_edge('Ted', 'Alice', sign=1)
>>> S.add_edge('Ted', 'Eve', sign=1)
>>> frustrated_edges, colors = dnx.structural_imbalance(S, sampler)
>>> print(frustrated_edges) # doctest: +SKIP
{('Ted', 'Eve'): {'sign': 1}}
>>> print(colors) # doctest: +SKIP
{'Bob': 1, 'Ted': 1, 'Alice': 1, 'Eve': 0}
```

Notes

Samplers by their nature may not return the optimal solution. This function does not attempt to confirm the quality of the returned sample.

References

[Ising model on Wikipedia](#)

dwave_networkx.algorithms.social.structural_imbalance_ising

structural_imbalance_ising(*S*)

Construct the Ising problem to calculate the structural imbalance of a signed social network.

A signed social network graph is a graph whose signed edges represent friendly/hostile interactions between nodes. A signed social network is considered balanced if it can be cleanly divided into two factions, where all relations within a faction are friendly, and all relations between factions are hostile. The measure of imbalance or frustration is the minimum number of edges that violate this rule.

Parameters *S* (*NetworkX graph*) – A social graph on which each edge has a ‘sign’ attribute with a numeric value.

Returns

- **h** (*dict*) – The linear biases of the Ising problem. Each variable in the Ising problem represent a node in the signed social network. The solution that minimized the Ising problem will assign each variable a value, either -1 or 1. This bi-coloring defines the factions.
- **J** (*dict*) – The quadratic biases of the Ising problem.

Raises `ValueError` – If any edge does not have a ‘sign’ attribute.

Examples

```
>>> import dimod
>>> from dwave_networkx.algorithms.social import structural_imbalance_ising
...
>>> S = nx.Graph()
>>> S.add_edge('Alice', 'Bob', sign=1) # Alice and Bob are friendly
>>> S.add_edge('Alice', 'Eve', sign=-1) # Alice and Eve are hostile
>>> S.add_edge('Bob', 'Eve', sign=-1) # Bob and Eve are hostile
...
>>> h, J = structural_imbalance_ising(S)
>>> h # doctest: +SKIP
{'Alice': 0.0, 'Bob': 0.0, 'Eve': 0.0}
>>> J # doctest: +SKIP
{('Alice', 'Bob'): -1.0, ('Alice', 'Eve'): 1.0, ('Bob', 'Eve'): 1.0}
```

Traveling Salesperson

A traveling salesperson route is an ordering of the vertices in a complete weighted graph.

<code>traveling_salesperson(G[, sampler, ...])</code>	Returns an approximate minimum traveling salesperson route.
<code>traveling_salesperson_qubo(G[, lagrange, weight])</code>	Return the QUBO with ground states corresponding to a minimum TSP route.

dwave_networkx.algorithms.tsp.traveling_salesperson

traveling_salesperson (*G*, *sampler=None*, *lagrange=None*, *weight='weight'*, *start=None*, ***sampler_args*)

Returns an approximate minimum traveling salesperson route.

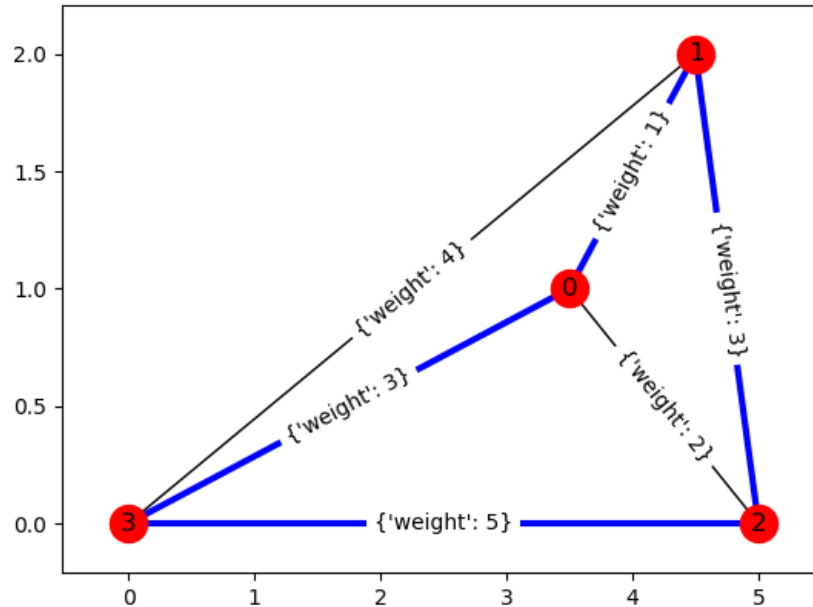


Fig. 7: A traveling salesperson route of [2, 1, 0, 3].

Defines a QUBO with ground states corresponding to the minimum routes and uses the sampler to sample from it.

A route is a cycle in the graph that reaches each node exactly once. A minimum route is a route with the smallest total edge weight.

Parameters

- **G** (*NetworkX graph*) – The graph on which to find a minimum traveling salesperson route. This should be a complete graph with non-zero weights on every edge.
- **sampler** – A binary quadratic model sampler. A sampler is a process that samples from low energy states in models defined by an Ising equation or a Quadratic Unconstrained Binary Optimization Problem (QUBO). A sampler is expected to have a ‘sample_qubo’ and ‘sample_ising’ method. A sampler is expected to return an iterable of samples, in order of increasing energy. If no sampler is provided, one must be provided using the `set_default_sampler` function.
- **lagrange** (*number, optional (default None)*) – Lagrange parameter to weight constraints (visit every city once) versus objective (shortest distance route).
- **weight** (*optional (default 'weight')*) – The name of the edge attribute containing the weight.
- **start** (*node, optional*) – If provided, the route will begin at *start*.
- **sampler_args** – Additional keyword parameters are passed to the sampler.

Returns `route` – List of nodes in order to be visited on a route

Return type `list`

Examples

```
>>> import dimod
...
>>> G = nx.Graph()
>>> G.add_weighted_edges_from([(0, 1, .1), (0, 2, .5), (0, 3, .1), (1, 2, .1),
...                             (1, 3, .5), (2, 3, .1)])
>>> dnx.traveling_salesperson(G, dimod.ExactSolver(), start=0) # doctest: +SKIP
[0, 1, 2, 3]
```

Notes

Samplers by their nature may not return the optimal solution. This function does not attempt to confirm the quality of the returned sample.

dwave_networkx.algorithms.tsp.traveling_salesperson_qubo

traveling_salesperson_qubo (*G*, *lagrange=None*, *weight='weight'*)

Return the QUBO with ground states corresponding to a minimum TSP route.

If $|G|$ is the number of nodes in the graph, the resulting qubo will have:

- $|G|^2$ variables/nodes
- $2|G|^2(|G| - 1)$ interactions/edges

Parameters

- **G** (*NetworkX graph*) – A complete graph in which each edge has a attribute giving its weight.
- **lagrange** (*number, optional (default None)*) – Lagrange parameter to weight constraints (no edges within set) versus objective (largest set possible).
- **weight** (*optional (default 'weight')*) – The name of the edge attribute containing the weight.

Returns QUBO – The QUBO with ground states corresponding to a minimum travelling salesperson route. The QUBO variables are labelled (*c*, *t*) where *c* is a node in *G* and *t* is the time index. For instance, if (*a*, 0) is 1 in the ground state, that means the node ‘a’ is visited first.

Return type dict

Drawing

Tools to visualize topologies of D-Wave QPUs and weighted graph problems on them.

Note: Some functionality requires NumPy and/or Matplotlib.

Chimera Graph Functions

Tools to visualize Chimera lattices and weighted graph problems on them.

<code>chimera_layout(G[, scale, center, dim])</code>	Positions the nodes of graph G in a Chimera cross topology.
<code>draw_chimera(G, **kwargs)</code>	Draws graph G in a Chimera cross topology.

dwave_networkx.drawing.chimera_layout.chimera_layout

chimera_layout (*G*, *scale=1.0*, *center=None*, *dim=2*)

Positions the nodes of graph G in a Chimera cross topology.

NumPy (<http://scipy.org>) is required for this function.

Parameters

- **G** (*NetworkX graph*) – Should be a Chimera graph or a subgraph of a Chimera graph. If every node in G has a *chimera_index* attribute, those are used to place the nodes. Otherwise makes a best-effort attempt to find positions.
- **scale** (*float (default 1.)*) – Scale factor. When *scale = 1*, all positions fit within $[0, 1]$ on the x-axis and $[-1, 0]$ on the y-axis.
- **center** (*None or array (default None)*) – Coordinates of the top left corner.
- **dim** (*int (default 2)*) – Number of dimensions. When *dim > 2*, all extra dimensions are set to 0.

Returns *pos* – A dictionary of positions keyed by node.

Return type *dict*

Examples

```
>>> G = dnx.chimera_graph(1)
>>> pos = dnx.chimera_layout(G)
```

dwave_networkx.drawing.chimera_layout.draw_chimera

draw_chimera (*G*, ***kwargs*)

Draws graph G in a Chimera cross topology.

If *linear_biases* and/or *quadratic_biases* are provided, these are visualized on the plot.

Parameters

- **G** (*NetworkX graph*) – Should be a Chimera graph or a subgraph of a Chimera graph.
- **linear_biases** (*dict (optional, default {})*) – A dict of biases associated with each node in G. Should be of form $\{\text{node: bias, ...}\}$. Each bias should be numeric.
- **quadratic_biases** (*dict (optional, default {})*) – A dict of biases associated with each edge in G. Should be of form $\{\text{edge: bias, ...}\}$. Each bias should be numeric. Self-loop edges (i.e., $i = j$) are treated as linear biases.
- **kwargs** (*optional keywords*) – See `networkx.draw_networkx()` for a description of optional keywords, with the exception of the *pos* parameter which is not used by this function. If *linear_biases* or *quadratic_biases* are provided, any provided *node_color* or *edge_color* arguments are ignored.

Examples

```
>>> # Plot 2x2 Chimera unit cells
>>> import networkx as nx
>>> import dwave_networkx as dnx
>>> import matplotlib.pyplot as plt # doctest: +SKIP
>>> G = dnx.chimera_graph(2, 2, 4)
>>> dnx.draw_chimera(G) # doctest: +SKIP
>>> plt.show() # doctest: +SKIP
```

Example

This example uses the `chimera_layout()` function to show the positions of nodes of a simple 5-node NetworkX graph in a Chimera lattice. It then uses the `chimera_graph()` and `draw_chimera()` functions to display those positions on a Chimera unit cell.

```
>>> import networkx as nx
>>> import dwave_networkx as dnx
>>> import matplotlib.pyplot as plt
>>> H = nx.Graph()
>>> H.add_nodes_from([0, 4, 5, 6, 7])
>>> H.add_edges_from([(0, 4), (0, 5), (0, 6), (0, 7)])
>>> pos=dnx.chimera_layout(H)
>>> pos
{0: array([ 0. , -0.5]),
 4: array([ 0.5,  0. ]),
 5: array([ 0.5 , -0.25]),
 6: array([ 0.5 , -0.75]),
 7: array([ 0.5, -1. ])}
>>> # Show graph H on a Chimera unit cell
>>> plt.ion()
>>> G=dnx.chimera_graph(1, 1, 4) # Draw a Chimera unit cell
>>> dnx.draw_chimera(G)
>>> dnx.draw_chimera(H, node_color='b', node_shape='*', style='dashed', edge_color='b
↔', width=3)
>>> # matplotlib commands to add labels to graphic not shown
```

Pegasus Graph Functions

Tools to visualize Pegasus lattices and weighted graph problems on them.

<code>draw_pegasus(G[, crosses])</code>	Draws graph G in a Pegasus topology.
<code>draw_pegasus_embedding(G, *args, **kwargs)</code>	Draws an embedding onto Pegasus graph G.
<code>pegasus_layout(G[, scale, center, dim, crosses])</code>	Positions the nodes of graph G in a Pegasus topology.
<code>pegasus_node_placer_2d(G[, scale, center, ...])</code>	Generates a function to convert Pegasus indices to plot-table coordinates.

`dwave_networkx.drawing.pegasus_layout.draw_pegasus`

`draw_pegasus` (*G*, *crosses=False*, ***kwargs*)

Draws graph G in a Pegasus topology.

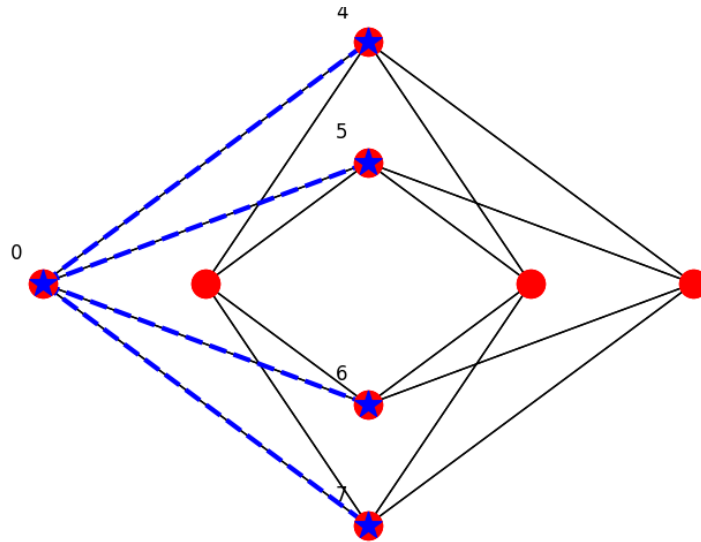


Fig. 8: Graph H (blue) overlaid on a Chimera unit cell (red nodes and black edges).

If `linear_biases` and/or `quadratic_biases` are provided, these are visualized on the plot.

Parameters

- **G** (*NetworkX graph*) – A Pegasus graph or a subgraph of a Pegasus graph, as produced by the `dwave_networkx.pegasus_graph()` function.
- **linear_biases** (*dict (optional, default {})*) – Biases as a dict, of form `{node: bias, ...}`, where keys are nodes in G and biases are numeric.
- **quadratic_biases** (*dict (optional, default {})*) – Biases as a dict, of form `{edge: bias, ...}`, where keys are edges in G and biases are numeric. Self-loop edges (i.e., $i = j$) are treated as linear biases.
- **crosses** (*boolean (optional, default False)*) – If True, $K_{4,4}$ subgraphs are shown in a cross rather than L configuration. Ignored if G is defined with `nice_coordinates=True`.
- **kwargs** (*optional keywords*) – See `networkx.draw_networkx()` for a description of optional keywords, with the exception of the `pos` parameter, which is not used by this function. If `linear_biases` or `quadratic_biases` are provided, any provided `node_color` or `edge_color` arguments are ignored.

Examples

This example plots a Pegasus graph with size parameter 2.

```
>>> import networkx as nx
>>> import dwave_networkx as dnx
>>> import matplotlib.pyplot as plt    # doctest: +SKIP
>>> G = dnx.pegasus_graph(2)
>>> dnx.draw_pegasus(G)                # doctest: +SKIP
>>> plt.show()                          # doctest: +SKIP
```

dwave_networkx.drawing.pegasus_layout.draw_pegasus_embedding

draw_pegasus_embedding (*G*, *args, **kwargs)

Draws an embedding onto Pegasus graph *G*.

Parameters

- **G** (*NetworkX graph*) – A Pegasus graph or a subgraph of a Pegasus graph, as produced by the `dwave_networkx.pegasus_graph()` function.
- **emb** (*dict*) – Chains, as a dict of form {qubit: chain, ...}, where qubits are nodes in *G* and chains are iterables of qubit labels.
- **embedded_graph** (*NetworkX graph (optional, default None)*) – A graph that contains all keys of *emb* as nodes. If specified, edges of *G* are considered interactions if and only if (1) they exist between two chains of *emb* and (2) their keys are connected by an edge in this graph. If given, only couplers between chains based on this graph are displayed.
- **interaction_edges** (*list (optional, default None)*) – A list of edges used as interactions. If given, only these couplers are displayed.
- **show_labels** (*boolean (optional, default False)*) – If True, each chain in *emb* is labelled with its key.
- **chain_color** (*dict (optional, default None)*) – Colors as a dict of form {node: rgba_color, ...} associated with each key in *emb*, where colors are length-4 tuples of floats between 0 and 1 inclusive. If None, each chain is assigned a different color.
- **unused_color** (*tuple (optional, default (0.9,0.9,0.9,1.0))*) – Color for nodes of *G* that are not part of chains, and edges that are neither chain edges nor interactions. If None, these nodes and edges are not shown.
- **crosses** (*boolean (optional, default False)*) – If True, $K_{4,4}$ subgraphs are shown in a cross rather than L configuration. Ignored if *G* is defined with `nice_coordinates=True`.
- **overlapped_embedding** (*boolean (optional, default False)*) – If True, chains in *emb* may overlap (contain the same vertices in *G*), and these overlaps are displayed as concentric circles.
- **kwargs** (*optional keywords*) – See `networkx.draw_networkx()` for a description of optional keywords, with the exception of the `pos` parameter, which is not used by this function. If `linear_biases` or `quadratic_biases` are provided, any provided `node_color` or `edge_color` arguments are ignored.

dwave_networkx.drawing.pegasus_layout.pegasus_layout

pegasus_layout (*G*, *scale=1.0*, *center=None*, *dim=2*, *crosses=False*)

Positions the nodes of graph *G* in a Pegasus topology.

NumPy is required for this function.

Parameters

- **G** (*NetworkX graph*) – A Pegasus graph or a subgraph of a Pegasus graph, as produced by the `dwave_networkx.pegasus_graph()` function.
- **scale** (*float (default 1.)*) – Scale factor. A setting of `scale = 1` fits all positions within [0, 1] on the x-axis and [-1, 0] on the y-axis.

- **center** (*None or array (default None)*) – Coordinates of the top left corner.
- **dim** (*int (default 2)*) – Number of dimensions. When $\text{dim} > 2$, all extra dimensions are set to 0.
- **crosses** (*boolean (optional, default False)*) – If True, $K_{4,4}$ subgraphs are shown in a cross rather than L configuration. Ignored if G is defined with `nice_coordinates=True`.

Returns `pos` – Positions as a dictionary keyed by node.

Return type `dict`

Examples

This example gives the positions of a Pegasus lattice of size 2.

```
>>> G = dnx.pegasus_graph(2)
>>> pos = dnx.pegasus_layout(G)
```

`dwave_networkx.drawing.pegasus_layout.pegasus_node_placer_2d`

pegasus_node_placer_2d (*G, scale=1.0, center=None, dim=2, crosses=False*)

Generates a function to convert Pegasus indices to plottable coordinates.

Parameters

- **G** (*NetworkX graph*) – A Pegasus graph or a subgraph of a Pegasus graph, as produced by the `dwave_networkx.pegasus_graph()` function.
- **scale** (*float (default 1.)*) – Scale factor. A setting of `scale = 1` fits all positions within $[0, 1]$ on the x-axis and $[-1, 0]$ on the y-axis.
- **center** (*None or array (default None)*) – Coordinates of the top left corner.
- **dim** (*int (default 2)*) – Number of dimensions. When $\text{dim} > 2$, all extra dimensions are set to 0.
- **crosses** (*boolean (optional, default False)*) – If True, $K_{4,4}$ subgraphs are shown in a cross rather than L configuration.

Returns `xy_coords` – A function that maps a Pegasus index (u, w, k, z) in a Pegasus lattice to plottable x,y coordinates.

Return type `function`

Example

This example uses the `draw_pegasus()` function to show the positions of nodes of a simple 5-node graph on a small Pegasus lattice.

```
>>> import dwave_networkx as dnx
>>> import matplotlib.pyplot as plt
>>> G = dnx.pegasus_graph(2)
>>> H = dnx.pegasus_graph(2, node_list=[4, 40, 41, 42, 43],
>>>     edge_list=[(4, 40), (4, 41), (4, 42), (4, 43)])
>>> # Show graph H on a small Pegasus lattice
```

(continues on next page)

(continued from previous page)

```
>>> plt.ion()
>>> # Show graph H on a small Pegasus lattice
>>> plt.ion()
>>> dnx.draw_pegasus(G, with_labels=True, crosses=True, node_color="Yellow")
>>> dnx.draw_pegasus(H, crosses=True, node_color='b', style='dashed',
                    edge_color='b', width=3)
```

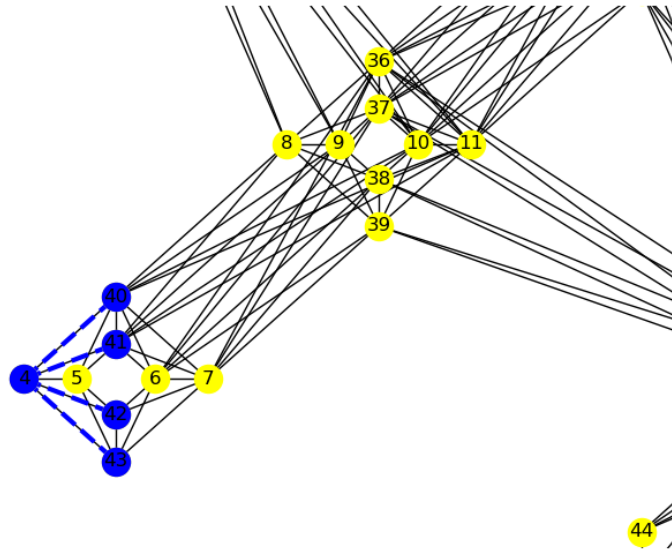


Fig. 9: Graph H (blue) overlaid on a small Pegasus lattice(yellow nodes and black edges).

Graph Generators

Generators for graphs, such the graphs (topologies) of D-Wave System QPUs.

D-Wave Systems

<code>chimera_graph(m[, n, t, create_using, ...])</code>	Creates a Chimera lattice of size (m, n, t).
<code>pegasus_graph(m[, create_using, node_list, ...])</code>	Creates a Pegasus graph with size parameter <i>m</i> .

dwave_networkx.chimera_graph

chimera_graph (*m*, *n=None*, *t=None*, *create_using=None*, *node_list=None*, *edge_list=None*, *data=True*, *coordinates=False*)
 Creates a Chimera lattice of size (m, n, t).

Parameters

- **m** (*int*) – Number of rows in the Chimera lattice.
- **n** (*int optional, default m*) – Number of columns in the Chimera lattice.
- **t** (*int optional, default 4*) – Size of the shore within each Chimera tile.
- **create_using** (*Graph optional, default None*) – If provided, this graph

is cleared of nodes and edges and filled with the new graph. Usually used to set the type of the graph.

- **node_list** (*iterable (optional, default None)*) – Iterable of nodes in the graph. If None, calculated from (m, n, t). Note that this list is used to remove nodes, so any nodes specified not in $range(m * n * 2 * t)$ are not added.
- **edge_list** (*iterable (optional, default None)*) – Iterable of edges in the graph. If None, edges are generated as described below. The nodes in each edge must be integer-labeled in $range(m * n * t * 2)$.
- **data** (*bool (optional, default True)*) – If True, each node has a *chimera_index* attribute. The attribute is a 4-tuple Chimera index as defined below.
- **coordinates** (*bool (optional, default False)*) – If True, node labels are 4-tuples, equivalent to the *chimera_index* attribute as below. In this case, the *data* parameter controls the existence of a *linear_index* attribute, which is an int.

Returns **G** – An (m, n, t) Chimera lattice. Nodes are labeled by integers.

Return type NetworkX Graph

A Chimera lattice is an m-by-n grid of Chimera tiles. Each Chimera tile is itself a bipartite graph with shores of size t. The connection in a Chimera lattice can be expressed using a node-indexing notation (i,j,u,k) for each node.

- (i,j) indexes the (row, column) of the Chimera tile. i must be between 0 and m-1, inclusive, and j must be between 0 and n-1, inclusive.
- u=0 indicates the left-hand nodes in the tile, and u=1 indicates the right-hand nodes.
- k=0,1,...,t-1 indexes nodes within either the left- or right-hand shores of a tile.

In this notation, two nodes (i, j, u, k) and (i', j', u', k') are neighbors if and only if:

$$(i = i' \text{ AND } j = j' \text{ AND } u \neq u') \text{ OR } (i = i' \pm 1 \text{ AND } j = j' \text{ AND } u = 0 \text{ AND } u' = 0 \text{ AND } k = k') \text{ OR } (i = i' \text{ AND } j = j' \pm 1 \text{ AND } u = 1 \text{ AND } u' = 1 \text{ AND } k = k')$$

The first of the three terms of the disjunction gives the bipartite connections within the tile. The second and third terms give the vertical and horizontal connections between blocks respectively.

Node (i, j, u, k) is labeled by:

$$\text{label} = i * n * 2 * t + j * 2 * t + u * t + k$$

Examples

```
>>> G = dnx.chimera_graph(1, 1, 2) # a single Chimera tile
>>> len(G)
4
>>> list(G.nodes()) # doctest: +SKIP
[0, 1, 2, 3]
>>> list(G.nodes(data=True)) # doctest: +SKIP
[(0, {'chimera_index': (0, 0, 0, 0)}),
 (1, {'chimera_index': (0, 0, 0, 1)}),
 (2, {'chimera_index': (0, 0, 1, 0)}),
 (3, {'chimera_index': (0, 0, 1, 1)})]
>>> list(G.edges()) # doctest: +SKIP
[(0, 2), (0, 3), (1, 2), (1, 3)]
```

dwave_networkx.pegasus_graph

pegasus_graph (*m*, *create_using=None*, *node_list=None*, *edge_list=None*, *data=True*, *offset_lists=None*, *offsets_index=None*, *coordinates=False*, *fabric_only=True*, *nice_coordinates=False*)

Creates a Pegasus graph with size parameter *m*.

Parameters

- **m** (*int*) – Size parameter for the Pegasus lattice.
- **create_using** (*Graph*, *optional (default None)*) – If provided, this graph is cleared of nodes and edges and filled with the new graph. Usually used to set the type of the graph.
- **node_list** (*iterable*, *optional (default None)*) – Iterable of nodes in the graph. If None, calculated from *m*. Note that this list is used to remove nodes, so any nodes specified not in $\text{range}(24 * m * (m-1))$ are not added.
- **edge_list** (*iterable*, *optional (default None)*) – Iterable of edges in the graph. If None, edges are generated as described below. The nodes in each edge must be integer-labeled in $\text{range}(24 * m * (m-1))$.
- **data** (*bool*, *optional (default True)*) – If True, each node has a `Pegasus_index` attribute. The attribute is a 4-tuple Pegasus index as defined below. If the `coordinates` parameter is True, a `linear_index`, which is an integer, is used.
- **coordinates** (*bool*, *optional (default False)*) – If True, node labels are 4-tuple Pegasus indices. Ignored if the `nice_coordinates` parameter is True.
- **offset_lists** (*pair of lists*, *optional (default None)*) – Directly controls the offsets. Each list in the pair must have length 12 and contain even ints. If `offset_lists` is not None, the `offsets_index` parameter must be None.
- **offsets_index** (*int*, *optional (default None)*) – A number between 0 and 7, inclusive, that selects a preconfigured set of topological parameters. If both the `offsets_index` and `offset_lists` parameters are None, the `offsets_index` parameters is set to zero. At least one of these two parameters must be None.
- **fabric_only** (*bool*, *optional (default True)*) – The Pegasus graph, by definition, has some disconnected components. If True, the generator only constructs nodes from the largest component. If False, the full disconnected graph is constructed. Ignored if the `edge_lists` parameter is not None or `nice_coordinates` is True
- **nice_coordinates** (*bool*, *optional (default False)*) – If the `offsets_index` parameter is 0, the graph uses a “nicer” coordinate system, more compatible with Chimera addressing. These coordinates are 5-tuples taking the form (t, y, x, u, k) where $0 \leq x < M - 1$, $0 \leq y < M - 1$, $0 \leq u < 2$, $0 \leq k < 4$, and $0 \leq t < 3$. For any given $0 \leq t_0 < 3$, the subgraph of nodes with $t = t_0$ has the structure of `chimera(M-1, M-1, 4)` with the addition of odd couplers. Supersedes both the `fabric_only` and `coordinates` parameters.

Returns **G** – A Pegasus lattice for size parameter *m*.

Return type NetworkX Graph

The maximum degree of this graph is 15. The number of nodes depends on multiple parameters; for example,

- `pegasus_graph(1)`: zero nodes
- `pegasus_graph(m, fabric_only=False)`: $24m(m - 1)$ nodes
- `pegasus_graph(m, fabric_only=True)`: $24m(m - 1) - 8(m - 1)$ nodes

- `pegasus_graph(m, nice_coordinates=True)`: $24(m - 1)^2$ nodes

Counting formulas for edges have a complicated dependency on parameter settings. Some example upper bounds are:

- `pegasus_graph(1, fabric_only=False)`: zero edges
- `pegasus_graph(m, fabric_only=False)`: $12 * (15 * (m - 1)^2 + m - 3)$ edges if $m > 1$

Note that the formulas above are valid for default offset parameters.

A Pegasus lattice is a graph minor of a lattice similar to Chimera, where unit tiles are completely connected. In its most general definition, prelatice $Q(N0, N1)$ contains nodes of the form

- vertical nodes: $(i, j, 0, k)$ with $0 \leq k < 2$
- horizontal nodes: $(i, j, 1, k)$ with $0 \leq k < 2$

for $0 \leq i \leq N0$ and $0 \leq j < N1$, and edges of the form

- external: $(i, j, u, k) \sim (i + u, j + 1 - u, u, k)$
- internal: $(i, j, 0, k) \sim (i, j, 1, k)$
- odd: $(i, j, u, 0) \sim (i, j, u, 1)$

Given two lists of offsets, $S0$ and $S1$, of length $L0$ and $L1$, where both lengths and values must be divisible by 2, the minor—a Pegasus lattice—is constructed by contracting the complete intervals of external edges:

```
I(0, w, k, z) = [(L1*w + k, L0*z + S0[k] + r, 0, k % 2) for 0 <= r < L0]
I(1, w, k, z) = [(L1*z + S1[k] + r, L0*w + k, 1, k % 2) for 0 <= r < L1]
```

and deleting the prelatice nodes of any interval not fully contained in $Q(N0, N1)$.

This generator, `pegasus_graph()`, is specialized for the minor constructed by prelatice and offset parameters $L0 = L1 = 12$ and $N0 = N1 = 12m$.

The *Pegasus index* of a node in a Pegasus lattice, (u, w, k, z) , can be interpreted as:

- u : qubit orientation (0 = vertical, 1 = horizontal)
- w : orthogonal major offset
- k : orthogonal minor offset
- z : parallel offset

Edges in the minor have the form

- external: $(u, w, k, z) \sim (u, w, k, z + 1)$
- internal: $(0, w0, k0, z0) \sim (1, w1, k1, z1)$
- odd: $(u, w, 2k, z) \sim (u, w, 2k + 1, z)$

where internal edges only exist when

1. $w1 = z0 + (1 \text{ if } k1 < S0[k0] \text{ else } 0)$
2. $z1 = w0 - (1 \text{ if } k0 < S1[k1] \text{ else } 0)$

Linear indices are computed from Pegasus indices by the formula:

```
q = ((u * m + w) * 12 + k) * (m - 1) + z
```


Examples

```
>>> G = dnx.pegasus_graph(2, nice_coordinates=True)
>>> G.nodes(data=True)[(0, 0, 0, 0, 0)] # doctest: +SKIP
{'linear_index': 4, 'pegasus_index': (0, 0, 4, 0)}
```

Example

This example uses the `chimera_graph()` function to create a Chimera lattice of size (1, 1, 4), which is a single unit cell in Chimera topology, and the `find_chimera_indices()` function to determine the Chimera indices.

```
>>> import networkx as nx
>>> import dwave_networkx as dnx
>>> G = dnx.chimera_graph(1, 1, 4)
>>> chimera_indices = dnx.find_chimera_indices(G)
>>> print chimera_indices
{0: (0, 0, 0, 0),
 1: (0, 0, 0, 1),
 2: (0, 0, 0, 2),
 3: (0, 0, 0, 3),
 4: (0, 0, 1, 0),
 5: (0, 0, 1, 1),
 6: (0, 0, 1, 2),
 7: (0, 0, 1, 3)}
```

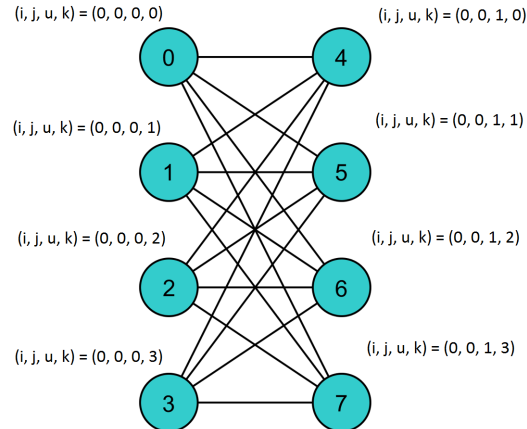


Fig. 10: Indices of a Chimera unit cell found by creating a lattice of size (1, 1, 4).

Other Graphs

`markov_network(potentials)`

Creates a Markov Network from potentials.

dwave_networkx.markov_network

markov_network (potentials)

Creates a Markov Network from potentials.

A Markov Network is also known as a [Markov Random Field](#)

Parameters `potentials` (*dict[tuple, dict]*) – A dict where the keys are either nodes or edges and the values are a dictionary of potentials. The potential dict should map each possible assignment of the nodes/edges to their energy.

Returns `MN` – A markov network as a graph where each node/edge stores its potential dict as above.

Return type `networkx.Graph`

Examples

```
>>> potentials = {('a', 'b'): {(0, 0): -1,
...                           (0, 1): .5,
...                           (1, 0): .5,
...                           (1, 1): 2}}
>>> MN = dnx.markov_network(potentials)
>>> MN['a']['b']['potential'][(0, 0)]
-1
```

Utilities

Decorators

Decorators allow for input checking and default parameter setting for algorithms.

`binary_quadratic_model_sampler`(*which_args*) Decorator to validate sampler arguments.

dwave_networkx.utils.decorators.binary_quadratic_model_sampler

binary_quadratic_model_sampler (which_args)

Decorator to validate sampler arguments.

Parameters `which_args` (*int or sequence of ints*) – Location of the sampler arguments of the input function in the form `function_name(args, *kw)`. If more than one sampler is allowed, can be a list of locations.

Returns `_binary_quadratic_model_sampler` – Caller function that validates the sampler format. A sampler is expected to have `sample_qubo` and `sample_ising` methods. Alternatively, if no sampler is provided (or sampler is None), the sampler set by the `set_default_sampler` function is provided to the function.

Return type `function`

Examples

Decorate functions like this:

```
@binary_quadratic_model_sampler(1)
def maximal_matching(G, sampler, **sampler_args):
    pass
```

This example validates two placeholder samplers, which return a correct response only in the case of finding an independent set on a complete graph (where one node is always an independent set), the first valid, the second missing a method.

```
>>> import networkx as nx
>>> import dwave_networkx as dnx
>>> from dwave_networkx.utils import decorators
>>> # Create two placeholder samplers
>>> class WellDefinedSampler:
...     # an example sampler, only works for independent set on complete
...     # graphs
...     def __init__(self, name):
...         self.name = name
...     def sample_ising(self, h, J):
...         sample = {v: -1 for v in h}
...         sample[0] = 1 # set one node to true
...         return [sample]
...     def sample_qubo(self, Q):
...         sample = {v: 0 for v in set().union(*Q)}
...         sample[0] = 1 # set one node to true
...         return [sample]
...     def __str__(self):
...         return self.name
...
>>> class IllDefinedSampler:
...     # an example sampler missing a `sample_qubo` method
...     def __init__(self, name):
...         self.name = name
...     def sample_ising(self, h, J):
...         sample = {v: -1 for v in h}
...         sample[0] = 1 # set one node to true
...         return [sample]
...     def __str__(self):
...         return self.name
...
>>> sampler1 = WellDefinedSampler('sampler1')
>>> sampler2 = IllDefinedSampler('sampler2')
>>> # Define a placeholder independent-set function with the decorator
>>> @dnx.utils.binary_quadratic_model_sampler(1)
... def independent_set(G, sampler, **sampler_args):
...     Q = {(node, node): -1 for node in G}
...     Q.update({edge: 2 for edge in G.edges})
...     response = sampler.sample_qubo(Q, **sampler_args)
...     sample = next(iter(response))
...     return [node for node in sample if sample[node] > 0]
...
>>> # Validate the samplers
>>> G = nx.complete_graph(5)
>>> independent_set(G, sampler1)
[0]
>>> independent_set(G, sampler2) # doctest: +SKIP
-----
TypeError                                 Traceback (most recent call last)
```

(continues on next page)

(continued from previous page)

```

<ipython-input-35-670b71b268c7> in <module>()
----> 1 independent_set(G, IllDefinedSampler)
<decorator-gen-628> in independent_set(G, sampler, **sampler_args)
/usr/local/lib/python2.7/dist-packages/dwave_networkx/utils/decorators.pyc in _
↳binary_quadratic_model_sampler(f, *args, **kw)
    61
    62         if not hasattr(sampler, "sample_qubo") or not_
↳callable(sampler.sample_qubo):
--> 63             raise TypeError("expected sampler to have a 'sample_qubo'_
↳method")
    64         if not hasattr(sampler, "sample_ising") or not_
↳callable(sampler.sample_ising):
    65             raise TypeError("expected sampler to have a 'sample_ising
↳' method")
TypeError: expected sampler to have a 'sample_qubo' method

```

Coordinates Conversion

class chimera_coordinates (*m, n=None, t=None*)

Provides coordinate converters for the chimera indexing scheme.

Parameters

- **m** (*int*) – The number of rows in the Chimera lattice.
- **n** (*int, optional (default m)*) – The number of columns in the Chimera lattice.
- **t** (*int, optional (default 4)*) – The size of the shore within each Chimera tile.

Examples

Convert between Chimera coordinates and linear indices directly

```

>>> coords = dnx.chimera_coordinates(16, 16, 4)
>>> coords.chimera_to_linear((0, 2, 0, 1))
17
>>> coords.linear_to_chimera(17)
(0, 2, 0, 1)

```

Construct a new graph with the coordinate labels

```

>>> C16 = dnx.chimera_graph(16)
>>> coords = dnx.chimera_coordinates(16)
>>> G = nx.Graph()
>>> G.add_nodes_from(coords.iter_linear_to_chimera(C16.nodes))
>>> G.add_edges_from(coords.iter_linear_to_chimera_pairs(C16.edges))

```

See also:

[*chimera_graph\(\)*](#) Describes the various conventions.

class pegasus_coordinates (*m*)

Provides coordinate converters for the Pegasus indexing schemes.

Parameters **m** (*int*) – Size parameter for the Pegasus lattice.

See also:

`pegasus_graph()` Describes the various coordinate conventions.

Graph Indexing

See *Coordinates Conversion* on instantiating the needed lattice size and setting correct domain and range for coordinates in a QPU working graph.

For the iterator versions of these functions, see the code.

Chimera

<code>chimera_coordinates.chimera_to_linear(q)</code>	Convert a 4-term Chimera coordinate to a linear index.
<code>chimera_coordinates.linear_to_chimera(r)</code>	Convert a linear index to a 4-term Chimera coordinate.
<code>find_chimera_indices(G)</code>	Attempts to determine the Chimera indices of the nodes in graph G.

`dwave_networkx.chimera_coordinates.chimera_to_linear`

`chimera_coordinates.chimera_to_linear(q)`
Convert a 4-term Chimera coordinate to a linear index.

Parameters `q` (*4-tuple*) – Chimera coordinate.

Examples

```
>>> dnx.chimera_coordinates(16).chimera_to_linear((2, 2, 0, 0))
272
```

`dwave_networkx.chimera_coordinates.linear_to_chimera`

`chimera_coordinates.linear_to_chimera(r)`
Convert a linear index to a 4-term Chimera coordinate.

Parameters `r` (*int*) – Linear index.

Examples

```
>>> dnx.chimera_coordinates(16).linear_to_chimera(272)
(2, 2, 0, 0)
```

`dwave_networkx.find_chimera_indices`

`find_chimera_indices(G)`
Attempts to determine the Chimera indices of the nodes in graph G.

See the `chimera_graph()` function for a definition of a Chimera graph and Chimera indices.

Parameters `G` (*NetworkX graph*) – Should be a single-tile Chimera graph.

Returns `chimera_indices` – A dict of the form {node: (i, j, u, k), ... } where (i, j, u, k) is a 4-tuple of integer Chimera indices.

Return type `dict`

Examples

```
>>> G = dnx.chimera_graph(1, 1, 4)
>>> chimera_indices = dnx.find_chimera_indices(G)
```

```
>>> G = nx.Graph()
>>> G.add_edges_from([(0, 2), (1, 2), (1, 3), (0, 3)])
>>> chimera_indices = dnx.find_chimera_indices(G)
>>> nx.set_node_attributes(G, chimera_indices, 'chimera_index')
```

Pegasus

<code>pegasus_coordinates.linear_to_nice(r)</code>	Convert a linear index into a 5-term nice coordinate.
<code>pegasus_coordinates.linear_to_pegasus(r)</code>	Convert a linear index into a 4-term Pegasus coordinate.
<code>pegasus_coordinates.nice_to_linear(n)</code>	Convert a 5-term nice coordinate into a linear index.
<code>pegasus_coordinates.nice_to_pegasus(n)</code>	Convert a 5-term nice coordinate into a 4-term Pegasus coordinate.
<code>pegasus_coordinates.pegasus_to_linear(q)</code>	Convert a 4-term Pegasus coordinate into a linear index.
<code>pegasus_coordinates.pegasus_to_nice(p)</code>	Convert a 4-term Pegasus coordinate to a 5-term nice coordinate.

`dwave_networkx.pegasus_coordinates.linear_to_nice`

`pegasus_coordinates.linear_to_nice(r)`
Convert a linear index into a 5-term nice coordinate.

Parameters `r` (*int*) – Linear index.

Examples

```
>>> dnx.pegasus_coordinates(2).linear_to_nice(4)
(0, 0, 0, 0, 0)
```

`dwave_networkx.pegasus_coordinates.linear_to_pegasus`

`pegasus_coordinates.linear_to_pegasus(r)`
Convert a linear index into a 4-term Pegasus coordinate.

Parameters `r` (*int*) – Linear index.

Examples

```
>>> dnx.pegasus_coordinates(2).linear_to_pegasus(4)
(0, 0, 4, 0)
```

`dwave_networkx.pegasus_coordinates.nice_to_linear`

`pegasus_coordinates.nice_to_linear`(*n*)

Convert a 5-term nice coordinate into a linear index.

Parameters *n* (*5-tuple*) – Nice coordinate.

Examples

```
>>> dnx.pegasus_coordinates(2).nice_to_linear((0, 0, 0, 0, 0))
4
```

`dwave_networkx.pegasus_coordinates.nice_to_pegasus`

static `pegasus_coordinates.nice_to_pegasus`(*n*)

Convert a 5-term nice coordinate into a 4-term Pegasus coordinate.

Parameters *n* (*5-tuple*) – Nice coordinate.

Examples

```
>>> dnx.pegasus_coordinates.nice_to_pegasus((0, 0, 0, 0, 0))
(0, 0, 4, 0)
```

Note that this method does not depend on the size of the Pegasus lattice.

`dwave_networkx.pegasus_coordinates.pegasus_to_linear`

`pegasus_coordinates.pegasus_to_linear`(*q*)

Convert a 4-term Pegasus coordinate into a linear index.

Parameters *q* (*4-tuple*) – Pegasus indices.

Examples

```
>>> dnx.pegasus_coordinates(2).pegasus_to_linear((0, 0, 4, 0))
4
```

dwave_networkx.pegasus_coordinates.pegasus_to_nice

static `pegasus_coordinates.pegasus_to_nice(p)`
Convert a 4-term Pegasus coordinate to a 5-term nice coordinate.

Parameters `p` (*4-tuple*) – Pegasus coordinate.

Examples

```
>>> dnx.pegasus_coordinates.pegasus_to_nice((0, 0, 4, 0))
(0, 0, 0, 0, 0)
```

Note that this method does not depend on the size of the Pegasus lattice.

Exceptions

Base exceptions and errors for D-Wave NetworkX.

All exceptions are derived from `NetworkXException`.

<i>DWaveNetworkXException</i>	Base class for exceptions in <code>DWaveNetworkX</code> .
<i>DWaveNetworkXMissingSampler</i>	Exception raised by an algorithm requiring a discrete model sampler when none is provided.

dwave_networkx.exceptions.DWaveNetworkXException

exception `DWaveNetworkXException`
Base class for exceptions in `DWaveNetworkX`.

dwave_networkx.exceptions.DWaveNetworkXMissingSampler

exception `DWaveNetworkXMissingSampler`
Exception raised by an algorithm requiring a discrete model sampler when none is provided.

Default sampler

Sets a binary quadratic model sampler used by default for functions that require a sample when none is specified.

A sampler is a process that samples from low-energy states in models defined by an Ising equation or a Quadratic Unconstrained Binary Optimization Problem (QUBO).

Sampler API

- Required Methods: ‘sample_qubo’ and ‘sample_ising’
- Return value: iterable of samples, in order of increasing energy

See `dimod` for details.

Example

This example creates and uses a placeholder for binary quadratic model samplers that returns a correct response only in the case of finding an independent set on a complete graph (where one node is always an independent set). The placeholder sampler can be used to test the simple examples of the functions for configuring a default sampler.

```
>>> # Create a placeholder sampler
>>> class ExampleSampler:
...     # an example sampler, only works for independent set on complete
...     # graphs
...     def __init__(self, name):
...         self.name = name
...     def sample_ising(self, h, J):
...         sample = {v: -1 for v in h}
...         sample[0] = 1 # set one node to true
...         return [sample]
...     def sample_qubo(self, Q):
...         sample = {v: 0 for v in set().union(*Q)}
...         sample[0] = 1 # set one node to true
...         return [sample]
...     def __str__(self):
...         return self.name
...
>>> # Identify the new sampler as the default sampler
>>> sampler0 = ExampleSampler('sampler0')
>>> dnx.set_default_sampler(sampler0)
>>> # Find an independent set using the default sampler
>>> G = nx.complete_graph(5)
>>> dnx.maximum_independent_set(G)
[0]
```

Functions

<code>set_default_sampler(sampler)</code>	Sets a default binary quadratic model sampler.
<code>unset_default_sampler()</code>	Resets the default sampler back to None.
<code>get_default_sampler()</code>	Queries the current default sampler.

dwave_networkx.default_sampler.set_default_sampler

`set_default_sampler(sampler)`

Sets a default binary quadratic model sampler.

Parameters `sampler` – A binary quadratic model sampler. A sampler is a process that samples from low-energy states in models defined by an Ising equation or a Quadratic Unconstrained Binary Optimization Problem (QUBO). A sampler is expected to have a ‘sample_qubo’ and ‘sample_ising’ method. A sampler is expected to return an iterable of samples, in order of increasing energy.

Examples

This example sets `sampler0` as the default sampler and finds an independent set for graph `G`, first using the default sampler and then overriding it by specifying a different sampler.

```
>>> dnx.set_default_sampler(sampler0) # doctest: +SKIP
>>> indep_set = dnx.maximum_independent_set_dm(G) # doctest: +SKIP
>>> indep_set = dnx.maximum_independent_set_dm(G, sampler1) # doctest: +SKIP
```

`dwave_networkx.default_sampler.unset_default_sampler`

`unset_default_sampler()`

Resets the default sampler back to None.

Examples

This example sets `sampler0` as the default sampler, verifies the setting, then resets the default, and verifies the resetting.

```
>>> dnx.set_default_sampler(sampler0) # doctest: +SKIP
>>> print(dnx.get_default_sampler()) # doctest: +SKIP
'sampler0'
>>> dnx.unset_default_sampler() # doctest: +SKIP
>>> print(dnx.get_default_sampler()) # doctest: +SKIP
None
```

`dwave_networkx.default_sampler.get_default_sampler`

`get_default_sampler()`

Queries the current default sampler.

Examples

This example queries the default sampler before and after specifying a default sampler.

```
>>> print(dnx.get_default_sampler()) # doctest: +SKIP
None
>>> dnx.set_default_sampler(sampler) # doctest: +SKIP
>>> print(dnx.get_default_sampler()) # doctest: +SKIP
'sampler'
```

2.1.3 Bibliography

2.1.4 Installation

Installation from PyPi:

```
pip install dwave_networkx
```

Installation from source:

```
pip install -r requirements.txt
python setup.py install
```

2.1.5 License

Apache License

Version 2.0, January 2004

<http://www.apache.org/licenses/>

TERMS AND CONDITIONS FOR USE, REPRODUCTION, AND DISTRIBUTION

1. Definitions.

“License” shall mean the terms and conditions for use, reproduction, and distribution as defined by Sections 1 through 9 of this document.

“Licensor” shall mean the copyright owner or entity authorized by the copyright owner that is granting the License.

“Legal Entity” shall mean the union of the acting entity and all other entities that control, are controlled by, or are under common control with that entity. For the purposes of this definition, “control” means (i) the power, direct or indirect, to cause the direction or management of such entity, whether by contract or otherwise, or (ii) ownership of fifty percent (50%) or more of the outstanding shares, or (iii) beneficial ownership of such entity.

“You” (or “Your”) shall mean an individual or Legal Entity exercising permissions granted by this License.

“Source” form shall mean the preferred form for making modifications, including but not limited to software source code, documentation source, and configuration files.

“Object” form shall mean any form resulting from mechanical transformation or translation of a Source form, including but not limited to compiled object code, generated documentation, and conversions to other media types.

“Work” shall mean the work of authorship, whether in Source or Object form, made available under the License, as indicated by a copyright notice that is included in or attached to the work (an example is provided in the Appendix below).

“Derivative Works” shall mean any work, whether in Source or Object form, that is based on (or derived from) the Work and for which the editorial revisions, annotations, elaborations, or other modifications represent, as a whole, an original work of authorship. For the purposes of this License, Derivative Works shall not include works that remain separable from, or merely link (or bind by name) to the interfaces of, the Work and Derivative Works thereof.

“Contribution” shall mean any work of authorship, including the original version of the Work and any modifications or additions to that Work or Derivative Works thereof, that is intentionally submitted to Licensor for inclusion in the Work by the copyright owner or by an individual or Legal Entity authorized to submit on behalf of the copyright owner. For the purposes of this definition, “submitted” means any form of electronic, verbal, or written communication sent to the Licensor or its representatives, including but not limited to communication on electronic mailing lists, source code control systems, and issue tracking systems that are managed by, or on behalf of, the Licensor for the purpose of discussing and improving the Work, but excluding communication that is conspicuously marked or otherwise designated in writing by the copyright owner as “Not a Contribution.”

“Contributor” shall mean Licensor and any individual or Legal Entity on behalf of whom a Contribution has been received by Licensor and subsequently incorporated within the Work.

2. Grant of Copyright License. Subject to the terms and conditions of this License, each Contributor hereby grants to You a perpetual, worldwide, non-exclusive, no-charge, royalty-free, irrevocable copyright license to reproduce, prepare Derivative Works of, publicly display, publicly perform, sublicense, and distribute the Work and such Derivative Works in Source or Object form.

3. **Grant of Patent License.** Subject to the terms and conditions of this License, each Contributor hereby grants to You a perpetual, worldwide, non-exclusive, no-charge, royalty-free, irrevocable (except as stated in this section) patent license to make, have made, use, offer to sell, sell, import, and otherwise transfer the Work, where such license applies only to those patent claims licensable by such Contributor that are necessarily infringed by their Contribution(s) alone or by combination of their Contribution(s) with the Work to which such Contribution(s) was submitted. If You institute patent litigation against any entity (including a cross-claim or counterclaim in a lawsuit) alleging that the Work or a Contribution incorporated within the Work constitutes direct or contributory patent infringement, then any patent licenses granted to You under this License for that Work shall terminate as of the date such litigation is filed.
4. **Redistribution.** You may reproduce and distribute copies of the Work or Derivative Works thereof in any medium, with or without modifications, and in Source or Object form, provided that You meet the following conditions:
 - (a) You must give any other recipients of the Work or Derivative Works a copy of this License; and
 - (b) You must cause any modified files to carry prominent notices stating that You changed the files; and
 - (c) You must retain, in the Source form of any Derivative Works that You distribute, all copyright, patent, trademark, and attribution notices from the Source form of the Work, excluding those notices that do not pertain to any part of the Derivative Works; and
 - (d) If the Work includes a “NOTICE” text file as part of its distribution, then any Derivative Works that You distribute must include a readable copy of the attribution notices contained within such NOTICE file, excluding those notices that do not pertain to any part of the Derivative Works, in at least one of the following places: within a NOTICE text file distributed as part of the Derivative Works; within the Source form or documentation, if provided along with the Derivative Works; or, within a display generated by the Derivative Works, if and wherever such third-party notices normally appear. The contents of the NOTICE file are for informational purposes only and do not modify the License. You may add Your own attribution notices within Derivative Works that You distribute, alongside or as an addendum to the NOTICE text from the Work, provided that such additional attribution notices cannot be construed as modifying the License.

You may add Your own copyright statement to Your modifications and may provide additional or different license terms and conditions for use, reproduction, or distribution of Your modifications, or for any such Derivative Works as a whole, provided Your use, reproduction, and distribution of the Work otherwise complies with the conditions stated in this License.

5. **Submission of Contributions.** Unless You explicitly state otherwise, any Contribution intentionally submitted for inclusion in the Work by You to the Licensor shall be under the terms and conditions of this License, without any additional terms or conditions. Notwithstanding the above, nothing herein shall supersede or modify the terms of any separate license agreement you may have executed with Licensor regarding such Contributions.
6. **Trademarks.** This License does not grant permission to use the trade names, trademarks, service marks, or product names of the Licensor, except as required for reasonable and customary use in describing the origin of the Work and reproducing the content of the NOTICE file.
7. **Disclaimer of Warranty.** Unless required by applicable law or agreed to in writing, Licensor provides the Work (and each Contributor provides its Contributions) on an “AS IS” BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied, including, without limitation, any warranties or conditions of TITLE, NON-INFRINGEMENT, MERCHANTABILITY, or FITNESS FOR A PARTICULAR PURPOSE. You are solely responsible for determining the appropriateness of using or redistributing the Work and assume any risks associated with Your exercise of permissions under this License.

8. **Limitation of Liability.** In no event and under no legal theory, whether in tort (including negligence), contract, or otherwise, unless required by applicable law (such as deliberate and grossly negligent acts) or agreed to in writing, shall any Contributor be liable to You for damages, including any direct, indirect, special, incidental, or consequential damages of any character arising as a result of this License or out of the use or inability to use the Work (including but not limited to damages for loss of goodwill, work stoppage, computer failure or malfunction, or any and all other commercial damages or losses), even if such Contributor has been advised of the possibility of such damages.
9. **Accepting Warranty or Additional Liability.** While redistributing the Work or Derivative Works thereof, You may choose to offer, and charge a fee for, acceptance of support, warranty, indemnity, or other liability obligations and/or rights consistent with this License. However, in accepting such obligations, You may act only on Your own behalf and on Your sole responsibility, not on behalf of any other Contributor, and only if You agree to indemnify, defend, and hold each Contributor harmless for any liability incurred by, or claims asserted against, such Contributor by reason of your accepting any such warranty or additional liability.

2.2 Indices and tables

- [genindex](#)
- [modindex](#)
- [search](#)
- [Glossary](#)

Bibliography

- [AL] Lucas, A. (2014). Ising formulations of many NP problems. *Frontiers in Physics*, Volume 2, Article 5.
- [DWMP] Dahl, E., “Programming the D-Wave: Map Coloring Problem”, <https://www.dwavesys.com/sites/default/files/Map%20Coloring%20WP2.pdf>
- [DWMP] Dahl, E., “Programming the D-Wave: Map Coloring Problem”, <https://www.dwavesys.com/sites/default/files/Map%20Coloring%20WP2.pdf>
- [AL] Lucas, A. (2014). Ising formulations of many NP problems. *Frontiers in Physics*, Volume 2, Article 5.
- [bbrr] Boothby, K., P. Bunky, J. Raymond, A. Roy. Next-Generation Topology of D-Wave Quantum Processors. Technical Report, February 2019. <https://www.dwavesys.com/resources/publications?type=white#publication-987>
- [GD] Gogate & Dechter, “A Complete Anytime Algorithm for Treewidth”, <https://arxiv.org/abs/1207.4109>
- [GD] Gogate & Dechter. “A Complete Anytime Algorithm for Treewidth.” <https://arxiv.org/abs/1207.4109>
- [AL] Lucas, A. (2014). Ising formulations of many NP problems. *Frontiers in Physics*, Volume 2, Article 5.
- [AL] Lucas, A. (2014). Ising formulations of many NP problems. *Frontiers in Physics*, Volume 2, Article 5.
- [AL] Lucas, A. (2014). Ising formulations of many NP problems. *Frontiers in Physics*, Volume 2, Article 5.
- [FIA] Facchetti, G., Iacono G., and Altafini C. (2011). Computing global structural balance in large-scale signed social networks. *PNAS*, 108, no. 52, 20953-20958
- [NX] Aric A. Hagberg, Daniel A. Schult and Pieter J. Swart, “Exploring network structure, dynamics, and function using NetworkX”, in *Proceedings of the 7th Python in Science Conference (SciPy2008)*, Gael Varoquaux, Travis Vaught, and Jarrod Millman (Eds), (Pasadena, CA USA), pp. 11–15, Aug 2008
- [GD] Gogate & Dechter, “A Complete Anytime Algorithm for Treewidth”, <https://arxiv.org/abs/1207.4109>
- [AL] Lucas, A. (2014). Ising formulations of many NP problems. *Frontiers in Physics*, Volume 2, Article 5.
- [FIA] Facchetti, G., Iacono G., and Altafini C. (2011). Computing global structural balance in large-scale signed social networks. *PNAS*, 108, no. 52, 20953-20958
- [DWMP] Dahl, E., “Programming the D-Wave: Map Coloring Problem:”, <https://www.dwavesys.com/sites/default/files/Map%20Coloring%20WP2.pdf>

a

`dwave_networkx.algorithms.canonicalization`,
6
`dwave_networkx.algorithms.clique`, 6
`dwave_networkx.algorithms.coloring`, 9
`dwave_networkx.algorithms.cover`, 12
`dwave_networkx.algorithms.elimination_ordering`,
15
`dwave_networkx.algorithms.independent_set`,
27
`dwave_networkx.algorithms.markov`, 21
`dwave_networkx.algorithms.matching`, 22
`dwave_networkx.algorithms.max_cut`, 25
`dwave_networkx.algorithms.social`, 30
`dwave_networkx.algorithms.tsp`, 33

d

`dwave_networkx`, 48
`dwave_networkx.default_sampler`, 52
`dwave_networkx.drawing.chimera_layout`,
35
`dwave_networkx.drawing.pegasus_layout`,
37

e

`dwave_networkx.exceptions`, 52

u

`dwave_networkx.utils`, 46
`dwave_networkx.utils.decorators`, 46

B

`binary_quadratic_model_sampler()` (in module `dwave_networkx.utils.decorators`), 46

C

`canonical_chimera_labeling()` (in module `dwave_networkx`), 6

`chimera_coordinates` (class in `dwave_networkx`), 48

`chimera_elimination_order()` (in module `dwave_networkx.algorithms.elimination_ordering`), 16

`chimera_graph()` (in module `dwave_networkx`), 41

`chimera_layout()` (in module `dwave_networkx.drawing.chimera_layout`), 36

`chimera_to_linear()` (`chimera_coordinates` method), 49

`clique_number()` (in module `dwave_networkx`), 8

D

`draw_chimera()` (in module `dwave_networkx.drawing.chimera_layout`), 36

`draw_pegasus()` (in module `dwave_networkx.drawing.pegasus_layout`), 37

`draw_pegasus_embedding()` (in module `dwave_networkx.drawing.pegasus_layout`), 39

`dwave_networkx` (module), 48

`dwave_networkx.algorithms.canonicalization` (module), 6

`dwave_networkx.algorithms.clique` (module), 6

`dwave_networkx.algorithms.coloring` (module), 9

`dwave_networkx.algorithms.cover` (module), 12

`dwave_networkx.algorithms.elimination_ordering` (module), 15

`dwave_networkx.algorithms.independent_set` (module), 27

`dwave_networkx.algorithms.markov` (module), 21

`dwave_networkx.algorithms.matching` (module), 22

`dwave_networkx.algorithms.max_cut` (module), 25

`dwave_networkx.algorithms.social` (module), 30

`dwave_networkx.algorithms.tsp` (module), 33

`dwave_networkx.default_sampler` (module), 52

`dwave_networkx.drawing.chimera_layout` (module), 35

`dwave_networkx.drawing.pegasus_layout` (module), 37

`dwave_networkx.exceptions` (module), 52

`dwave_networkx.utils` (module), 46

`dwave_networkx.utils.decorators` (module), 46

`DWaveNetworkXException`, 52

`DWaveNetworkXMissingSampler`, 52

E

`elimination_order_width()` (in module `dwave_networkx.algorithms.elimination_ordering`), 16

F

`find_chimera_indices()` (in module `dwave_networkx`), 49

G

`get_default_sampler()` (in module `dwave_networkx.default_sampler`), 54

I

`is_almost_simplicial()` (in module `dwave_networkx.algorithms.elimination_ordering`), 17

`is_clique()` (in module `dwave_networkx`), 8

`is_independent_set()` (in module `dwave_networkx`), 29

`is_matching()` (in module `dwave_networkx.algorithms.matching`), 24

`is_maximal_matching()` (in module `dwave_networkx.algorithms.matching`), 24

`is_simplicial()` (in module `dwave_networkx.algorithms.elimination_ordering`), 17

`is_vertex_coloring()` (in module `dwave_networkx.algorithms.coloring`), 10

`is_vertex_cover()` (in module `dwave_networkx.algorithms.cover`), 15

L

`linear_to_chimera()` (`chimera_coordinates` method), 49

`linear_to_nice()` (`pegasus_coordinates` method), 50

`linear_to_pegasus()` (`pegasus_coordinates` method), 50

M

`markov_network()` (in module `dwave_networkx`), 46

`markov_network_bqm()` (in module `dwave_networkx.algorithms.markov`), 22

`max_cardinality_heuristic()` (in module `dwave_networkx.algorithms.elimination_ordering`), 18

`maximum_clique()` (in module `dwave_networkx`), 7

`maximum_cut()` (in module `dwave_networkx.algorithms.max_cut`), 25

`maximum_independent_set()` (in module `dwave_networkx`), 28

`maximum_weighted_independent_set()` (in module `dwave_networkx`), 27

`maximum_weighted_independent_set_qubo()` (in module `dwave_networkx.algorithms.independent_set`), 30

`min_fill_heuristic()` (in module `dwave_networkx.algorithms.elimination_ordering`), 19

`min_maximal_matching()` (in module `dwave_networkx.algorithms.matching`), 23

`min_vertex_color()` (in module `dwave_networkx.algorithms.coloring`), 10

`min_vertex_color_qubo()` (in module `dwave_networkx.algorithms.coloring`), 11

`min_vertex_cover()` (in module `dwave_networkx.algorithms.cover`), 14

`min_weighted_vertex_cover()` (in module `dwave_networkx.algorithms.cover`), 13

`min_width_heuristic()` (in module `dwave_networkx.algorithms.elimination_ordering`), 19

`minor_min_width()` (in module `dwave_networkx.algorithms.elimination_ordering`), 18

N

`nice_to_linear()` (`pegasus_coordinates` method), 51

`nice_to_pegasus()` (`pegasus_coordinates` static method), 51

P

`pegasus_coordinates` (class in `dwave_networkx`), 48

`pegasus_elimination_order()` (in module `dwave_networkx.algorithms.elimination_ordering`), 20

`pegasus_graph()` (in module `dwave_networkx`), 43

`pegasus_layout()` (in module `dwave_networkx.drawing.pegasus_layout`), 39

`pegasus_node_placer_2d()` (in module `dwave_networkx.drawing.pegasus_layout`), 40

`pegasus_to_linear()` (`pegasus_coordinates` method), 51

`pegasus_to_nice()` (`pegasus_coordinates` static method), 52

S

`sample_markov_network()` (in module `dwave_networkx.algorithms.markov`), 21

`set_default_sampler()` (in module `dwave_networkx.default_sampler`), 53

`structural_imbalance()` (in module `dwave_networkx.algorithms.social`), 30

`structural_imbalance_ising()` (in module `dwave_networkx.algorithms.social`), 33

T

`traveling_salesperson()` (in module `dwave_networkx.algorithms.tsp`), 33

`traveling_salesperson_qubo()` (in module `dwave_networkx.algorithms.tsp`), 35

`treewidth_branch_and_bound()` (in module `dwave_networkx.algorithms.elimination_ordering`), 20

U

`unset_default_sampler()` (*in module `dwave_networkx.default_sampler`*), [54](#)

V

`vertex_color()` (*in module `dwave_networkx.algorithms.coloring`*), [11](#)

`vertex_color_qubo()` (*in module `dwave_networkx.algorithms.coloring`*), [12](#)

W

`weighted_maximum_cut()` (*in module `dwave_networkx.algorithms.max_cut`*), [26](#)