minorminer Documentation

Release 0.2.6

D-Wave Systems

Nov 12, 2021

CONTENTS

minorminer is a heuristic tool for minor embedding: given a minor and target graph, it tries to find a mapping that embeds the minor into the target.

CHAPTER

ONE

DOCUMENTATION

Note: This documentation is for the latest version of minorminer. Documentation for the version currently installed by dwave-ocean-sdk is here: minorminer.

1.1 Introduction

minorminer is a library of tools for finding graph minor embeddings, developed to embed Ising problems onto quantum annealers (QA). While this library can be used to find minors in arbitrary graphs, it is particularly geared towards state-of-the-art QA: problem graphs of a few to a few hundred variables, and hardware graphs of a few thousand qubits.

minorminer has both a Python and C++ API, and includes implementations of multiple embedding algorithms to best fit different problems.

1.1.1 Minor-Embedding and QPU Topology

For an introduction to minor-embedding, see Minor-Embedding.

For an introduction to the topologies of D-Wave hardware graphs, see QPU Topology. Leap users also have access to the Exploring Pegasus Jupyter Notebook that explains the architecture of D-Wave's quantum computer, Advantage, in further detail.

Minor-embedding can be done manually, though typically for very small problems only. For a walkthrough of the manual minor-embedding process, see the Constraints Example: Minor-Embedding.

1.1.2 Minor-Embedding in Ocean

Minor-embedding can also be automated through Ocean. *minorminer* is used by several Ocean embedding composites for this purpose. For details on automated (and manual) minor-embedding through Ocean, see how the *EmbeddingComposite* and *FixedEmbeddingComposite* are used in this Boolean AND Gate example.

Once an embedding has been found, D-Wave's Problem Inspector tool can be used to evaluate its quality. See Using the Problem Inspector for more information.

1.2 Reference Documentation

1.2.1 General Embedding

General embedding refers to embedding that may be useful for any type of graph.

The primary utility function, `find_embedding()`, is an implementation of the heuristic algorithm described in [1]. It accepts various optional parameters used to tune the algorithm's execution or constrain the given problem.

This implementation performs on par with tuned, non-configurable implementations while providing users with hooks to easily use the code as a basic building block in research.

[1] https://arxiv.org/abs/1406.2741

`minorminer.find_embedding()`

Heuristically attempt to find a minor-embedding of source graph S into a target graph T.

Parameters

- **S** (*iterable/NetworkX Graph*) – The source graph as an iterable of label pairs representing the edges, or a NetworkX Graph.
- **T** (*iterable/NetworkX Graph*) – The target graph as an iterable of label pairs representing the edges, or a NetworkX Graph.
- ****params** (*optional*) – See below.

Returns

When the optional parameter `return_overlap` is False (the default), the function returns a dict that maps labels in S to lists of labels in T. If the heuristic fails to find an embedding, an empty dictionary is returned.

When `return_overlap` is True, the function returns a tuple consisting of a dict that maps labels in S to lists of labels in T and a bool indicating whether or not a valid embedding was found.

When interrupted by Ctrl-C, the function returns the best embedding found so far.

Note that failure to return an embedding does not prove that no embedding exists.

Optional Parameters:

max_no_improvement (int, optional, default=10): Maximum number of failed iterations to improve the current solution, where each iteration attempts to find an embedding for each variable of S such that it is adjacent to all its neighbours.

random_seed (int, optional, default=None): Seed for the random number generator. If None, seed is set by `os.urandom()`.

timeout (int, optional, default=1000): Algorithm gives up after timeout seconds.

max_beta (double, optional, max_beta=None): Qubits are assigned weight according to a formula (beta^n) where n is the number of chains containing that qubit. This value should never be less than or equal to 1. If None, `max_beta` is effectively infinite.

tries (int, optional, default=10): Number of restart attempts before the algorithm stops. On D-WAVE 2000Q, a typical restart takes between 1 and 60 seconds.

inner_rounds (int, optional, default=None): The algorithm takes at most this many iterations between restart attempts; restart attempts are typically terminated due to `max_no_improvement`. If None, `inner_rounds` is effectively infinite.

chainlength_patience (int, optional, default=10): Maximum number of failed iterations to improve chain lengths in the current solution, where each iteration attempts to find an embedding for each variable of S such that it is adjacent to all its neighbours.

max_fill (int, optional, default=None): Restricts the number of chains that can simultaneously incorporate the same qubit during the search. Values above 63 are treated as 63. If None, `max_fill` is effectively infinite.

threads (int, optional, default=1): Maximum number of threads to use. Note that the parallelization is only advantageous where the expected degree of variables is significantly greater than the number of threads. Value must be greater than 1.

return_overlap (bool, optional, default=False): This function returns an embedding, regardless of whether or not qubits are used by multiple variables. `return_overlap` determines the function's return value. If True, a 2-tuple is returned, in which the first element is the embedding and the second element is a bool representing the embedding validity. If False, only an embedding is returned.

skip_initialization (bool, optional, default=False): Skip the initialization pass. Note that this only works if the chains passed in through `initial_chains` and `fixed_chains` are semi-valid. A semi-valid embedding is a collection of chains such that every adjacent pair of variables (u,v) has a coupler (p,q) in the hardware graph where p is in chain(u) and q is in chain(v). This can be used on a valid embedding to immediately skip to the chain length improvement phase. Another good source of semi-valid embeddings is the output of this function with the `return_overlap` parameter enabled.

verbose (int, optional, default=0): Level of output verbosity.

When set to 0: Output is quiet until the final result.

When set to 1: Output looks like this:

```
initialized
max qubit fill 3; num maxfull qubits=3
embedding trial 1
max qubit fill 2; num maxfull qubits=21
embedding trial 2
embedding trial 3
embedding trial 4
embedding trial 5
embedding found.
max chain length 4; num max chains=1
reducing chain lengths
max chain length 3; num max chains=5
```

When set to 2: Output the information for lower levels and also report progress on minor statistics (when searching for an embedding, this is when the number of maxfull qubits decreases; when improving, this is when the number of max chains decreases).

When set to 3: Report before each pass. Look here when tweaking `tries`, `inner_rounds`, and `chainlength_patience`.

When set to 4: Report additional debugging information. By default, this package is built without this functionality. In the C++ headers, this is controlled by the `CPPDEBUG` flag.

Detailed explanation of the output information:

max qubit fill: Largest number of variables represented in a qubit.

num maxfull: Number of qubits that have max overfill.

max chain length: Largest number of qubits representing a single variable.

num max chains: Number of variables that have max chain size.

interactive (bool, optional, default=False): If *logging* is None or False, the verbose output will be printed to stdout/stderr as appropriate, and keyboard interrupts will stop the embedding process and the current state will be returned to the user. Otherwise, output will be directed to the logger `logging.getLogger(minorminer.__name__)` and keyboard interrupts will be propagated back to the user. Errors will use `logger.error()`, verbosity levels 1 through 3 will use `logger.info()` and level 4 will use `logger.debug()`.

initial_chains (dict, optional): Initial chains inserted into an embedding before `fixed_chains` are placed, which occurs before the initialization pass. These can be used to restart the algorithm in a similar state to a previous embedding; for example, to improve chain length of a valid embedding or to reduce overlap in a semi-valid embedding (see `skip_initialization`) previously returned by the algorithm. Missing or empty entries are ignored. Each value in the dictionary is a list of qubit labels.

fixed_chains (dict, optional): Fixed chains inserted into an embedding before the initialization pass. As the algorithm proceeds, these chains are not allowed to change, and the qubits used by these chains are not used by other chains. Missing or empty entries are ignored. Each value in the dictionary is a list of qubit labels.

restrict_chains (dict, optional): Throughout the algorithm, it is guaranteed that chain[i] is a subset of `restrict_chains[i]` for each i, except those with missing or empty entries. Each value in the dictionary is a list of qubit labels.

suspend_chains (dict, optional): This is a metafeature that is only implemented in the Python interface. `suspend_chains[i]` is an iterable of iterables; for example, `suspend_chains[i] = [blob_1, blob_2]`, with each *blob_j* an iterable of target node labels.

This enforces the following:

```
for each suspended variable i,
    for each blob_j in the suspension of i,
        at least one qubit from blob_j will be contained in the chain for i
```

We accomplish this through the following problem transformation for each iterable *blob_j* in `suspend_chains[i]`,

- Add an auxiliary node *Zij* to both source and target graphs
- Set *fixed_chains[Zij] = [Zij]*
- Add the edge *(i,Zij)* to the source graph
- Add the edges *(q,Zij)* to the target graph for each *q* in *blob_j*

Examples

This example minor embeds a triangular source K3 graph onto a square target graph.

```
from minorminer import find_embedding

# A triangle is a minor of a square.
triangle = [(0, 1), (1, 2), (2, 0)]
square = [(0, 1), (1, 2), (2, 3), (3, 0)]

# Find an assignment of sets of square variables to the triangle variables
```

(continues on next page)

(continued from previous page)

```
embedding = find_embedding(triangle, square, random_seed=10)
print(len(embedding))  # 3, one set for each variable in the triangle
print(embedding)
# We don't know which variables will be assigned where, here are a
# couple possible outputs:
# [[0, 1], [2], [3]]
# [[3], [1, 0], [2]]
```

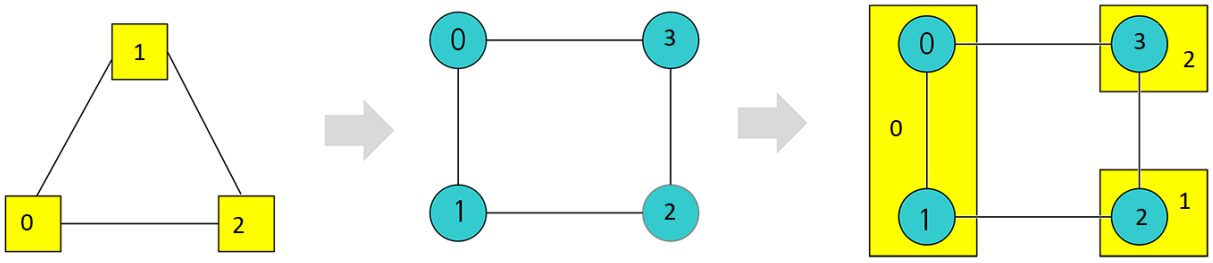


Fig. 1: Embedding a K_3 source graph into a square target graph by chaining two target nodes to represent one source node.

This minorminer execution of the example requires that source variable 0 always be assigned to target node 2.

```
embedding = find_embedding(triangle, square, fixed_chains={0: [2]})
print(embedding)
# [[2], [3, 0], [1]]
# [[2], [1], [0, 3]]
# And more, but all of them start with [2]
```

This minorminer execution of the example suggests that source variable 0 be assigned to target node 2 as a starting point for finding an embedding.

```
embedding = find_embedding(triangle, square, initial_chains={0: [2]})
print(embedding)
# [[2], [0, 3], [1]]
# [[0], [3], [1, 2]]
# Output where source variable 0 has switched to a different target node is possible.
```

This example minor embeds a fully connected K6 graph into a 30-node random regular graph of degree 3.

```
import networkx as nx

clique = nx.complete_graph(6).edges()
target_graph = nx.random_regular_graph(d=3, n=30).edges()

embedding = find_clique_embedding(clique, target_graph)

print(embedding)
```

(continues on next page)

(continued from previous page)

```
# There are many possible outputs, and sometimes it might fail
# and return an empty list
# One run returned the following embedding:
{0: [10, 9, 19, 8],
1: [18, 7, 0, 12, 27],
2: [1, 17, 22],
3: [16, 28, 4, 21, 15, 23, 25],
4: [11, 24, 13],
5: [2, 14, 26, 5, 3]}
```

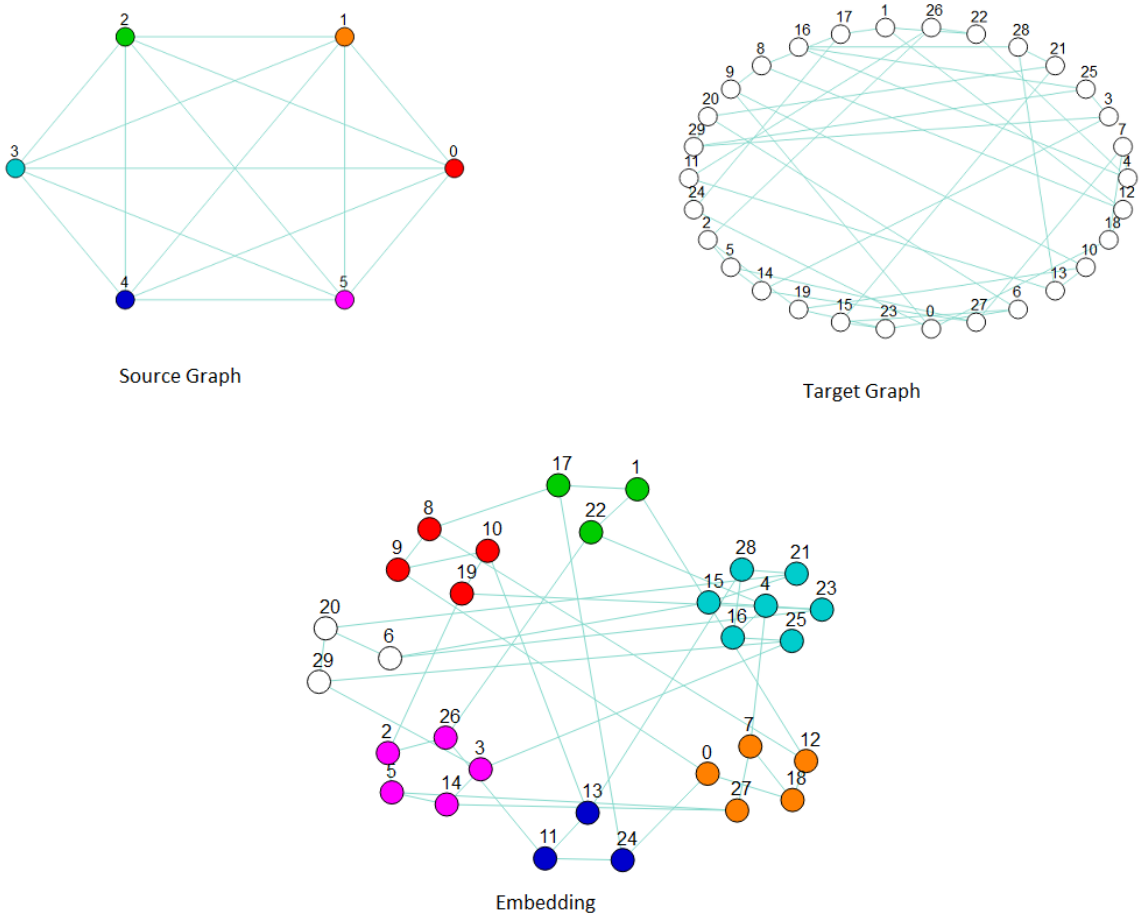


Fig. 2: Embedding a K_6 source graph (upper left) into a 30-node random target graph of degree 3 (upper right) by chaining several target nodes to represent one source node (bottom). The graphic of the embedding clusters chains representing nodes in the source graph: the cluster of red nodes is a chain of target nodes that represent source node 0, the orange nodes represent source node 1, and so on.

1.2.2 Clique Embedding

If your source graph is a clique and your target graph is either a Chimera or Pegasus graph, *find_clique_embedding()* may produce better embeddings than the generic *find_embedding()* method.

minorminer.busclique.**find_clique_embedding**()

Finds a clique embedding in the graph g using a polynomial-time algorithm.

Parameters

- **nodes** (*int/iterable*) – A number (indicating the size of the desired clique) or an iterable (specifying the node labels of the desired clique).
- **g** (*NetworkX Graph*) – The target graph that is either a dwave_networkx.chimera_graph() or dwave_networkx.pegasus_graph().
- **use_cache** (*bool, optional, default=True*) – Whether or not to compute/restore a cache of clique embeddings for g. Note that this function only uses the filesystem cache, and does not maintain the cache in memory. If many (or even several) embeddings are desired in a single session, it is recommended to use *busgraph_cache*.
- **seed** (*int, optional*) – A seed for an internal random number generator. If use_cache is True, then the seed defaults to an internally-defined value which is consistent between runs. Otherwise, a seed is generated from the current python random state.

Returns An embedding of node labels (either nodes, or range(nodes)) mapped to chains of a clique embedding.

Return type dict

Note: Due to internal optimizations, not all Chimera graphs are supported by this code. Specifically, the graphs dwave_networkx.chimera_graph(m, n, t)() are only supported for $t <= 8$. The code currently supports D-Wave products, which have $t = 4$, but not all graphs. For graphs with $t > 8$, use the legacy chimera-embedding package.

Note: When the cache is used, clique embeddings of all sizes are computed and cached. This takes somewhat longer than a single embedding, but tends to pay off after a fairly small number of calls. An exceptional use case is when there are a large number of missing internal couplers, where the result is nondeterministic – avoiding the cache in this case may be preferable.

Caching

If multiple clique or biclique embeddings need to be computed for a single Chimera or Pegasus graph, it may be more efficient to retrieve these embeddings through the *busgraph_cache*, which creates LRU file-caches for the target graph's cliques and bicliques.

Class

class minorminer.busclique.**busgraph_cache**(*g*, *seed=0*)

A cache class for Chimera, Pegasus and Zephyr graphs, and their associated cliques and bicliques.

The cache files are stored in a directory determined by *homebase* (use *busgraph_cache.cache_rootdir()* to retrieve the path to this directory). Subdirectories named *cliques* and *bicliques* are then created to store the respective caches in each.

Parameters **g** (*NetworkX Graph*) –

A dwave_networkx.pegasus_graph() or dwave_networkx.chimera_graph(). or dwave_networkx.zephyr_graph().

Note: Due to internal optimizations, not all Chimera graphs are supported by this code. Specifically, the graphs dwave_networkx.chimera_graph(m, n, t)() are only supported for $t <= 8$. The code currently supports D-Wave products, which have $t = 4$, but not all graphs. For graphs with $t > 8$, use the legacy chimera-embedding package.

Methods

busgraph_cache.cache_rootdir([version])	Returns the directory corresponding to the provided cache version.
busgraph_cache.clear_all_caches()	Removes all caches created by this class, up to and including the current version.
busgraph_cache.find_biclique_embedding(nn, mm)	Returns a biclique embedding, minimizing the maximum chain length given its size.
busgraph_cache.find_clique_embedding(nn)	Returns a clique embedding, minimizing the maximum chainlength given its size.
busgraph_cache.largest_balanced_biclique()	Returns the largest-size biclique where both sides have equal size.
busgraph_cache.largest_clique()	Returns the largest-found clique in the clique cache.
busgraph_cache.largest_clique_by_chainlength(c)	Returns the largest-found clique in the clique cache, with a specified maximum chainlength.

minorminer.busclique.busgraph_cache.cache_rootdir

static busgraph_cache.**cache_rootdir**(*version=6*)

Returns the directory corresponding to the provided cache version.

Parameters **version** (*int, optional, default=current cache version*) – Cache version.

Returns str

minorminer.busclique.busgraph_cache.clear_all_caches

static busgraph_cache.**clear_all_caches**()

Removes all caches created by this class, up to and including the current version.

Returns None

minorminer.busclique.busgraph_cache.find_biclique_embedding

busgraph_cache.**find_biclique_embedding**(*nn*, *mm*)

Returns a biclique embedding, minimizing the maximum chain length given its size.

This will compute the entire biclique cache if it is missing from the filesystem.

Parameters

- **nn** (*int/iterable*) – A number (indicating the size of one side of the desired biclique) or an iterable (specifying the node labels of one side the desired biclique).
- **mm** (*int/iterable*) – Same as nn, for the other side of the desired biclique.

In the case that nn is a number, the first side will have nodes labeled from `range(nn)`. In the case that mm is a number, the second side will have nodes labeled from `range(n, n + mm)`; where n is either nn or `len(nn)`.

Returns An embedding of node labels (described above) mapped to chains of a biclique embedding.

Return type dict

minorminer.busclique.busgraph_cache.find_clique_embedding

busgraph_cache.**find_clique_embedding**(*nn*)

Returns a clique embedding, minimizing the maximum chainlength given its size.

This will compute the entire clique cache if it is missing from the filesystem.

Parameters **nn** (*int/iterable*) – A number (indicating the size of the desired clique) or an iterable (specifying the node labels of the desired clique).

Returns An embedding of node labels (either nn, or `range(nn)`) mapped to chains of a clique embedding.

Return type dict

minorminer.busclique.busgraph_cache.largest_balanced_biclique

busgraph_cache.**largest_balanced_biclique**()

Returns the largest-size biclique where both sides have equal size.

Nodes of the embedding dict are from `range(len(embedding))`, where the nodes `range(len(embedding)/2)` are completely connected to the nodes `range(len(embedding)/2, len(embedding))`.

This will compute the entire biclique cache if it is missing from the filesystem.

Returns An embedding of node labels (described above) mapped to chains of the largest balanced biclique.

Return type dict

minorminer.busclique.busgraph_cache.largest_clique

busgraph_cache.**largest_clique**()
Returns the largest-found clique in the clique cache.

This will compute the entire clique cache if it is missing from the filesystem.

Returns An embedding of node labels from `range(len(embedding))` mapped to chains of the largest-found clique.

Return type dict

minorminer.busclique.busgraph_cache.largest_clique_by_chainlength

busgraph_cache.**largest_clique_by_chainlength**(*chainlength*)
Returns the largest-found clique in the clique cache, with a specified maximum `chainlength`.

This will compute the entire clique cache if it is missing from the filesystem.

Parameters **chainlength** (*int*) – Max chain length.

Returns An embedding of node labels from `range(len(embedding))` mapped to chains of the largest-found clique with maximum `chainlength`.

Return type dict

Examples

This example minor embeds a source clique of size 5 into a target Chimera graph.

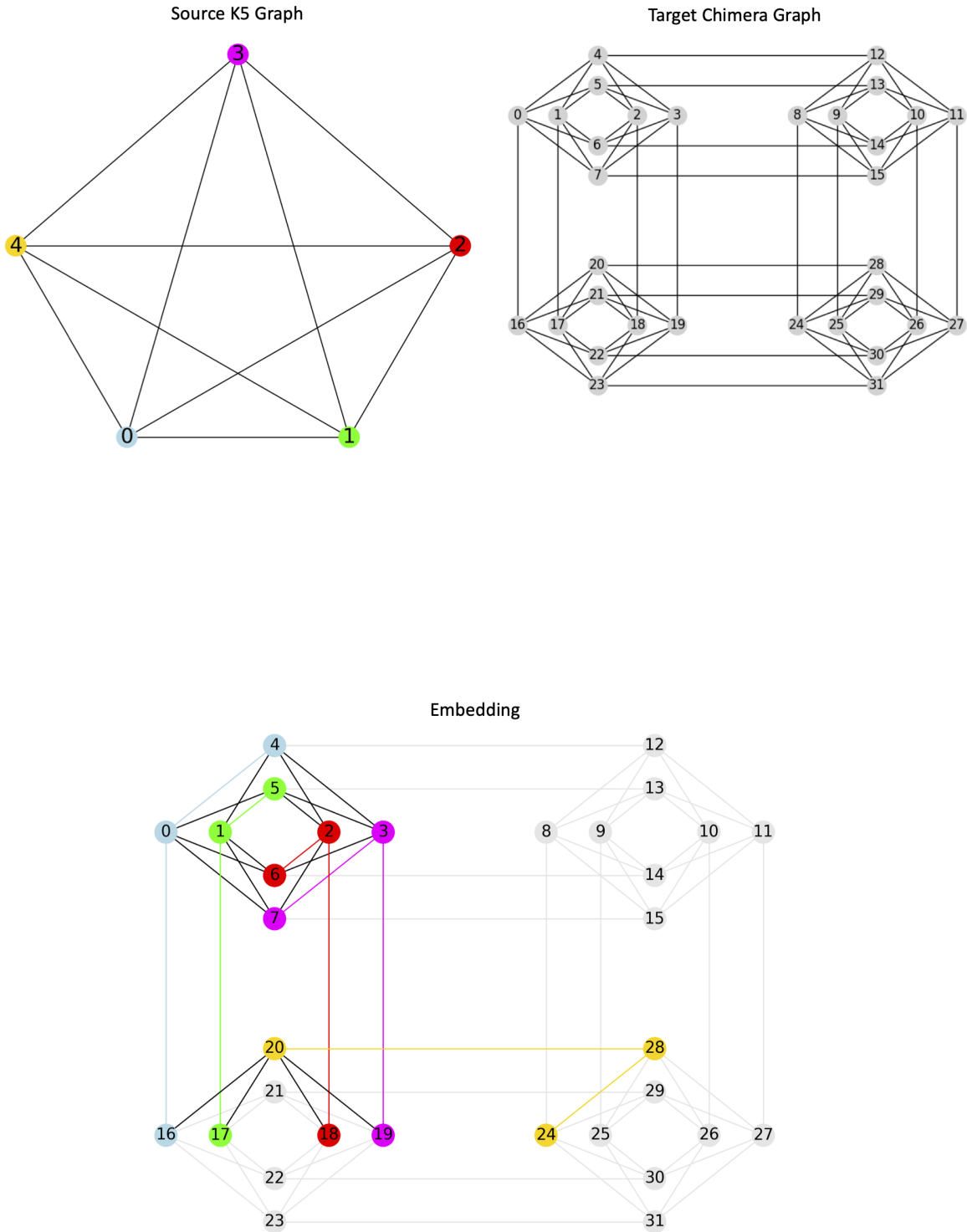
```
from minorminer import busclique
import dwave_networkx as dnx

C = dnx.chimera_graph(2, 2)
embedding = busclique.find_clique_embedding(5, C)

print(embedding)
{0: (0, 16, 4), 1: (1, 17, 5), 2: (2, 18, 6), 3: (3, 19, 7), 4: (24, 20, 28)}
```

1.2.3 Layout Embedding

minorminer.layout.find_embedding() offers a more specialized approach to find an embedding through the use of the *Layout* and *Placement* classes. This kind of embedding may be useful when the underlying data of your source graph is spatial. It can also be useful for embedding graphs with nodes of a low degree (i.e., a cubic graph).

minorminer.layout.**find_embedding**(*S*, *T*, *layout=(<function p_norm>*, *None)*, *placement=<function closest>*, *mm_hint_type='initial_chains'*, *return_layouts=False*, ***kwargs*)
Tries to embed S in T by computing layout-aware chains and passing them to *minorminer.find_embedding()*. Chains are passed as either *initial_chains* or *suspend_chains* (see documentation for *minorminer.find_embedding()* to learn more).

Parameters


Source K5 Graph
Target Chimera Graph
Embedding

- **S** (*NetworkX Graph/edges data structure (dict, list, ...)*) – The source graph being embedded or a NetworkX supported data structure for edges (see `nx.convert.to_networkx_graph()` for details).
- **T** (*NetworkX Graph/edges data structure (dict, list, ...)*) – The target graph being embedded into or a NetworkX supported data structure for edges (see `nx.convert.to_networkx_graph()` for details).
- **layout** (*function/(function/dict/Layout, function/dict/Layout), optional*) – A function to compute the *Layout* for both S and T, or a 2-tuple that either consists of a pair of functions or pre-computed layouts (in the form of *Layout* or dicts). The first entry in the 2-tuple applies to S while the second applies to T.

  Note: If `layout` is a single function and T is a dnx_graph, then the function passed in is only applied to S and the dnx_layout is applied to T. To run a layout function explicitly on T, pass it in as a 2-tuple; i.e. (p_norm, p_norm).

- **placement** (*function/dict, optional, default=minorminer.placement.closest*) – A function that uses the layouts of S and T to map the vertices of S to subsets of vertices of T (*Placement*), or a dict that contains the precomputed mapping/*Placement*.

  By default, *closest()* is called to compute placement.
- **mm_hint_type** (*str, optional, default="initial_chains"*) – This is the hint type passed to *minorminer.find_embedding()*. Supported types are "initial_chains" and "suspend_chains". See *minorminer.find_embedding()* for more information.
- **return_layouts** (*bool, optional, default=False*) – If True, layout objects of S and T are also returned.
- ****kwargs** (*dict*) – Keyword arguments passed to *Layout*, *Placement* or *minorminer.find_embedding()*.

Returns An embedding of vertices of S (keys) to chains in T (values). This embedding is dependent on the kwargs being passed in. If `return_layouts` is True, a 2-tuple is returned in which the first element is the embedding dict and the second element is another 2-tuple containing the source and target *Layout* objects.

Return type dict

Examples

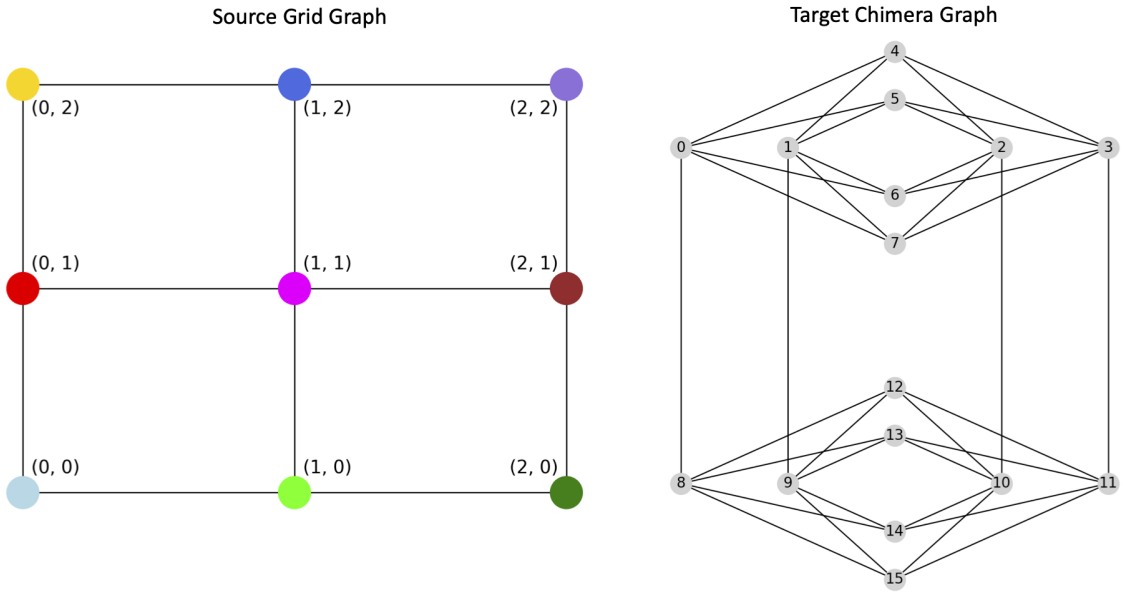
This example minor embeds a 3x3 grid graph onto a Chimera graph.

```
import networkx as nx
import dwave_networkx as dnx
import minorminer.layout as mml

grid_graph = nx.generators.lattice.grid_2d_graph(3, 3)
C = dnx.chimera_graph(2,1)

embedding = mml.find_embedding(grid_graph, C)
print(embedding)
# There are many possible outputs, and sometimes it might fail
```

(continues on next page)

(continued from previous page)

```
# and return an empty list
# One run returned the following embedding:
{(0, 0): [13], (1, 0): [0, 8], (0, 1): [9], (1, 1): [12], (0, 2): [14], (1, 2): [10], (2,
↪ 0): [7], (2, 1): [11, 3], (2, 2): [15]}
```



Layout

class minorminer.layout.layout.**Layout**(*G*, *layout=None*, *dim=None*, *center=None*, *scale=None*, *pack_components=True*, ***kwargs*)

Class that stores (or computes) coordinates in dimension `dim` for each node in graph `G`.

Parameters

- **G** (*NetworkX Graph/edges data structure (dict, list, ...)*) – The graph to compute the layout for or a NetworkX supported data structure for edges (see `nx.convert.to_networkx_graph()` for details).
- **layout** (*dict/function, optional, default=None*) – If a dict, this specifies a pre-computed layout for `G`.

  If a function, this should be in the form of `layout(G, **kwargs)`, in which `dim`, `center`, and `scale` are passed in as kwargs and the return value is a dict representing a layout of `G`.

  If None, `nx.random_layout(G, **kwargs)()` is called.
- **dim** (*int, optional, default=None*) – The desired dimension of the computed layout, $R^{\dim}$. If None, `dim` is set as the dimension of `layout`.
- **center** (*tuple, optional, default=None*) – The desired center point of the computed layout. If None, `center` is set as the center of `layout`.


_images/layout_embedding.png

- **scale** (*float, optional, default=None*) – The desired scale of the computed layout; i.e. the layout is in $[center - scale, center + scale]^d$ space. If None, `scale` is set to be the scale of `layout`.
- **pack_components** (*bool, optional, default=True*) – If True, and if the graph contains multiple components and `dim` is None or 2, the components will be laid out individually and packed into a rectangle.
- ****kwargs** (*dict*) – Keyword arguments are passed to `layout` if it is a function.

Functions for Creating Layouts

minorminer.layout.layout.**p_norm**(*G*, *p=2*, *starting_layout=None*, *G_distances=None*, *dim=None*, *center=None*, *scale=None*, ***kwargs*)

Embeds graph `G` in R^d with the p-norm and minimizes a Kamada-Kawai-esque objective function to achieve an embedding with low distortion.

This computes a *Layout* for `G` where the graph distance and the p-distance are very close to each other.

By default, *p_norm()* is used to compute the layout for source graphs when *minorminer.layout.find_embedding()* is called.

Parameters

- **G** (*NetworkX Graph*) – The graph to compute the layout for.
- **p** (*int, optional, default=2*) – The order of the p-norm to use as a metric.
- **starting_layout** (*dict, optional, default=None*) – A mapping from the vertices of `G` to points in R^d. If None, `nx.spectral_layout()` is used if possible. Otherwise, `nx.random_layout()` is used.
- **G_distances** (*dict, optional, default=None*) – A dictionary of dictionaries representing distances from every vertex in `G` to every other vertex in `G`. If None, it is computed.
- **dim** (*int, optional, default=None*) – The desired dimension of the returned layout, $R^{\dim}$. If None, the dimension of `center` is used. If `center` is None, `dim` is set to 2.
- **center** (*tuple, optional, default=None*) – The desired center point of the returned layout. If None, `center` is set as the origin in $R^{\dim}$ space.
- **scale** (*float, optional, default=None*) – The desired scale of the returned layout; i.e. the layout is in $[center - scale, center + scale]^d$ space. If None, no scale is set.

Returns `Layout.layout`, a mapping from vertices of `G` (keys) to points in R^d (values).

Return type dict

Examples

This example creates a *Layout* object for a hexagonal lattice graph, with coordinates computed using *p_norm()*.

```
>>> import networkx as nx
>>> import minorminer.layout as mml
...
>>> G = nx.hexagonal_lattice_graph(2,2)
>>> layout = mml.Layout(G, mml.p_norm, center=(1,1))
```

`layout` may be passed in directly to *minorminer.layout.find_embedding()*.

Alternatively, *p_norm()* may be passed in instead.

This next example finds an embedding of a hexagonal lattice graph on a Chimera graph, in which the layouts of both the source and target graphs are computed using p_norm.

```
>>> import networkx as nx
>>> import dwave_networkx as dnx
>>> import minorminer.layout as mml
...
>>> G = nx.hexagonal_lattice_graph(2,2)
>>> C = dnx.chimera_graph(2,2)
>>> embedding = mml.find_embedding(G,
...                                C,
...                                layout=(mml.p_norm, mml.p_norm),
...                                center=(1,1))
```

minorminer.layout.layout.**dnx_layout**(*G*, *dim=None*, *center=None*, *scale=None*, ***kwargs*)

The Chimera or Pegasus layout from *dwave_networkx* centered at the origin with `scale` as a function of the number of rows or columns. Note: As per the implementation of *dnx.*_layout*, if $dim > 2$, coordinates beyond the second are 0.

By default, *dnx_layout()* is used to compute the layout for target Chimera or Pegasus graphs when *minorminer.layout.find_embedding()* is called.

Parameters

- **G** (*NetworkX Graph*) – The graph to compute the layout for.
- **dim** (*int, optional, default=None*) – The desired dimension of the returned layout, $R^{\dim}$. If None, the dimension of `center` is used. If `center` is None, `dim` is set to 2.
- **center** (*tuple, optional, default=None*) – The desired center point of the returned layout. If None, it is set as the origin in $R^{\dim}$ space.
- **scale** (*float, optional, default=None*) – The desired scale of the returned layout; i.e. the layout is in $[center-scale, center+scale]^d$ space. If None, it is set as $max(n, m)/2$, where n, m are the number of columns, rows respectively in `G`.

Returns `Layout.layout`, a mapping from vertices of `G` (keys) to points in R^d (values).

Return type dict

Examples

This example creates a *Layout* object for a Pegasus graph, with coordinates computed using *dnx_layout()*.

```
>>> import networkx as nx
>>> import minorminer.layout as mml
...
>>> P = dnx.pegasus_graph(4)
>>> layout = mml.Layout(P, mml.dnx_layout, center=(1,1), scale=2)
```

Placement

class minorminer.layout.placement.**Placement**(*S_layout*, *T_layout*, *placement=None*, *scale_ratio=None*, ***kwargs*)

Class that stores (or computes) a mapping of source nodes to collections of target nodes without any constraints. In mathematical terms, map V(S) to $2^{V(T)}$.

Parameters

- **S_layout** (*Layout*) – A layout for S; i.e. a map from S to R^d.
- **T_layout** (*Layout*) – A layout for T; i.e. a map from T to R^d.
- **placement** (*dict/function, optional, default=None*) – If a dict, this specifies a pre-computed placement for S in T.

  If a function, the function is called on `S_layout` and `T_layout`, `placement(S_layout, T_layout)`, and should return a dict representing a placement of S in T.

  If None, a random placement of S in T is selected.
- **scale_ratio** (*float, optional, default=None*) – If None, `S_layout` is not scaled. Otherwise, `S_layout` is scaled to `scale_ratio*T_layout.scale`.
- ****kwargs** (*dict*) – Keyword arguments are passed to `placement` if it is a function.

Functions for Creating Placements

minorminer.layout.placement.**intersection**(*S_layout*, *T_layout*, ***kwargs*)

Map each vertex of S to its nearest row/column intersection qubit in T (T must be a D-Wave hardware graph). Note: This will modify `S_layout`.

Parameters

- **S_layout** (*Layout*) – A layout for S; i.e. a map from S to R^d.
- **T_layout** (*Layout*) – A layout for T; i.e. a map from T to R^d.

Returns A mapping from vertices of S (keys) to vertices of T (values).

Return type dict

Examples

This example creates a *Placement* object that stores a mapping computed with *intersection()*, in which the nodes from a source hexagonal lattice graph are mapped to a target Chimera graph.

```
>>> import networkx as nx
>>> import dwave_networkx as dnx
>>> import minorminer.layout as mml
...
>>> G = nx.hexagonal_lattice_graph(2,2)
>>> G_layout = mml.Layout(G, mml.p_norm)
>>> C = dnx.chimera_graph(2,2)
>>> C_layout = mml.Layout(C, mml.dnx_layout)
>>> placement = mml.Placement(G_layout, C_layout, placement=mml.intersection)
```

placement may be passed in directly to *minorminer.layout.find_embedding()*.

Alternatively, *intersection()* may be passed in instead, as shown in the example below.

```
>>> import networkx as nx
>>> import dwave_networkx as dnx
>>> import minorminer.layout as mml
...
>>> G = nx.hexagonal_lattice_graph(2,2)
>>> C = dnx.chimera_graph(2,2)
>>> embedding = mml.find_embedding(G,
...                                C,
...                                placement=mml.intersection)
```

minorminer.layout.placement.**closest**(*S_layout*, *T_layout*, *subset_size=(1, 1)*, *num_neighbors=1*, ***kwargs*)

Maps vertices of S to the closest vertices of T as given by S_layout and T_layout. i.e. For each vertex u in S_layout and each vertex v in T_layout, map u to the v with minimum Euclidean distance ($||u - v||_2$).

By default, *closest()* is used to compute the placement of an embedding when *minorminer.layout.find_embedding()* is called.

Parameters

- **S_layout** (*Layout*) – A layout for S; i.e. a map from S to R^d.
- **T_layout** (*Layout*) – A layout for T; i.e. a map from T to R^d.
- **subset_size** (*tuple, optional, default=(1, 1)*) – A lower (subset_size[0]) and upper (subset_size[1]) bound on the size of subsets of T that will be considered when mapping vertices of S.
- **num_neighbors** (*int, optional, default=1*) – The number of closest neighbors to query from the KDTree–the neighbor with minimum overlap is chosen. Increasing this reduces overlap, but increases runtime.

Returns A mapping from vertices of S (keys) to subsets of vertices of T (values).

Return type dict

Examples

This example creates a *Placement* object that stores a mapping computed with *closest()*, in which the nodes from a source hexagonal lattice graph are mapped to a target Chimera graph.

```
>>> import networkx as nx
>>> import dwave_networkx as dnx
>>> import minorminer.layout as mml
...
>>> G = nx.hexagonal_lattice_graph(2,2)
>>> G_layout = mml.Layout(G, mml.p_norm)
>>> C = dnx.chimera_graph(2,2)
>>> C_layout = mml.Layout(C, mml.dnx_layout)
>>> placement = mml.Placement(G_layout, C_layout, placement=mml.closest)
```

1.2.4 C++ Library

Namespace list

Namespace busclique

namespace **busclique**

Typedefs

using **biclique_result_cache** = std::unordered_map<pair<size_t, size_t>, value_t, *craphash*>

using **chimera_spec** = *topo_spec_cellmask*<*chimera_spec_base*>

using **pegasus_spec** = *topo_spec_cellmask*<*pegasus_spec_base*>

using **zephyr_spec** = *topo_spec_cellmask*<*zephyr_spec_base*>

Enums

enum **corner**
Values:

enumerator **NW**

enumerator **NE**

enumerator **SW**

enumerator **SE**

enumerator **NWskip**

enumerator **NEskip**

enumerator **SWskip**

enumerator **SEskip**

enumerator **skipmask**

enumerator **shift**

enumerator **mask**

enumerator **none**

Functions

template<typename **topo_spec**>
void **best_bicliques**(const *topo_spec* &topo, const vector<size_t> &nodes, const vector<pair<size_t, size_t>> &edges, vector<pair<pair<size_t, size_t>, vector<vector<size_t>>>> &embs)

template<typename **topo_spec**>
void **best_bicliques**(*topo_cache*<*topo_spec*> &topology, vector<pair<pair<size_t, size_t>, vector<vector<size_t>>>> &embs)

template<typename **T**>
size_t **get_maxlen**(vector<*T*> &emb, size_t size)

template<typename **topo_spec**>
bool **find_clique_nice**(const *cell_cache*<*topo_spec*>&, size_t size, vector<vector<size_t>> &emb, size_t &min_width, size_t &max_width, size_t &max_length)

template<>
bool **find_clique_nice**(const *cell_cache*<*chimera_spec*> &cells, size_t size, vector<vector<size_t>> &emb, size_t&, size_t&, size_t &max_length)

template<typename **clique_cache_t**>
size_t **check_sol**(const *clique_cache_t* &rects, vector<vector<size_t>> &emb, size_t size)

template<>
bool **find_clique_nice**(const *cell_cache*<*zephyr_spec*> &cells, size_t size, vector<vector<size_t>> &emb, size_t&, size_t&, size_t &max_length)

template<>
bool **find_clique_nice**(const *cell_cache*<*pegasus_spec*> &cells, size_t size, vector<vector<size_t>> &emb, size_t&, size_t&, size_t &max_length)

template<typename **topo_spec**>
bool **find_clique**(const *topo_spec* &topo, const vector<size_t> &nodes, const vector<pair<size_t, size_t>> &edges, size_t size, vector<vector<size_t>> &emb)

template<typename **topo_spec**>
bool **find_clique**(*topo_cache*<*topo_spec*> &topology, size_t size, vector<vector<size_t>> &emb)

template<typename **topo_spec**>
bool **find_clique_nice**(const *topo_spec* &topo, const vector<size_t> &nodes, const vector<pair<size_t, size_t>> &edges, size_t size, vector<vector<size_t>> &emb)

template<typename **topo_spec**>
void **short_clique**(const *topo_spec*&, const vector<size_t> &nodes, const vector<pair<size_t, size_t>> &edges, vector<vector<size_t>> &emb)

template<typename **topo_spec**>
void **best_cliques**(*topo_cache*<*topo_spec*> &topology, vector<vector<vector<size_t>>> &embs,
vector<vector<size_t>> &emb_1)

bool **find_generic_1**(const vector<size_t> &nodes, vector<vector<size_t>> &emb)

bool **find_generic_2**(const vector<pair<size_t, size_t>> &edges, vector<vector<size_t>> &emb)

bool **find_generic_3**(const vector<pair<size_t, size_t>> &edges, vector<vector<size_t>> &emb)

bool **find_generic_4**(const vector<pair<size_t, size_t>> &edges, vector<vector<size_t>> &emb)

inline size_t **binom**(size_t x)

template<typename **T**>
inline size_t **_serial_helper**(*serialize_size_tag*, uint8_t*, const *fat_pointer*<*T*> &value)

_serial_helper both computes the size of a field of an object being serialized, and also writes it into an output buffer, advancing its buffer.

The construction provides a single source of truth for the serialization of an object field or data pointer.

template<typename **T**>
inline size_t **_serial_helper**(*serialize_size_tag*, uint8_t*, const *T* &value)

template<typename **T**>
const *T* ***_serial_addr**(const *fat_pointer*<*T*> &value)

template<typename **T**>
const *T* ***_serial_addr**(const *T* &value)

template<typename **serialize_tag**, typename **T**>
inline size_t **_serial_helper**(*serialize_tag*, uint8_t *output, const *T* &value)

template<typename **serialize_tag**, typename...>
inline size_t **_serialize**(*serialize_tag*, uint8_t*)

_serialize computes the size of a sequence of fields associated with an object being serialized, and also writes those fields into an output buffer advancing the buffer by the corresponding amount.

This provides a single source of truth for the serialization of a class.

template<typename **serialize_tag**, typename **T**, typename ...**Args**>
inline size_t **_serialize**(*serialize_tag*, uint8_t *output, const *T* &value, const *Args*&... args)

Variables

const vector<vector<size_t>> **empty_emb**

const uint8_t **popcount**[256] = {0, 1, 1, 2, 1, 2, 2, 3, 1, 2, 2, 3, 2, 3, 3, 4, 1, 2, 2, 3, 2, 3, 3, 4, 2, 3, 3, 4, 3, 4, 4, 5, 1, 2, 2, 3, 2, 3, 3, 4, 2, 3, 3, 4, 3, 4, 4, 5, 2, 3, 3, 4, 3, 4, 4, 5, 3, 4, 4, 5, 4, 5, 5, 6, 1, 2, 2, 3, 2, 3, 3, 4, 2, 3, 3, 4, 3, 4, 4, 5, 2, 3, 3, 4, 3, 4, 4, 5, 3, 4, 4, 5, 4, 5, 5, 6, 2, 3, 3, 4, 3, 4, 4, 5, 3, 4, 4, 5, 4, 5, 5, 6, 3, 4, 4, 5, 4, 5, 5, 6, 4, 5, 5, 6, 5, 6, 6, 7, 1, 2, 2, 3, 2, 3, 3, 4, 2, 3, 3, 4, 3, 4, 4, 5, 2, 3, 3, 4, 3, 4, 4, 5, 3, 4, 4, 5, 4, 5, 5, 6, 2, 3, 3, 4, 3, 4, 4, 5, 3, 4, 4, 5, 4, 5, 5, 6, 3, 4, 4, 5, 4, 5, 5, 6, 4, 5, 5, 6, 5, 6, 6, 7, 2, 3, 3, 4, 3, 4, 4, 5, 3, 4, 4, 5, 4, 5, 5, 6, 3, 4, 4, 5, 4, 5, 5, 6, 4, 5, 5, 6, 5, 6, 6, 7, 3, 4, 4, 5, 4, 5, 5, 6, 4, 5, 5, 6, 5, 6, 6, 7, 4, 5, 5, 6, 5, 6, 6, 7, 5, 6, 6, 7, 6, 7, 7, 8}

const uint8_t **first_bit**[256] = {0, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0, 4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0, 5, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0, 4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0, 6, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0, 4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0, 5, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0, 4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0, 7, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0, 4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0, 5, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0, 4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0, 6, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0, 4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0, 5, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0, 4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0}

const uint8_t **mask_bit**[8] = {1, 2, 4, 8, 16, 32, 64, 128}

const uint16_t **mask_subsets**[8] = {1, 2, 4, 8, 16, 32, 64, 128}

const std::set<size_t> **_emptyset**

template<typename **topo_spec**>

class **biclique_cache**
#include <biclique_cache.hpp>

template<typename **topo_spec**>

class **biclique_yield_cache**
#include <biclique_cache.hpp>

class **iterator**
#include <biclique_cache.hpp>

template<typename **topo_spec**>

class **bundle_cache**
#include <bundle_cache.hpp>

template<typename **topo_spec**>

class **cell_cache**
#include <cell_cache.hpp>

class **chimera_spec_base** : public *busclique*::*topo_spec_base*
#include <util.hpp>

template<typename **topo_spec**>

class **clique_cache**
#include <clique_cache.hpp>

template<typename **topo_spec**>

class **clique_iterator**
#include <clique_cache.hpp>

template<typename **topo_spec**>

class **clique_yield_cache**
#include <clique_cache.hpp>

class **craphash**
#include <find_biclique.hpp>

template<typename **T**>

class **fat_pointer**
#include <util.hpp>

class **ignore_badmask**
#include <util.hpp>

class **maxcache**
#include <clique_cache.hpp>

class **pegasus_spec_base** : public *busclique*::*topo_spec_base*
#include <util.hpp>

class **populate_badmask**
#include <util.hpp>

class **serialize_size_tag**
#include <util.hpp>

class **serialize_write_tag**
#include <util.hpp>

template<typename **topo_spec**>

class **topo_cache**
#include <topo_cache.hpp>

class **topo_spec_base**
#include <util.hpp> Subclassed by *busclique::chimera_spec_base*, *busclique::pegasus_spec_base*, *busclique::zephyr_spec_base*

template<typename **topo_spec**>

class **topo_spec_cellmask** : public *topo_spec*
#include <util.hpp>

class **yieldcache**
#include <biclique_cache.hpp>

class **zephyr_spec_base** : public *busclique*::*topo_spec_base*
#include <util.hpp>

class **zerocache**
#include <clique_cache.hpp>

Namespace fastrng

namespace **fastrng**

class **fastrng**
#include <fastrng.hpp>

Namespace find_embedding

namespace **find_embedding**

Typedefs

using **distance_t** = long long int

using **RANDOM** = *fastrng*

using **clock** = std::chrono::high_resolution_clock

using **min_queue** = std::priority_queue<*priority_node*<P, *min_heap_tag*>>

using **max_queue** = std::priority_queue<*priority_node*<P, *max_heap_tag*>>

using **distance_queue** = *pairing_queue*<*priority_node*<*distance_t*, *min_heap_tag*>>

typedef shared_ptr<*LocalInteraction*> **LocalInteractionPtr**

Enums

enum **VARORDER**
Values:

enumerator **VARORDER_SHUFFLE**

enumerator **VARORDER_DFS**

enumerator **VARORDER_BFS**

enumerator **VARORDER_PFS**

enumerator **VARORDER_RPFS**

enumerator **VARORDER_KEEP**

Functions

int **findEmbedding**(*graph*::*input_graph* &var_g, *graph*::*input_graph* &qubit_g, *optional_parameters* &params, vector<vector<int>> &chains)

The main entry function of this library.

This method primarily dispatches the proper implementation of the algorithm where some parameters/behaviours have been fixed at compile time.

In terms of dispatch, there are three dynamically-selected classes which are combined, each according to a specific optional parameter.

- a domain_handler, described in embedding_problem.hpp, manages constraints of the form "variable a's chain must be a subset of…"
- a fixed_handler, described in embedding_problem.hpp, manages contstraints of the form "variable a's chain must be exactly…"
- a pathfinder, described in pathfinder.hpp, which come in two flavors, serial and parallel The optional parameters themselves can be found in util.hpp. Respectively, the controlling options for the above are restrict_chains, fixed_chains, and threads.

template<typename **T**>
void **collectMinima**(const vector<*T*> &input, vector<int> &output)

Fill output with the index of all of the minimum and equal values in input.

Variables

constexpr *distance_t* **max_distance** = numeric_limits<*distance_t*>::max()

class **BadInitializationException** : public *minorminer*::*MinorMinerException*

#include <util.hpp>

class **chain**

#include <chain.hpp>

Public Functions

inline **chain**(vector<int> &w, int l)

construct this chain, linking it to the qubit_weight vector `w` (common to all chains in an embedding, typically) and setting its variable label `l`

inline *chain* &**operator=**(const vector<int> &c)

assign this to a vector of ints.

each incoming qubit will have itself as a parent.

inline *chain* &**operator=**(const *chain* &c)

assign this to another chain

inline size_t **size**() const

number of qubits in chain

inline size_t **count**(const int q) const

returns 0 if `q` is not contained in `this`, 1 otherwise

inline int **get_link**(const int x) const
: get the qubit, in `this`, which links `this` to the chain of x (if x==label, interpret the linking qubit as the chain's root)

inline void **set_link**(const int x, const int q)
: set the qubit, in `this`, which links `this` to the chain of x (if x==label, interpret the linking qubit as the chain's root)

inline int **drop_link**(const int x)
: discard and return the linking qubit for `x`, or -1 if that link is not set

inline void **set_root**(const int q)
: insert the qubit `q` into `this`, and set `q` to be the root (represented as the linking qubit for `label`)

inline void **clear**()
: empty this data structure

inline void **add_leaf**(const int q, const int parent)
: add the qubit `q` as a leaf, with `parent` as its parent

inline int **trim_branch**(int q)
: try to delete the qubit `q` from this chain, and keep deleting until no more qubits are free to be deleted.

  return the first ancestor which cannot be deleted

inline int **trim_leaf**(int q)
: try to delete the qubit `q` from this chain.

  if `q` cannot be deleted, return it; otherwise return its parent

inline int **parent**(const int q) const
: the parent of `q` in this chain which might be `q` but otherwise cycles should be impossible

inline void **adopt**(const int p, const int q)
: assign `p` to be the parent of `q`, on condition that both `p` and `q` are contained in `this`, `q` is its own parent, and `q` is not the root

inline int **refcount**(const int q) const
: return the number of references that `this` makes to the qubit `q` where a "reference" is an occurrence of `q` as a parent or an occurrence of `q` as a linking qubit / root

inline size_t **freeze**(vector<*chain*> &others, *frozen_chain* &keep)
: store this chain into a *frozen_chain*, unlink all chains from this, and *clear()*

inline void **thaw**(vector<*chain*> &others, *frozen_chain* &keep)
: restore a *frozen_chain* into this, re-establishing links from other chains.

  precondition: this is empty.

template<typename **embedding_problem_t**>
inline void **steal**(*chain* &other, *embedding_problem_t* &ep, int chainsize = 0)
: assumes `this` and `other` have links for eachother's labels steals all qubits from `other` which are available to be taken by `this`; starting with the qubit links and updating qubit links after all

inline void **link_path**(*chain* &other, int q, const vector<int> &parents)
: link this chain to another, following the path `q`, `parent[q]`, `parent[parent[q]]`, ...

  from `this` to `other` and intermediate nodes (all but the last) into `this` (preconditions: `this` and `other` are not linked, `q` is contained in `this`, and the parent-path is eventually contained in `other`)

inline *iterator* **begin**() const
: iterator pointing to the first qubit in this chain

inline *iterator* **end**() const
iterator pointing to the end of this chain

inline void **diagnostic**()
run the diagnostic, and if it fails, report the failure to the user and throw a *CorruptEmbeddingException*.

the `last_op` argument is used in the error message

inline int **run_diagnostic**() const
run the diagnostic and return a nonzero status `r` in case of failure if(`r`&1), then the parent of a qubit is not contained in this chain if(`r`&2), then there is a refcounting error in this chain

class **iterator**
#include <chain.hpp>

class **CorruptEmbeddingException** : public *minorminer*::*MinorMinerException*
#include <util.hpp>

class **CorruptParametersException** : public *minorminer*::*MinorMinerException*
#include <util.hpp>

class **domain_handler_masked**
#include <embedding_problem.hpp> this domain handler stores masks for each variable so that prepare_visited and prepare_distances are barely more expensive than a memcopy

class **domain_handler_universe**
#include <embedding_problem.hpp> this is the trivial domain handler, where every variable is allowed to use every qubit

template<typename **embedding_problem_t**>

class **embedding**
#include <embedding.hpp> This class is how we represent and manipulate embedding objects, using as much encapsulation as possible.

We provide methods to view and modify chains.

Public Functions

inline **embedding**(*embedding_problem_t* &e_p)
constructor for an empty embedding

inline **embedding**(*embedding_problem_t* &e_p, map<int, vector<int>> &fixed_chains, map<int, vector<int>> &initial_chains)
constructor for an initial embedding: accepts fixed and initial chains, populates the embedding based on them, and attempts to link adjacent chains together.

inline *embedding*<*embedding_problem_t*> &**operator=**(const *embedding*<*embedding_problem_t*> &other)
copy the data from `other.var_embedding` into `this.var_embedding`

inline const *chain* &**get_chain**(int v) const
Get the variables in a chain.

inline unsigned int **chainsize**(int v) const
Get the size of a chain.

inline int **weight**(int q) const
Get the weight of a qubit.

inline int **max_weight**() const
Get the maximum of all qubit weights.

inline int **max_weight**(const int start, const int stop) const
Get the maximum of all qubit weights in a range.

inline bool **has_qubit**(const int v, const int q) const
Check if variable v is includes qubit q in its chain.

inline void **set_chain**(const int u, const vector<int> &incoming)
Assign a chain for variable u.

inline void **fix_chain**(const int u, const vector<int> &incoming)
Permanently assign a chain for variable u.

NOTE: This must be done before any chain is assigned to u.

inline bool **operator==**(const *embedding* &other) const
check if `this` and `other` have the same chains (up to qubit containment per chain; linking and parent information is not checked)

inline void **construct_chain**(const int u, const int q, const vector<vector<int>> &parents)
construct the chain for `u`, rooted at `q`, with a vector of parent info, where for each neibor `v` of `u`, following `q` -> `parents[v][q]` -> `parents[v][parents[v][q]]` ...

terminates in the chain for `v`

inline void **construct_chain_steiner**(const int u, const int q, const vector<vector<int>> &parents,
const vector<vector<*distance_t*>> &distances,
vector<vector<int>> &visited_list)
construct the chain for `u`, rooted at `q`.

for the first neighbor `v` of `u`, we follow the parents until we terminate in the chain for `v` `q` -> `parents[v][q]` -> .... adding all but the last node to the chain of `u`. for each subsequent neighbor `w`, we pick a nearest Steiner node, `qw`, from the current chain of `u`, and add the path starting at `qw`, similar to the above... `qw` -> `parents[w][qw]` -> ... this has an opportunity to make shorter chains than `construct_chain`

inline void **flip_back**(int u, const int target_chainsize)
distribute path segments to the neighboring chains path segments are the qubits that are ONLY used to join link_qubit[u][v] to link_qubit[u][u] and aren't used for any other variable

- if the target chainsize is zero, dump the entire segment into the neighbor
- if the target chainsize is k, stop when the neighbor's size reaches k

inline void **tear_out**(int u)
short tearout procedure blank out the chain, its linking qubits, and account for the qubits being freed

inline int **freeze_out**(int u)
undo-able tearout procedure.

similar to `tear_out(u)`, but can be undone with `thaw_back(u)`. note that this embedding type has a space for a single frozen chain, and `freeze_out(u)` overwrites the previously-frozen chain consequently, `freeze_out(u)` can be called an arbitrary (nonzero) number of times before `thaw_back(u)`, but `thaw_back(u)` MUST be preceeded by at least one `freeze_out(u)`. returns the size of the chain being frozen

inline void **thaw_back**(int u)
undo for the freeze_out procedure: replaces the chain previously frozen, and destroys the data in the frozen chain `thaw_back(u)` must be preceeded by at least one `freeze_out(u)` and the chain for `u` must currently be empty (accomplished either by `tear_out(u)` or `freeze_out(u)`)

inline void **steal_all**(int u)
grow the chain for `u`, stealing all available qubits from neighboring variables

inline int **statistics**(vector<int> &stats) const
compute statistics for this embedding and return 1 if no chains are overlapping when no chains are overlapping, populate `stats` with a chainlength histogram chains do overlap, populate `stats` with a qubit overfill histogram a histogram, in this case, is a vector of size (maximum attained value+1) where `stats[i]` is either the number of qubits contained in `i+2` chains or the number of chains with size `i`

inline bool **linked**() const
check if the embedding is fully linked that is, if each pair of adjacent variables is known to correspond to a pair of adjacent qubits

inline bool **linked**(int u) const
check if a single variable is linked with all adjacent variables.

inline void **print**() const
print out this embedding to a level of detail that is useful for debugging purposes TODO describe the output format.

inline void **long_diagnostic**(std::string current_state)
run a long diagnostic, and if debugging is enabled, record `current_state` so that the error message has a little more context.

if an error is found, throw a *CorruptEmbeddingException*

inline void **run_long_diagnostic**(std::string current_state) const
run a long diagnostic to verify the integrity of this datastructure.

the guts of this function are its documentation, because this function only exists for debugging purposes

template<class **fixed_handler**, class **domain_handler**, class **output_handler**>

class **embedding_problem** : public *find_embedding*::*embedding_problem_base*, public *fixed_handler*, public *domain_handler*, public *find_embedding*::*output_handler*<verbose>
#include <embedding_problem.hpp> A template to construct a complete embedding problem by combining `embedding_problem_base` with fixed/domain handlers.

class **embedding_problem_base**
#include <embedding_problem.hpp> Common form for all embedding problems.

Needs to be extended with a fixed handler and domain handler to be complete.

Subclassed by *find_embedding::embedding_problem< fixed_handler, domain_handler, output_handler >*

Public Functions

inline void **reset_mood**()
resets some internal, ephemeral, variables to a default state

inline void **populate_weight_table**(int max_weight)
precomputes a table of weights corresponding to various overlap values `c`, for `c` from 0 to `max_weight`, inclusive.

inline *distance_t* **weight**(unsigned int c) const
returns the precomputed weight associated with an overlap value of `c`

inline const vector<int> &**var_neighbors**(int u) const
a vector of neighbors for the variable `u`

inline const vector<int> &**var_neighbors**(int u, *shuffle_first*)
a vector of neighbors for the variable `u`, pre-shuffling them

inline const vector<int> &**var_neighbors**(int u, *rndswap_first*)
a vector of neighbors for the variable `u`, applying a random transposition before returning the reference

inline const vector<int> &**qubit_neighbors**(int q) const
a vector of neighbors for the qubit `q`

inline int **num_vars**() const
number of variables which are not fixed

inline int **num_qubits**() const
number of qubits which are not reserved

inline int **num_fixed**() const
number of fixed variables

inline int **num_reserved**() const
number of reserved qubits

inline int **randint**(int a, int b)
make a random integer between 0 and `m-1`

template<typename **A**, typename **B**>
inline void **shuffle**(*A* a, *B* b)
shuffle the data bracketed by iterators `a` and `b`

inline void **qubit_component**(int q0, vector<int> &component, vector<int> &visited)
compute the connected component of the subset `component` of qubits, containing `q0`, and using`visited` as an indicator for which qubits have been explored

inline const vector<int> &**var_order**(*VARORDER* order = VARORDER_SHUFFLE)
compute a variable ordering according to the `order` strategy

inline void **dfs_component**(int x, const vector<vector<int>> &neighbors, vector<int> &component, vector<int> &visited)
Perform a depth first search.

Public Members

optional_parameters &**params**
A mutable reference to the user specified parameters.

class **fixed_handler_hival**
#include <embedding_problem.hpp> This fixed handler is used when the fixed variables are processed before instantiation and relabeled such that variables v >= num_v are fixed and qubits q >= num_q are reserved.

class **fixed_handler_none**
#include <embedding_problem.hpp> This fixed handler is used when there are no fixed variables.

struct **frozen_chain**
#include <chain.hpp> This class stores chains for embeddings, and performs qubit-use accounting.

The `label` is the index number for the variable represented by this chain. The `links` member of a chain is an unordered map storing the linking information for this chain. The `data` member of a chain stores the connectivity information for the chain.

Links: If `u` and `v` are variables which are connected by an edge, the following must be true: either chain_u or chain_v is empty,

or

chain_u.links[v] is a key in chain_u.data, chain_v.links[u] is a key in chain_v.data, and (chain_u.links[v], chain_v.links[u]) are adjacent in the qubit graph

Moreover, (chain_u.links[u]) must exist if chain_u is not empty, and this is considered the root of the chain.

Data: The `data` member stores the connectivity information. More precisely, `data` is a mapping `qubit->(parent, refs)` where: `parent` is also contained in the chain `refs` is the total number of references to `qubit`, counting both parents and links the chain root is its own parent.

class **LocalInteraction**
#include <util.hpp> Interface for communication between the library and various bindings.

Any bindings of this library need to provide a concrete subclass.

Public Functions

inline void **displayOutput**(int loglevel, const string &msg) const
Print a message through the local output method.

inline void **displayError**(int loglevel, const string &msg) const
Print an error through the local output method.

inline int **cancelled**(const *clock*::time_point stoptime) const
Check if someone is trying to cancel the embedding process.

class **max_heap_tag**
#include <pairing_queue.hpp>

class **min_heap_tag**
#include <pairing_queue.hpp>

class **optional_parameters**
#include <util.hpp> Set of parameters used to control the embedding process.

Public Functions

inline **optional_parameters**(*optional_parameters* &p, map<int, vector<int>> fixed_chains, map<int, vector<int>> initial_chains, map<int, vector<int>> restrict_chains)
duplicate all parameters but chain hints, and seed a new rng.

this vaguely peculiar behavior is utilized to spawn parameters for component subproblems

Public Members

LocalInteractionPtr **localInteractionPtr**
actually not controlled by user, not initialized here, but initialized in Python, MATLAB, C wrappers level

double **timeout** = 1000
Number of seconds before the process unconditionally stops.

template<bool **verbose**>

class **output_handler**
#include <embedding_problem.hpp> Output handlers are used to control output.

We provide two handlers one which only reports all errors (and optimizes away all other output) and another which provides full output. When verbose is zero, we recommend the errors-only handler and otherwise, the full handler Here's the full output handler

Subclassed by *find_embedding::embedding_problem< fixed_handler, domain_handler, output_handler >*

Public Functions

template<typename ...**Args**>
inline void **error**(const char *format, *Args*... args) const
printf regardless of the verbosity level

template<typename ...**Args**>
inline void **major_info**(const char *format, *Args*... args) const
printf at the major_info verbosity level

template<typename ...**Args**>
inline void **minor_info**(const char *format, *Args*... args) const
print at the minor_info verbosity level

template<typename ...**Args**>
inline void **extra_info**(const char *format, *Args*... args) const
print at the extra_info verbosity level

template<typename ...**Args**>
inline void **debug**(const char*, *Args*...) const
print at the debug verbosity level (only works when CPPDEBUG is set)

template<typename **N**>

class **pairing_node** : public *N*
#include <pairing_queue.hpp>

Public Functions

inline *pairing_node*<*N*> ***merge_roots**(*pairing_node*<*N*> *other)*
the basic operation of the pairing queue put `this` and `other` into heap-order

template<typename **N**>

class **pairing_queue**
#include <pairing_queue.hpp>

class **parameter_processor**
#include <find_embedding.hpp>

template<typename **embedding_problem_t**>

class **pathfinder_base** : public *find_embedding*::*pathfinder_public_interface*
#include <pathfinder.hpp> Subclassed by *find_embedding::pathfinder_parallel< embedding_problem_t >*, *find_embedding::pathfinder_serial< embedding_problem_t >*

Public Functions

inline virtual void **set_initial_chains**(map<int, vector<int>> chains) override
setter for the initial_chains parameter

inline bool **check_improvement**(const *embedding_t* &emb)
nonzero return if this is an improvement on our previous best embedding

inline virtual const *chain* &**get_chain**(int u) const override
chain accessor

inline virtual int **heuristicEmbedding**() override
perform the heuristic embedding, returning 1 if an embedding was found and 0 otherwise

template<typename **embedding_problem_t**>

class **pathfinder_parallel** : public *find_embedding*::*pathfinder_base*<*embedding_problem_t*>
#include <pathfinder.hpp> A pathfinder where the Dijkstra-from-neighboring-chain passes are done serially.

Public Functions

inline virtual void **prepare_root_distances**(const *embedding_t* &emb, const int u) override
compute the distances from all neighbors of `u` to all qubits

class **pathfinder_public_interface**
#include <pathfinder.hpp> Subclassed by *find_embedding::pathfinder_base< embedding_problem_t >*

template<typename **embedding_problem_t**>

class **pathfinder_serial** : public *find_embedding*::*pathfinder_base*<*embedding_problem_t*>
#include <pathfinder.hpp> A pathfinder where the Dijkstra-from-neighboring-chain passes are done serially.

Public Functions

inline virtual void **prepare_root_distances**(const *embedding_t* &emb, const int u) override
compute the distances from all neighbors of `u` to all qubits

template<bool **parallel**, bool **fixed**, bool **restricted**, bool **verbose**>

class **pathfinder_type**
#include <find_embedding.hpp>

class **pathfinder_wrapper**
#include <find_embedding.hpp>

template<typename **P**, typename **heap_tag** = *min_heap_tag*>

class **priority_node**
#include <pairing_queue.hpp>

class **ProblemCancelledException** : public *minorminer*::*MinorMinerException*
#include <util.hpp>

struct **rndswap_first**
#include <embedding_problem.hpp>

struct **shuffle_first**
#include <embedding_problem.hpp>

class **TimeoutException** : public *minorminer*::*MinorMinerException*
#include <util.hpp>

Namespace graph

namespace **graph**

class **components**
#include <graph.hpp> Represents a graph as a series of connected components.

The input graph may consist of many components, they will be separated in the construction.

Public Functions

inline const std::vector<int> &**nodes**(int c) const
Get the set of nodes in a component.

inline size_t **size**() const
Get the number of connected components in the graph.

inline size_t **num_reserved**(int c) const
returns the number of reserved nodes in a component

inline size_t **size**(int c) const
Get the size (in nodes) of a component.

inline const *input_graph* &**component_graph**(int c) const
Get a const reference to the graph object of a component.

inline std::vector<std::vector<int>> **component_neighbors**(int c) const
Construct a neighborhood list for component c, with reserved nodes as sources.

template<typename **T**>
inline bool **into_component**(const int c, *T* &nodes_in, std::vector<int> &nodes_out) const
translate nodes from the input graph, to their labels in component c

template<typename **T**>
inline void **from_component**(const int c, *T* &nodes_in, std::vector<int> &nodes_out) const
translate nodes from labels in component c, back to their original input labels

class **input_graph**
#include <graph.hpp> Represents an undirected graph as a list of edges.

Provides methods to extract those edges into neighbor lists (with options to relabel and produce directed graphs).

As an input to the library this may be a disconnected graph, but when returned from components it is a connected sub graph.

Public Functions

inline **input_graph**()
Constructs an empty graph.

inline **input_graph**(int n_v, const std::vector<int> &aside, const std::vector<int> &bside)
Constructs a graph from the provided edges.

The ends of edge ii are aside[ii] and bside[ii].

Parameters

- **n_v** – Number of nodes in the graph.
- **aside** – List of nodes describing edges.
- **bside** – List of nodes describing edges.

inline void **clear**()
Remove all edges and nodes from a graph.

inline int **a**(const int i) const
Return the nodes on either end of edge `i`

inline int **b**(const int i) const
Return the nodes on either end of edge `i`

inline size_t **num_nodes**() const
Return the size of the graph in nodes.

inline size_t **num_edges**() const
Return the size of the graph in edges.

inline void **push_back**(int ai, int bi)
Add an edge to the graph.

template<typename **T1**, typename ...**Args**>
inline std::vector<std::vector<int>> **get_neighbors_sources**(const *T1* &sources, *Args*... args) const
produce a std::vector<std::vector<int>> of neigborhoods, with certain nodes marked as sources (inbound edges are omitted) sources is either a std::vector<int> (where non-sources x have sources[x] = 0), or another type for which we have a unaryint specialization optional arguments: relabel, mask (any type with a unaryint specialization) relabel is applied to the nodes as they are placed into the neighborhood list (and not used for checking sources / mask) mask is used to filter down to the induced graph on nodes x with mask[x] = 1

template<typename **T2**, typename ...**Args**>
inline std::vector<std::vector<int>> **get_neighbors_sinks**(const *T2* &sinks, *Args*... args) const
produce a std::vector<std::vector<int>> of neigborhoods, with certain nodes marked as sinks (outbound edges are omitted) sinks is either a std::vector<int> (where non-sinks x have sinks[x] = 0), or another type for which we have a unaryint specialization optional arguments: relabel, mask (any type with a unaryint specialization) relabel is applied to the nodes as they are placed into the neighborhood list (and not used for checking sinks / mask) mask is used to filter down to the induced graph on nodes x with mask[x] = 1

template<typename ...**Args**>
inline std::vector<std::vector<int>> **get_neighbors**(*Args*... args) const
produce a std::vector<std::vector<int>> of neigborhoods optional arguments: relabel, mask (any type with a unaryint specialization) relabel is applied to the nodes as they are placed into the neighborhood list (and not used for checking mask) mask is used to filter down to the induced graph on nodes x with mask[x] = 1

template<typename **T**>

class **unaryint**
#include <graph.hpp>

template<>

class **unaryint**<bool>
#include <graph.hpp>

template<>

class **unaryint**<int>
#include <graph.hpp>

template<> vector< int > >
#include <graph.hpp>

template<>

class **unaryint**<void*>
#include <graph.hpp> this one is a little weird construct a unaryint(nullptr) and get back the identity function f(x) -> x

Namespace minorminer

namespace **minorminer**

class **MinorMinerException** : public runtime_error
#include <errors.hpp> Subclassed by *find_embedding::BadInitializationException*, *find_embedding::CorruptEmbeddingException*, *find_embedding::CorruptParametersException*, *find_embedding::ProblemCancelledException*, *find_embedding::TimeoutException*

File list

File biclique_cache.hpp

namespace **busclique**

template<typename **topo_spec**>

class **biclique_cache**
#include <biclique_cache.hpp>

Public Functions

biclique_cache(const *biclique_cache*&) = delete

biclique_cache(*biclique_cache*&&) = delete

inline *yieldcache* **get**(size_t h, size_t w) const

inline **biclique_cache**(const *cell_cache*<*topo_spec*> &c, const *bundle_cache*<*topo_spec*> &b)

inline **~biclique_cache**()

inline std::pair<size_t, size_t> **score**(size_t y0, size_t y1, size_t x0, size_t x1) const

Public Members

const *cell_cache*<*topo_spec*> &**cells**

Private Functions

inline size_t **memrows**(size_t h) const

inline size_t **memcols**(size_t w) const

inline size_t **memsize**(size_t h, size_t w) const

inline size_t **memsize**() const

inline size_t **mem_addr**(size_t h, size_t w) const

inline void **make_access_table**()

inline void **compute_cache**(const *bundle_cache*<*topo_spec*> &bundles)

Private Members

size_t ***mem**

template<typename **topo_spec**>

class **biclique_yield_cache**
#include <biclique_cache.hpp>

Public Functions

biclique_yield_cache(const *biclique_yield_cache*&) = delete

biclique_yield_cache(*biclique_yield_cache*&&) = delete

inline **biclique_yield_cache**(const *cell_cache*<*topo_spec*> &c, const *bundle_cache*<*topo_spec*> &b, const *biclique_cache*<*topo_spec*> &bicliques)

inline *iterator* **begin**() const

inline *iterator* **end**() const

Public Members

const *cell_cache*<*topo_spec*> &**cells**

const *bundle_cache*<*topo_spec*> &**bundles**

Private Types

using **bound_t** = std::tuple<size_t, size_t, size_t, size_t>

Private Functions

inline void **compute_cache**(const *biclique_cache*<*topo_spec*> &bicliques)

Private Members

const size_t **rows**

const size_t **cols**

vector<vector<size_t>> **chainlength**

vector<vector<*bound_t*>> **biclique_bounds**

class **iterator**
#include <biclique_cache.hpp>

Public Functions

inline **iterator**(size_t _s0, size_t _s1, const size_t &r, const size_t &c, const vector<vector<size_t>> &cl, const vector<vector<*bound_t*>> &_bounds, const *bundle_cache*<*topo_spec*> &_bundles)

inline *iterator* **operator++**()

inline *iterator* **operator++**(int)

inline std::tuple<size_t, size_t, size_t, vector<vector<size_t>>> **operator***()

inline bool **operator==**(const *iterator* &rhs)

inline bool **operator!=**(const *iterator* &rhs)

Private Functions

inline void **adv**()

inline bool **inc**()

Private Members

size_t **s0**

size_t **s1**

const size_t &**rows**

const size_t &**cols**

const vector<vector<size_t>> &**chainlength**

const vector<vector<*bound_t*>> &**bounds**

const *bundle_cache*<*topo_spec*> &**bundles**

class **yieldcache**
#include <biclique_cache.hpp>

Public Functions

inline **yieldcache**(size_t r, size_t c, size_t *m)

inline size_t **get**(size_t y, size_t x, size_t u) const

inline void **set**(size_t y, size_t x, size_t u, size_t score)

Public Members

const size_t **rows**

const size_t **cols**

Private Members

size_t ***mem**

File bundle_cache.hpp

namespace **busclique**

template<typename **topo_spec**>

class **bundle_cache**
#include <bundle_cache.hpp>

Public Functions

inline **~bundle_cache**()

inline **bundle_cache**(const *cell_cache*<*topo_spec*> &c)

inline size_t **score**(size_t yc, size_t xc, size_t y0, size_t y1, size_t x0, size_t x1) const

inline void **inflate**(size_t yc, size_t xc, size_t y0, size_t y1, size_t x0, size_t x1,
vector<vector<size_t>> &emb) const

inline void **inflate**(size_t y0, size_t y1, size_t x0, size_t x1, vector<vector<size_t>> &emb) const

inline void **inflate**(size_t u, size_t y0, size_t y1, size_t x0, size_t x1, vector<vector<size_t>> &emb) const

inline size_t **length**(size_t yc, size_t xc, size_t y0, size_t y1, size_t x0, size_t x1) const

inline uint8_t **get_line_score**(size_t u, size_t w, size_t z0, size_t z1) const

Private Functions

bundle_cache(const *bundle_cache*&) = delete

bundle_cache(*bundle_cache*&&) = delete

inline uint8_t **get_line_mask**(size_t u, size_t w, size_t z0, size_t z1) const

inline void **compute_line_masks**()

Private Members

const *cell_cache*<*topo_spec*> &**cells**

const size_t **linestride**[2]

const size_t **orthstride**

uint8_t *__line_mask__

File util.hpp

Warning: doxygenfile: Found multiple matches for file "util.hpp

File cell_cache.hpp

namespace **busclique**

template<typename **topo_spec**>

class **cell_cache**

#include <cell_cache.hpp>

Public Functions

cell_cache(const *cell_cache*&) = delete

cell_cache(*cell_cache*&&) = delete

inline **~cell_cache**()

inline **cell_cache**(const *topo_spec* p, const vector<size_t> &nodes, const vector<pair<size_t, size_t>> &edges)

inline **cell_cache**(const *topo_spec* p, uint8_t *nm, uint8_t *em)

inline uint8_t **qmask**(size_t u, size_t w, size_t z) const

inline uint8_t **emask**(size_t u, size_t w, size_t z) const

Public Members

const *topo_spec* **topo**

Private Members

bool **borrow**

uint8_t ***nodemask**

uint8_t ***edgemask**

File chain.hpp

Defines

DIAGNOSE_CHAINS(other)

DIAGNOSE_CHAIN()

namespace **find_embedding**

class **chain**
#include <chain.hpp>

Public Functions

inline **chain**(vector<int> &w, int l)
construct this chain, linking it to the qubit_weight vector `w` (common to all chains in an embedding, typically) and setting its variable label `l`

inline *chain* &**operator=**(const vector<int> &c)
assign this to a vector of ints.

each incoming qubit will have itself as a parent.

inline *chain* &**operator=**(const *chain* &c)
assign this to another chain

inline size_t **size**() const
number of qubits in chain

inline size_t **count**(const int q) const
returns 0 if `q` is not contained in `this`, 1 otherwise

inline int **get_link**(const int x) const
get the qubit, in `this`, which links `this` to the chain of x (if x==label, interpret the linking qubit as the chain's root)

inline void **set_link**(const int x, const int q)
set the qubit, in `this`, which links `this` to the chain of x (if x==label, interpret the linking qubit as the chain's root)

inline int **drop_link**(const int x)
discard and return the linking qubit for `x`, or -1 if that link is not set

inline void **set_root**(const int q)
insert the qubit `q` into `this`, and set `q` to be the root (represented as the linking qubit for `label`)

inline void **clear**()
empty this data structure

inline void **add_leaf**(const int q, const int parent)
add the qubit `q` as a leaf, with `parent` as its parent

inline int **trim_branch**(int q)
try to delete the qubit `q` from this chain, and keep deleting until no more qubits are free to be deleted.

return the first ancestor which cannot be deleted

inline int **trim_leaf**(int q)
try to delete the qubit `q` from this chain.

if `q` cannot be deleted, return it; otherwise return its parent

inline int **parent**(const int q) const
the parent of `q` in this chain which might be `q` but otherwise cycles should be impossible

inline void **adopt**(const int p, const int q)
assign `p` to be the parent of `q`, on condition that both `p` and `q` are contained in `this`, `q` is its own parent, and `q` is not the root

inline int **refcount**(const int q) const
return the number of references that `this` makes to the qubit `q` where a "reference" is an occurrence of `q` as a parent or an occurrence of `q` as a linking qubit / root

inline size_t **freeze**(vector<*chain*> &others, *frozen_chain* &keep)
store this chain into a *frozen_chain*, unlink all chains from this, and *clear()*

inline void **thaw**(vector<*chain*> &others, *frozen_chain* &keep)
restore a *frozen_chain* into this, re-establishing links from other chains.

precondition: this is empty.

template<typename **embedding_problem_t**>
inline void **steal**(*chain* &other, *embedding_problem_t* &ep, int chainsize = 0)
assumes `this` and `other` have links for eachother's labels steals all qubits from `other` which are available to be taken by `this`; starting with the qubit links and updating qubit links after all

inline void **link_path**(*chain* &other, int q, const vector<int> &parents)
link this chain to another, following the path `q`, `parent[q]`, `parent[parent[q]]`, ...

from `this` to `other` and intermediate nodes (all but the last) into `this` (preconditions: `this` and `other` are not linked, `q` is contained in `this`, and the parent-path is eventually contained in `other`)

inline *iterator* **begin**() const
iterator pointing to the first qubit in this chain

inline *iterator* **end**() const
iterator pointing to the end of this chain

inline void **diagnostic**()
run the diagnostic, and if it fails, report the failure to the user and throw a *CorruptEmbeddingException*.

the `last_op` argument is used in the error message

inline int **run_diagnostic**() const
run the diagnostic and return a nonzero status `r` in case of failure if(`r`&1), then the parent of a qubit is not contained in this chain if(`r`&2), then there is a refcounting error in this chain

Public Members

const int **label**

Private Functions

inline const pair<int, int> &**fetch**(int q) const
const unsafe data accessor

inline pair<int, int> &**retrieve**(int q)
non-const unsafe data accessor

Private Members

vector<int> &**qubit_weight**

unordered_map<int, pair<int, int>> **data**

unordered_map<int, int> **links**

class **iterator**
#include <chain.hpp>

Public Functions

inline iterator (typename decltype(data)::const_iterator it)

inline *iterator* **operator++**()

inline bool **operator!=**(const *iterator* &other)

inline decltype(data) const ::key_type & operator* () const

Private Members

decltype(data) ::const_iterator it

struct **frozen_chain**

#include <*chain.hpp*> This class stores chains for embeddings, and performs qubit-use accounting.

The `label` is the index number for the variable represented by this chain. The `links` member of a chain is an unordered map storing the linking information for this chain. The `data` member of a chain stores the connectivity information for the chain.

Links: If `u` and `v` are variables which are connected by an edge, the following must be true: either chain_u or chain_v is empty,

or

chain_u.links[v] is a key in chain_u.data, chain_v.links[u] is a key in chain_v.data, and (chain_u.links[v], chain_v.links[u]) are adjacent in the qubit graph

Moreover, (chain_u.links[u]) must exist if chain_u is not empty, and this is considered the root of the chain.

Data: The `data` member stores the connectivity information. More precisely, `data` is a mapping `qubit->(parent, refs)` where: `parent` is also contained in the chain `refs` is the total number of references to `qubit`, counting both parents and links the chain root is its own parent.

Public Functions

inline void **clear**()

Public Members

unordered_map<int, pair<int, int>> **data**

unordered_map<int, int> **links**

File clique_cache.hpp

namespace **busclique**

Variables

const vector<vector<size_t>> **empty_emb**

template<typename **topo_spec**>

class **clique_cache**
#include <clique_cache.hpp>

Public Functions

clique_cache(const *clique_cache*&) = delete

clique_cache(*clique_cache*&&) = delete

inline **clique_cache**(const *cell_cache*<*topo_spec*> &c, const *bundle_cache*<*topo_spec*> &b, size_t w)

template<typename **C**>
inline **clique_cache**(const *cell_cache*<*topo_spec*> &c, const *bundle_cache*<*topo_spec*> &b, size_t w,
C &check)

inline ~**clique_cache**()

inline *maxcache* **get**(size_t i) const

inline void **print**()

inline bool **extract_solution**(vector<vector<size_t>> &emb) const

Private Functions

inline size_t **memrows**(size_t i) const

inline size_t **memcols**(size_t i) const

inline size_t **memsize**(size_t i) const

inline size_t **memsize**() const

template<typename **C**>

inline void **compute_cache**(*C* &check)

template<typename **T**, typename **C**, typename ...**Corners**>
inline void **extend_cache**(const *T* &prev, size_t h, size_t w, *C* &check, *Corners*... corners)

template<typename **T**, typename **C**, typename ...**Corners**>
inline void **extend_cache**(const *T* &prev, *maxcache* &next, size_t y0, size_t y1, size_t x0, size_t x1, *C* &check, *corner* c, *Corners*... corners)

template<typename **T**, typename **C**>
inline void **extend_cache**(const *T* &prev, *maxcache* &next, size_t y0, size_t y1, size_t x0, size_t x1, *C* &check, *corner* c)

inline *corner* **inflate_first_ell**(vector<vector<size_t>> &emb, size_t &y, size_t &x, size_t h, size_t w, *corner* c) const

Private Members

const *cell_cache*<*topo_spec*> &**cells**

const *bundle_cache*<*topo_spec*> &**bundles**

const size_t **width**

size_t ***mem**

Private Static Functions

static inline constexpr bool **nocheck**(size_t, size_t, size_t, size_t, size_t, size_t)

Friends

friend class clique_iterator< topo_spec >

template<typename **topo_spec**>

class **clique_iterator**
#include <clique_cache.hpp>

Public Functions

inline **clique_iterator**(const *cell_cache*<*topo_spec*> &c, const *clique_cache*<*topo_spec*> &q)

inline bool **next**(vector<vector<size_t>> &e)

Private Functions

inline bool **advance**()

inline bool **grow_stack**()

Private Members

const *cell_cache*<*topo_spec*> &**cells**

const *clique_cache*<*topo_spec*> &**cliq**

size_t **width**

vector<std::tuple<size_t, size_t, *corner*>> **basepoints**

vector<std::tuple<size_t, size_t, size_t, *corner*>> **stack**

vector<vector<size_t>> **emb**

template<typename **topo_spec**>

class **clique_yield_cache**
#include <clique_cache.hpp>

Public Functions

inline **clique_yield_cache**(const *cell_cache*<*zephyr_spec*> &cells)

inline **clique_yield_cache**(const *cell_cache*<*pegasus_spec*> &cells)

inline **clique_yield_cache**(const *cell_cache*<*chimera_spec*> &cells)

inline const vector<vector<vector<size_t>>> &**embeddings**()

Private Functions

inline size_t **emb_max_length**(const vector<vector<size_t>> &emb) const

inline void **process_cliques**(const *clique_cache*<*topo_spec*> &cliques)

inline void **compute_cache_width_1**(const *cell_cache*<*topo_spec*> &cells, const *bundle_cache*<*topo_spec*> &bundles)

inline void **compute_cache_width_gt_1**(const *cell_cache*<*pegasus_spec*> &cells, const *bundle_cache*<*pegasus_spec*> &bundles)

inline void **compute_cache_width_gt_1**(const *cell_cache*<*chimera_spec*> &cells, const *bundle_cache*<*chimera_spec*> &bundles)

inline void **compute_cache_width_gt_1**(const *cell_cache*<*zephyr_spec*> &cells, const *bundle_cache*<*zephyr_spec*> &bundles)

inline void **compute_cache**(const *cell_cache*<*zephyr_spec*> &cells)

inline void **compute_cache**(const *cell_cache*<*chimera_spec*> &cells)

inline void **compute_cache**(const *cell_cache*<*pegasus_spec*> &cells)

inline void **get_length_range**(const *bundle_cache*<*pegasus_spec*> &bundles, size_t width, size_t &min_length, size_t &max_length)

inline void **get_length_range**(const *bundle_cache*<*chimera_spec*>&, size_t width, size_t &min_length, size_t &max_length)

inline void **get_length_range**(const *bundle_cache*<*zephyr_spec*>&, size_t width, size_t &min_length, size_t &max_length)

Private Members

const size_t **length_bound**

vector<size_t> **clique_yield**

vector<vector<vector<size_t>>> **best_embeddings**

class **maxcache**

#include <clique_cache.hpp>

Public Functions

inline **maxcache**(size_t r, size_t c, size_t *m)

inline void **setmax**(size_t y, size_t x, size_t s, *corner* c)

inline size_t **score**(size_t y, size_t x) const

inline *corner* **corners**(size_t y, size_t x) const

Public Members

const size_t **rows**

const size_t **cols**

Private Members

size_t ***mem**

class **zerocache**
#include <clique_cache.hpp>

Public Functions

inline constexpr size_t **score**(size_t, size_t) const

File debug.hpp

Defines

minorminer_assert(X)

File embedding.hpp

Defines

DIAGNOSE_EMB(X)

namespace **find_embedding**

template<typename **embedding_problem_t**>

class **embedding**

#include <embedding.hpp> This class is how we represent and manipulate embedding objects, using as much encapsulation as possible.

We provide methods to view and modify chains.

Public Functions

inline **embedding**(*embedding_problem_t* &e_p)
constructor for an empty embedding

inline **embedding**(*embedding_problem_t* &e_p, map<int, vector<int>> &fixed_chains, map<int, vector<int>> &initial_chains)
constructor for an initial embedding: accepts fixed and initial chains, populates the embedding based on them, and attempts to link adjacent chains together.

inline *embedding*<*embedding_problem_t*> &**operator=**(const *embedding*<*embedding_problem_t*> &other)
copy the data from `other.var_embedding` into `this.var_embedding`

inline const *chain* &**get_chain**(int v) const
Get the variables in a chain.

inline unsigned int **chainsize**(int v) const
Get the size of a chain.

inline int **weight**(int q) const
Get the weight of a qubit.

inline int **max_weight**() const
Get the maximum of all qubit weights.

inline int **max_weight**(const int start, const int stop) const
Get the maximum of all qubit weights in a range.

inline bool **has_qubit**(const int v, const int q) const
Check if variable v is includes qubit q in its chain.

inline void **set_chain**(const int u, const vector<int> &incoming)
Assign a chain for variable u.

inline void **fix_chain**(const int u, const vector<int> &incoming)
Permanently assign a chain for variable u.

NOTE: This must be done before any chain is assigned to u.

inline bool **operator==**(const *embedding* &other) const
check if `this` and `other` have the same chains (up to qubit containment per chain; linking and parent information is not checked)

inline void **construct_chain**(const int u, const int q, const vector<vector<int>> &parents)
construct the chain for `u`, rooted at `q`, with a vector of parent info, where for each neibor `v` of `u`, following `q` -> `parents[v][q]` -> `parents[v][parents[v][q]]` ...

terminates in the chain for `v`

inline void **construct_chain_steiner**(const int u, const int q, const vector<vector<int>> &parents, const vector<vector<*distance_t*>> &distances, vector<vector<int>> &visited_list)
construct the chain for u, rooted at q.

for the first neighbor `v` of `u`, we follow the parents until we terminate in the chain for `v q -> parents[v][q] -> ....` adding all but the last node to the chain of `u`. for each subsequent neighbor `w`, we pick a nearest Steiner node, `qw`, from the current chain of `u`, and add the path starting at `qw`, similar to the above... `qw -> parents[w][qw] -> ...` this has an opportunity to make shorter chains than `construct_chain`

inline void **flip_back**(int u, const int target_chainsize)
distribute path segments to the neighboring chains path segments are the qubits that are ONLY used to join link_qubit[u][v] to link_qubit[u][u] and aren't used for any other variable

- if the target chainsize is zero, dump the entire segment into the neighbor
- if the target chainsize is k, stop when the neighbor's size reaches k

inline void **tear_out**(int u)
short tearout procedure blank out the chain, its linking qubits, and account for the qubits being freed

inline int **freeze_out**(int u)
undo-able tearout procedure.

similar to `tear_out(u)`, but can be undone with `thaw_back(u)`. note that this embedding type has a space for a single frozen chain, and `freeze_out(u)` overwrites the previously-frozen chain consequently, `freeze_out(u)` can be called an arbitrary (nonzero) number of times before `thaw_back(u)`, but `thaw_back(u)` MUST be preceeded by at least one `freeze_out(u)`. returns the size of the chain being frozen

inline void **thaw_back**(int u)
undo for the freeze_out procedure: replaces the chain previously frozen, and destroys the data in the frozen chain `thaw_back(u)` must be preceeded by at least one `freeze_out(u)` and the chain for `u` must currently be empty (accomplished either by `tear_out(u)` or `freeze_out(u)`)

inline void **steal_all**(int u)
grow the chain for `u`, stealing all available qubits from neighboring variables

inline int **statistics**(vector<int> &stats) const
compute statistics for this embedding and return 1 if no chains are overlapping when no chains are overlapping, populate `stats` with a chainlength histogram chains do overlap, populate `stats` with a qubit overfill histogram a histogram, in this case, is a vector of size (maximum attained value+1) where `stats[i]` is either the number of qubits contained in `i+2` chains or the number of chains with size `i`

inline bool **linked**() const
check if the embedding is fully linked that is, if each pair of adjacent variables is known to correspond to a pair of adjacent qubits

inline bool **linked**(int u) const
check if a single variable is linked with all adjacent variables.

inline void **print**() const
print out this embedding to a level of detail that is useful for debugging purposes TODO describe the output format.

inline void **long_diagnostic**(std::string current_state)
run a long diagnostic, and if debugging is enabled, record `current_state` so that the error message has a little more context.

if an error is found, throw a *CorruptEmbeddingException*

inline void **run_long_diagnostic**(std::string current_state) const
run a long diagnostic to verify the integrity of this datastructure.

the guts of this function are its documentation, because this function only exists for debugging purposes

Private Functions

inline bool **linkup**(int u, int v)
This method attempts to find the linking qubits for a pair of adjacent variables, and returns true/false on success/failure in finding that pair.

Private Members

embedding_problem_t &**ep**

int **num_qubits**

int **num_reserved**

int **num_vars**

int **num_fixed**

vector<int> **qub_weight**
weights, that is, the number of non-fixed chains that use each qubit this is used in pathfinder clases to determine non-overlapped, or or least-overlapped paths through the qubit graph

vector<*chain*> **var_embedding**
this is where we store chains see chain.hpp for how

frozen_chain **frozen**

File embedding_problem.hpp

namespace **find_embedding**

Enums

enum **VARORDER**
Values:

enumerator **VARORDER_SHUFFLE**

enumerator **VARORDER_DFS**

enumerator **VARORDER_BFS**

enumerator **VARORDER_PFS**

enumerator **VARORDER_RPFS**

enumerator **VARORDER_KEEP**

class **domain_handler_masked**
#include <embedding_problem.hpp> this domain handler stores masks for each variable so that prepare_visited and prepare_distances are barely more expensive than a memcopy

Public Functions

inline **domain_handler_masked**(*optional_parameters* &p, int n_v, int n_f, int n_q, int n_r)

inline virtual **~domain_handler_masked**()

inline void **prepare_visited**(vector<int> &visited, const int u, const int v)

inline void **prepare_distances**(vector<*distance_t*> &distance, const int u, const *distance_t* &mask_d)

inline void **prepare_distances**(vector<*distance_t*> &distance, const int u, const *distance_t* &mask_d, const int start, const int stop)

inline bool **accepts_qubit**(const int u, const int q)

Private Members

optional_parameters &**params**

vector<vector<int>> **masks**

class **domain_handler_universe**

#include <embedding_problem.hpp> this is the trivial domain handler, where every variable is allowed to use every qubit

Public Functions

inline **domain_handler_universe**(*optional_parameters*&, int, int, int, int)

inline virtual **~domain_handler_universe**()

Public Static Functions

static inline void **prepare_visited**(vector<int> &visited, int, int)

static inline void **prepare_distances**(vector<*distance_t*> &distance, const int, const *distance_t*&)

static inline void **prepare_distances**(vector<*distance_t*> &distance, const int, const *distance_t*&, const int start, const int stop)

static inline bool **accepts_qubit**(int, int)

template<class **fixed_handler**, class **domain_handler**, class **output_handler**>

class **embedding_problem** : public *find_embedding*::*embedding_problem_base*, public *fixed_handler*, public *domain_handler*, public *find_embedding*::*output_handler*<verbose>

#include <embedding_problem.hpp> A template to construct a complete embedding problem by combining `embedding_problem_base` with fixed/domain handlers.

Public Functions

inline **embedding_problem**(*optional_parameters* &p, int n_v, int n_f, int n_q, int n_r, vector<vector<int>> &v_n, vector<vector<int>> &q_n)

inline virtual **~embedding_problem**()

Private Types

using **ep_t** = *embedding_problem_base*

using **fh_t** = *fixed_handler*

using **dh_t** = *domain_handler*

using **oh_t** = *output_handler*

class **embedding_problem_base**

#include <embedding_problem.hpp> Common form for all embedding problems.

Needs to be extended with a fixed handler and domain handler to be complete.

Subclassed by *find_embedding::embedding_problem< fixed_handler, domain_handler, output_handler >*

Public Functions

inline **embedding_problem_base**(*optional_parameters* &p_, int n_v, int n_f, int n_q, int n_r, vector<vector<int>> &v_n, vector<vector<int>> &q_n)

inline virtual **~embedding_problem_base**()

inline void **reset_mood**()
resets some internal, ephemeral, variables to a default state

inline void **populate_weight_table**(int max_weight)
precomputes a table of weights corresponding to various overlap values `c`, for `c` from 0 to `max_weight`, inclusive.

inline *distance_t* **weight**(unsigned int c) const
returns the precomputed weight associated with an overlap value of `c`

inline const vector<int> &**var_neighbors**(int u) const
a vector of neighbors for the variable `u`

inline const vector<int> &**var_neighbors**(int u, *shuffle_first*)
a vector of neighbors for the variable `u`, pre-shuffling them

inline const vector<int> &**var_neighbors**(int u, *rndswap_first*)
a vector of neighbors for the variable `u`, applying a random transposition before returning the reference

inline const vector<int> &**qubit_neighbors**(int q) const
a vector of neighbors for the qubit `q`

inline int **num_vars**() const
number of variables which are not fixed

inline int **num_qubits**() const
number of qubits which are not reserved

inline int **num_fixed**() const
number of fixed variables

inline int **num_reserved**() const
number of reserved qubits

inline int **randint**(int a, int b)
make a random integer between 0 and `m-1`

template<typename **A**, typename **B**>
inline void **shuffle**(*A* a, *B* b)
shuffle the data bracketed by iterators `a` and `b`

inline void **qubit_component**(int q0, vector<int> &component, vector<int> &visited)
compute the connected component of the subset `component` of qubits, containing `q0`, and using`visited` as an indicator for which qubits have been explored

inline const vector<int> &**var_order**(*VARORDER* order = VARORDER_SHUFFLE)
compute a variable ordering according to the `order` strategy

inline void **dfs_component**(int x, const vector<vector<int>> &neighbors, vector<int> &component, vector<int> &visited)
Perform a depth first search.

Public Members

optional_parameters &**params**
A mutable reference to the user specified parameters.

double **max_beta**

double **round_beta**

double **bound_beta**

distance_t **weight_table**[64]

int **initialized**

int **embedded**

int **desperate**

int **target_chainsize**

int **improved**

int **weight_bound**

Protected Attributes

int **num_v**

int **num_f**

int **num_q**

int **num_r**

vector<vector<int>> &**qubit_nbrs**
Mutable references to qubit numbers and variable numbers.

vector<vector<int>> &**var_nbrs**

uniform_int_distribution **rand**
distribution over [0, 0xffffffff]

vector<int> **var_order_space**

vector<int> **var_order_visited**

vector<int> **var_order_shuffle**

unsigned int **exponent_margin**

Private Functions

inline size_t **compute_margin**()
computes an upper bound on the distances computed during tearout & replace

template<typename **queue_t**>
inline void **pfs_component**(int x, const vector<vector<int>> &neighbors, vector<int> &component, vector<int> &visited, vector<int> shuffled)
Perform a priority first search (priority = #of visited neighbors)

inline void **bfs_component**(int x, const vector<vector<int>> &neighbors, vector<int> &component, vector<int> &visited, vector<int> &shuffled)
Perform a breadth first search, shuffling level sets.

class **fixed_handler_hival**
#include <embedding_problem.hpp> This fixed handler is used when the fixed variables are processed before instantiation and relabeled such that variables v >= num_v are fixed and qubits q >= num_q are reserved.

Public Functions

inline **fixed_handler_hival**(*optional_parameters*&, int n_v, int, int n_q, int)

inline virtual ~**fixed_handler_hival**()

inline bool **fixed**(const int u)

inline bool **reserved**(const int q)

Private Members

int **num_v**

int **num_q**

class **fixed_handler_none**
#include <embedding_problem.hpp> This fixed handler is used when there are no fixed variables.

Public Functions

inline **fixed_handler_none**(*optional_parameters*&, int, int, int, int)

inline virtual **~fixed_handler_none**()

Public Static Functions

static inline bool **fixed**(int)

static inline bool **reserved**(int)

template<bool **verbose**>

class **output_handler**
#include <embedding_problem.hpp> Output handlers are used to control output.

We provide two handlers one which only reports all errors (and optimizes away all other output) and another which provides full output. When verbose is zero, we recommend the errors-only handler and otherwise, the full handler Here's the full output handler

Subclassed by *find_embedding::embedding_problem< fixed_handler, domain_handler, output_handler >*

Public Functions

inline **output_handler**(*optional_parameters* &p)

template<typename ...**Args**>
inline void **error**(const char *format, *Args*... args) const
printf regardless of the verbosity level

template<typename ...**Args**>
inline void **major_info**(const char *format, *Args*... args) const
printf at the major_info verbosity level

template<typename ...**Args**>
inline void **minor_info**(const char *format, *Args*... args) const
print at the minor_info verbosity level

template<typename ...**Args**>
inline void **extra_info**(const char *format, *Args*... args) const
print at the extra_info verbosity level

template<typename ...**Args**>
inline void **debug**(const char*, *Args*...) const
print at the debug verbosity level (only works when CPPDEBUG is set)

Private Members

optional_parameters &**params**

struct **rndswap_first**
#include <embedding_problem.hpp>

struct **shuffle_first**
#include <embedding_problem.hpp>

File errors.hpp

namespace **minorminer**

class **MinorMinerException** : public runtime_error
#include <errors.hpp> Subclassed by *find_embedding::BadInitializationException*, *find_embedding::CorruptEmbeddingException*, *find_embedding::CorruptParametersException*, *find_embedding::ProblemCancelledException*, *find_embedding::TimeoutException*

Public Functions

inline **MinorMinerException**(const std::string &m = "find embedding exception")

File fastrng.hpp

namespace **fastrng**

class **fastrng**
#include <fastrng.hpp>

Public Types

typedef uint64_t **result_type**

Public Functions

inline **fastrng**()

inline **fastrng**(uint64_t x)

inline void **seed**(uint32_t x)

inline void **seed**(uint64_t x)

inline uint64_t **operator()**()

inline void **discard**(int n)

Public Static Functions

static inline uint64_t **amplify_seed**(uint32_t x)

static inline constexpr uint64_t **min**()

static inline constexpr uint64_t **max**()

Private Members

uint64_t **S0**

uint64_t **S1**

Private Static Functions

static inline uint64_t **splitmix64**(uint64_t &x)

static inline uint32_t **splitmix32**(uint32_t &x)

File find_biclique.hpp

namespace **busclique**

Typedefs

using **biclique_result_cache** = std::unordered_map<pair<size_t, size_t>, value_t, *craphash*>

Functions

template<typename **topo_spec**>
void **best_bicliques**(const *topo_spec* &topo, const vector<size_t> &nodes, const vector<pair<size_t, size_t>> &edges, vector<pair<pair<size_t, size_t>, vector<vector<size_t>>>> &embs)

template<typename **topo_spec**>
void **best_bicliques**(*topo_cache*<*topo_spec*> &topology, vector<pair<pair<size_t, size_t>, vector<vector<size_t>>>> &embs)

class **craphash**
#include <find_biclique.hpp>

Public Functions

inline size_t **operator()**(const pair<size_t, size_t> x) const

File find_clique.hpp

namespace **busclique**

Functions

template<typename **T**>
size_t **get_maxlen**(vector<*T*> &emb, size_t size)

template<typename **topo_spec**>
bool **find_clique_nice**(const *cell_cache*<*topo_spec*>&, size_t size, vector<vector<size_t>> &emb, size_t &min_width, size_t &max_width, size_t &max_length)

template<>
bool **find_clique_nice**(const *cell_cache*<*chimera_spec*> &cells, size_t size, vector<vector<size_t>> &emb, size_t&, size_t&, size_t &max_length)

template<typename **clique_cache_t**>
size_t **check_sol**(const *clique_cache_t* &rects, vector<vector<size_t>> &emb, size_t size)

template<>

bool **find_clique_nice**(const *cell_cache*<*zephyr_spec*> &cells, size_t size, vector<vector<size_t>> &emb, size_t&, size_t&, size_t &max_length)

template<>
bool **find_clique_nice**(const *cell_cache*<*pegasus_spec*> &cells, size_t size, vector<vector<size_t>> &emb, size_t&, size_t&, size_t &max_length)

template<typename **topo_spec**>
bool **find_clique**(const *topo_spec* &topo, const vector<size_t> &nodes, const vector<pair<size_t, size_t>> &edges, size_t size, vector<vector<size_t>> &emb)

template<typename **topo_spec**>
bool **find_clique**(*topo_cache*<*topo_spec*> &topology, size_t size, vector<vector<size_t>> &emb)

template<typename **topo_spec**>
bool **find_clique_nice**(const *topo_spec* &topo, const vector<size_t> &nodes, const vector<pair<size_t, size_t>> &edges, size_t size, vector<vector<size_t>> &emb)

template<typename **topo_spec**>
void **short_clique**(const *topo_spec*&, const vector<size_t> &nodes, const vector<pair<size_t, size_t>> &edges, vector<vector<size_t>> &emb)

template<typename **topo_spec**>
void **best_cliques**(*topo_cache*<*topo_spec*> &topology, vector<vector<vector<size_t>>> &embs, vector<vector<size_t>> &emb_1)

File util.hpp

Warning: doxygenfile: Found multiple matches for file "util.hpp

File find_embedding.hpp

namespace **find_embedding**

Functions

int **findEmbedding**(*graph*::*input_graph* &var_g, *graph*::*input_graph* &qubit_g, *optional_parameters* &params, vector<vector<int>> &chains)

The main entry function of this library.

This method primarily dispatches the proper implementation of the algorithm where some parameters/behaviours have been fixed at compile time.

In terms of dispatch, there are three dynamically-selected classes which are combined, each according to a specific optional parameter.

- a domain_handler, described in embedding_problem.hpp, manages constraints of the form "variable a's chain must be a subset of…"
- a fixed_handler, described in embedding_problem.hpp, manages contstraints of the form "variable a's chain must be exactly…"
- a pathfinder, described in pathfinder.hpp, which come in two flavors, serial and parallel The optional parameters themselves can be found in util.hpp. Respectively, the controlling options for the above are restrict_chains, fixed_chains, and threads.

class **parameter_processor**
#include <find_embedding.hpp>

Public Functions

inline **parameter_processor**(*graph*::*input_graph* &var_g, *graph*::*input_graph* &qubit_g, *optional_parameters* &params_)

inline map<int, vector<int>> **input_chains**(map<int, vector<int>> &m)

inline vector<int> **input_vars**(vector<int> &V)

Public Members

unsigned int **num_vars**

unsigned int **num_qubits**

vector<int> **qub_reserved_unscrewed**

vector<int> **var_fixed_unscrewed**

unsigned int **num_reserved**

graph::*components* **qub_components**

unsigned int **problem_qubits**

unsigned int **problem_reserved**

unsigned int **num_fixed**

vector<int> **unscrew_vars**

vector<int> **screw_vars**

optional_parameters **params**

vector<vector<int>> **var_nbrs**

vector<vector<int>> **qubit_nbrs**

Private Functions

inline unsigned int **_reserved**(*optional_parameters* &params_)

inline vector<int> **_filter_fixed_vars**()

inline vector<int> **_inverse_permutation**(vector<int> &f)

template<bool **parallel**, bool **fixed**, bool **restricted**, bool **verbose**>

class **pathfinder_type**
#include <find_embedding.hpp>

Public Types

typedef std::conditional<*fixed*, *fixed_handler_hival*, *fixed_handler_none*>::type **fixed_handler_t**

typedef std::conditional<*restricted*, *domain_handler_masked*, *domain_handler_universe*>::type **domain_handler_t**

typedef *output_handler*<*verbose*> **output_handler_t**

typedef *embedding_problem*<*fixed_handler_t*, *domain_handler_t*, *output_handler_t*> **embedding_problem_t**

typedef std::conditional<*parallel*, *pathfinder_parallel*<*embedding_problem_t*>, *pathfinder_serial*<*embedding_problem_t*>>::type **pathfinder_t**

class **pathfinder_wrapper**
#include <find_embedding.hpp>

Public Functions

inline **pathfinder_wrapper**(*graph*::*input_graph* &var_g, *graph*::*input_graph* &qubit_g, *optional_parameters* &params_)

inline **~pathfinder_wrapper**()

inline void **get_chain**(int u, vector<int> &output) const

inline int **heuristicEmbedding**()

inline int **num_vars**()

inline void **set_initial_chains**(map<int, vector<int>> &init)

inline void **quickPass**(vector<int> &varorder, int chainlength_bound, int overlap_bound, bool local_search, bool clear_first, double round_beta)

inline void **quickPass**(*VARORDER* varorder, int chainlength_bound, int overlap_bound, bool local_search, bool clear_first, double round_beta)

Private Functions

template<bool **parallel**, bool **fixed**, bool **restricted**, bool **verbose**, typename ...**Args**>
inline std::unique_ptr<*pathfinder_public_interface*> **_pf_parse4**(*Args*&&... args)

template<bool **parallel**, bool **fixed**, bool **restricted**, typename ...**Args**>
inline std::unique_ptr<*pathfinder_public_interface*> **_pf_parse3**(*Args*&&... args)

template<bool **parallel**, bool **fixed**, typename ...**Args**>
inline std::unique_ptr<*pathfinder_public_interface*> **_pf_parse2**(*Args*&&... args)

template<bool **parallel**, typename ...**Args**>
inline std::unique_ptr<*pathfinder_public_interface*> **_pf_parse1**(*Args*&&... args)

template<typename ...**Args**>
inline std::unique_ptr<*pathfinder_public_interface*> **_pf_parse**(*Args*&&... args)

Private Members

parameter_processor **pp**

std::unique_ptr<*pathfinder_public_interface*> **pf**

File graph.hpp

template<>

class *graph*::**unaryint**<std::vector<int>>
#include <graph.hpp>

Public Functions

inline **unaryint**(const std::vector<int> m)

inline int **operator()**(int i) const

Private Members

const std::vector<int> **b**

namespace **graph**

class **components**

#include <graph.hpp> Represents a graph as a series of connected components.

The input graph may consist of many components, they will be separated in the construction.

Public Functions

template<typename **T**>
inline **components**(const *input_graph* &g, const *unaryint*<*T*> &reserve)

inline **components**(const *input_graph* &g)

inline **components**(const *input_graph* &g, const std::vector<int> reserve)

inline const std::vector<int> &**nodes**(int c) const
Get the set of nodes in a component.

inline size_t **size**() const
Get the number of connected components in the graph.

inline size_t **num_reserved**(int c) const
returns the number of reserved nodes in a component

inline size_t **size**(int c) const
Get the size (in nodes) of a component.

inline const *input_graph* &**component_graph**(int c) const
Get a const reference to the graph object of a component.

inline std::vector<std::vector<int>> **component_neighbors**(int c) const
Construct a neighborhood list for component c, with reserved nodes as sources.

template<typename **T**>
inline bool **into_component**(const int c, *T* &nodes_in, std::vector<int> &nodes_out) const
translate nodes from the input graph, to their labels in component c

template<typename **T**>
inline void **from_component**(const int c, *T* &nodes_in, std::vector<int> &nodes_out) const
translate nodes from labels in component c, back to their original input labels

Private Functions

inline int **__init_find**(int x)

inline void **__init_union**(int x, int y)

Private Members

std::vector<int> **index**

std::vector<int> **label**

std::vector<int> **_num_reserved**

std::vector<std::vector<int>> **component**

std::vector<*input_graph*> **component_g**

class **input_graph**

#include <graph.hpp> Represents an undirected graph as a list of edges.

Provides methods to extract those edges into neighbor lists (with options to relabel and produce directed graphs).

As an input to the library this may be a disconnected graph, but when returned from components it is a connected sub graph.

Public Functions

inline **input_graph**()

Constructs an empty graph.

inline **input_graph**(int n_v, const std::vector<int> &aside, const std::vector<int> &bside)

Constructs a graph from the provided edges.

The ends of edge ii are aside[ii] and bside[ii].

Parameters

- **n_v** – Number of nodes in the graph.
- **aside** – List of nodes describing edges.
- **bside** – List of nodes describing edges.

inline void **clear**()

Remove all edges and nodes from a graph.

inline int **a**(const int i) const

Return the nodes on either end of edge `i`

inline int **b**(const int i) const

Return the nodes on either end of edge `i`

inline size_t **num_nodes**() const

Return the size of the graph in nodes.

inline size_t **num_edges**() const

Return the size of the graph in edges.

inline void **push_back**(int ai, int bi)

Add an edge to the graph.

template<typename **T1**, typename ...**Args**>
inline std::vector<std::vector<int>> **get_neighbors_sources**(const *T1* &sources, *Args*... args) const
produce a std::vector<std::vector<int>> of neigborhoods, with certain nodes marked as sources (inbound edges are omitted) sources is either a std::vector<int> (where non-sources x have sources[x] = 0), or another type for which we have a unaryint specialization optional arguments: relabel, mask (any type with a unaryint specialization) relabel is applied to the nodes as they are placed into the neighborhood list (and not used for checking sources / mask) mask is used to filter down to the induced graph on nodes x with mask[x] = 1

template<typename **T2**, typename ...**Args**>
inline std::vector<std::vector<int>> **get_neighbors_sinks**(const *T2* &sinks, *Args*... args) const
produce a std::vector<std::vector<int>> of neigborhoods, with certain nodes marked as sinks (outbound edges are omitted) sinks is either a std::vector<int> (where non-sinks x have sinks[x] = 0), or another type for which we have a unaryint specialization optional arguments: relabel, mask (any type with a unaryint specialization) relabel is applied to the nodes as they are placed into the neighborhood list (and not used for checking sinks / mask) mask is used to filter down to the induced graph on nodes x with mask[x] = 1

template<typename ...**Args**>
inline std::vector<std::vector<int>> **get_neighbors**(*Args*... args) const
produce a std::vector<std::vector<int>> of neigborhoods optional arguments: relabel, mask (any type with a unaryint specialization) relabel is applied to the nodes as they are placed into the neighborhood list (and not used for checking mask) mask is used to filter down to the induced graph on nodes x with mask[x] = 1

Private Functions

inline std::vector<std::vector<int>> **_to_vectorhoods**(std::vector<std::set<int>> &_nbrs) const
this method converts a std::vector of sets into a std::vector of sets, ensuring that element i is not contained in nbrs[i].

this method is called by methods which produce neighbor sets (killing parallel/overrepresented edges), in order to kill self-loops and also store each neighborhood in a contiguous memory segment.

template<typename **T1**, typename **T2**, typename **T3**, typename **T4**>
inline std::vector<std::vector<int>> **__get_neighbors**(const *unaryint*<*T1*> &sources, const *unaryint*<*T2*> &sinks, const *unaryint*<*T3*> &relabel, const *unaryint*<*T4*> &mask) const
produce the node->nodelist mapping for our graph, where certain nodes are marked as sources (no incoming edges), relabeling all nodes along the way, and filtering according to a mask.

note that the mask itself is assumed to be a union of components only one side of each edge is checked

template<typename **T1**, typename **T2**, typename **T3** = void*, typename **T4** = bool>
inline std::vector<std::vector<int>> **_get_neighbors**(const *T1* &sources, const *T2* &sinks, const *T3* &relabel = nullptr, const *T4* &mask = true) const
smash the types through unaryint

Private Members

std::vector<int> **edges_aside**

std::vector<int> **edges_bside**

size_t **_num_nodes**

template<typename **T**>

class **unaryint**
#include <graph.hpp>

template<>

class **unaryint**<bool>
#include <graph.hpp>

Public Functions

inline **unaryint**(const bool x)

inline int **operator()**(int) const

Private Members

const bool **b**

template<>

class **unaryint**<int>
#include <graph.hpp>

Public Functions

inline **unaryint**(int m)

inline int **operator()**(int i) const

Private Members

const int **b**

template<> vector< int > >
#include <graph.hpp>

Public Functions

inline **unaryint**(const std::vector<int> m)

inline int **operator()**(int i) const

Private Members

const std::vector<int> **b**

template<>

class **unaryint**<void*>
#include <graph.hpp> this one is a little weird construct a unaryint(nullptr) and get back the identity function f(x) -> x

Public Functions

inline **unaryint**(void*const&)

inline int **operator()**(int i) const

File pairing_queue.hpp

namespace **find_embedding**

class **max_heap_tag**
#include <pairing_queue.hpp>

class **min_heap_tag**
#include <pairing_queue.hpp>

template<typename **N**>

class **pairing_node** : public *N*
#include <pairing_queue.hpp>

Public Functions

inline **pairing_node**()

template<class ...**Args**>
inline **pairing_node**(*Args*... args)

inline *pairing_node*<*N*> ***merge_roots**(*pairing_node*<*N*> *other)
the basic operation of the pairing queue put `this` and `other` into heap-order

template<class ...**Args**>
inline void **refresh**(*Args*... args)

inline *pairing_node*<*N*> ***next_root**()

inline *pairing_node*<*N*> ***merge_pairs**()

Private Functions

inline *pairing_node*<*N*> ***merge_roots_unsafe**(*pairing_node*<*N*> *other)
the basic operation of the pairing queue put `this` and `other` into heap-order

inline *pairing_node*<*N*> ***merge_roots_unchecked**(*pairing_node* *other)
merge_roots, assuming `other` is not null and that `val` < `other->val`.

may invalidate the internal data structure (see source for details)

Private Members

pairing_node ***next**

pairing_node ***desc**

template<typename **N**>

class **pairing_queue**
#include <pairing_queue.hpp>

Public Functions

inline **pairing_queue**(int n)

inline **pairing_queue**(*pairing_queue* &&other)

inline **~pairing_queue**()

inline void **reset**()

inline bool **empty**()

template<class ...**Args**>
inline void **emplace**(*Args*... args)

inline *N* **top**()

inline void **pop**()

Private Members

int **count**

int **size**

pairing_node<*N*> ***root**

pairing_node<*N*> ***mem**

template<typename **P**, typename **heap_tag** = *min_heap_tag*>

class **priority_node**
#include <pairing_queue.hpp>

Public Functions

inline **priority_node**()

inline **priority_node**(int n, int r, *P* d)

inline bool **operator<**(const *priority_node*<*P*, *heap_tag*> &b) const

Public Members

int **node**

int **dirt**

P **dist**

File pathfinder.hpp

namespace **find_embedding**

template<typename **embedding_problem_t**>

class **pathfinder_base** : public *find_embedding*::*pathfinder_public_interface*
#include <pathfinder.hpp> Subclassed by *find_embedding::pathfinder_parallel< embedding_problem_t >*, *find_embedding::pathfinder_serial< embedding_problem_t >*

Public Types

using **embedding_t** = *embedding*<*embedding_problem_t*>

Public Functions

inline **pathfinder_base**(*optional_parameters* &p_, int &n_v, int &n_f, int &n_q, int &n_r, vector<vector<int>> &v_n, vector<vector<int>> &q_n)

inline virtual void **set_initial_chains**(map<int, vector<int>> chains) override
setter for the initial_chains parameter

inline virtual ~**pathfinder_base**()

inline bool **check_improvement**(const *embedding_t* &emb)
nonzero return if this is an improvement on our previous best embedding

inline virtual const *chain* &**get_chain**(int u) const override
chain accessor

inline virtual void **quickPass**(*VARORDER* varorder, int chainlength_bound, int overlap_bound, bool local_search, bool clear_first, double round_beta) override

inline virtual void **quickPass**(const vector<int> &varorder, int chainlength_bound, int overlap_bound, bool local_search, bool clear_first, double round_beta) override

inline virtual int **heuristicEmbedding**() override
perform the heuristic embedding, returning 1 if an embedding was found and 0 otherwise

Protected Functions

inline int **find_chain**(*embedding_t* &emb, const int u)
tear out and replace the chain in `emb` for variable `u`

inline int **check_stops**(const int &return_value)
internal function to check if we're supposed to stop for an external reason namely if we've timed out (which we catch immediately and return -2 to allow the heuristic to terminate gracefully), or received a keyboard interrupt (which we allow to propagate back to the user).

If neither stopping condition is encountered, return `return_value`.

inline int **initialization_pass**(*embedding_t* &emb)
sweep over all variables, either keeping them if they are pre-initialized and connected, and otherwise finding new chains for them (each, in turn, seeking connection only with neighbors that already have chains)

inline int **improve_overfill_pass**(*embedding_t* &emb)
tear up and replace each variable

inline int **pushdown_overfill_pass**(*embedding_t* &emb)
tear up and replace each chain, strictly improving or maintaining the maximum qubit fill seen by each chain

inline int **improve_chainlength_pass**(*embedding_t* &emb)
tear up and replace each chain, attempting to rebalance the chains and lower the maximum chainlength

inline void **accumulate_distance_at_chain**(const *embedding_t* &emb, const int v)
incorporate the qubit weights associated with the chain for `v` into `total_distance`

inline void **accumulate_distance**(const *embedding_t* &emb, const int v, vector<int> &visited, const int start, const int stop)
incorporate the distances associated with the chain for `v` into `total_distance`

inline void **accumulate_distance**(const *embedding_t* &emb, const int v, vector<int> &visited)
a wrapper for `accumulate_distance` and `accumulate_distance_at_chain`

inline void **compute_distances_from_chain**(const *embedding_t* &emb, const int &v, vector<int> &visited)
run dijkstra's algorithm, seeded at the chain for `v`, using the `visited` vector note: qubits are only visited if `visited[q] = 1`.

the value `-1` is used to prevent searching of overfull qubits

inline void **compute_qubit_weights**(const *embedding_t* &emb)
compute the weight of each qubit, first selecting `alpha`

inline void **compute_qubit_weights**(const *embedding_t* &emb, const int start, const int stop)
compute the weight of each qubit in the range from `start` to `stop`, where the weight is `2^(alpha*fill)` where `fill` is the number of chains which use that qubit

Protected Attributes

embedding_problem_t **ep**

optional_parameters &**params**

embedding_t **bestEmbedding**

embedding_t **lastEmbedding**

embedding_t **currEmbedding**

embedding_t **initEmbedding**

int **num_qubits**

int **num_reserved**

int **num_vars**

int **num_fixed**

vector<vector<int>> **parents**

vector<*distance_t*> **total_distance**

vector<int> **min_list**

vector<*distance_t*> **qubit_weight**

vector<int> **tmp_stats**

vector<int> **best_stats**

int **pushback**

clock::time_point **stoptime**

vector<vector<int>> **visited_list**

vector<vector<*distance_t*>> **distances**

vector<vector<int>> **qubit_permutations**

Private Functions

virtual void **prepare_root_distances**(const *embedding_t* &emb, const int u) = 0
compute the distances from all neighbors of `u` to all qubits

inline int **find_chain**(*embedding_t* &emb, const int u, int target_chainsize)
after `u` has been torn out, perform searches from each neighboring chain, select a minimum-distance root, and construct the chain

inline void **find_short_chain**(*embedding_t* &emb, const int u, const int target_chainsize)
after `u` has been torn out, perform searches from each neighboring chain, iterating over potential roots to find a root with a smallest-possible actual chainlength whereas other variants of `find_chain` simply pick a random root candidate with minimum estimated chainlength.

this procedure takes quite a long time and requires that `emb` is a valid embedding with no overlaps.

template<typename **pq_t**, typename **behavior_tag**>
inline void **dijkstra_initialize_chain**(const *embedding_t* &emb, const int &v, vector<int> &parent, vector<int> &visited, *pq_t* &pq, *behavior_tag*)
this function prepares the parent & distance-priority-queue before running dijkstra's algorithm

Friends

friend class pathfinder_serial< embedding_problem_t >

friend class pathfinder_parallel< embedding_problem_t >

struct **default_tag**

struct **embedded_tag**

template<typename **embedding_problem_t**>

class **pathfinder_parallel** : public *find_embedding*::*pathfinder_base*<*embedding_problem_t*>
#include <pathfinder.hpp> A pathfinder where the Dijkstra-from-neighboring-chain passes are done serially.

Public Types

using **super** = *pathfinder_base*<*embedding_problem_t*>

using **embedding_t** = *embedding*<*embedding_problem_t*>

Public Functions

inline **pathfinder_parallel**(*optional_parameters* &p_, int n_v, int n_f, int n_q, int n_r, vector<vector<int>> &v_n, vector<vector<int>> &q_n)

inline virtual **~pathfinder_parallel**()

inline virtual void **prepare_root_distances**(const *embedding_t* &emb, const int u) override
compute the distances from all neighbors of u to all qubits

Private Functions

inline void **run_in_thread**(const *embedding_t* &emb, const int u)

template<typename **C**>
inline void **exec_chunked**(*C* e_chunk)

template<typename **C**>
inline void **exec_indexed**(*C* e_chunk)

Private Members

int **num_threads**

vector<std::future<void>> **futures**

vector<int> **thread_weight**

mutex **get_job**

unsigned int **nbr_i**

int **neighbors_embedded**

class **pathfinder_public_interface**
#include <pathfinder.hpp> Subclassed by *find_embedding::pathfinder_base< embedding_problem_t >*

Public Functions

virtual int **heuristicEmbedding**() = 0

virtual const *chain* &**get_chain**(int) const = 0

inline virtual **~pathfinder_public_interface**()

virtual void **set_initial_chains**(map<int, vector<int>>) = 0

virtual void **quickPass**(const vector<int>&, int, int, bool, bool, double) = 0

virtual void **quickPass**(*VARORDER*, int, int, bool, bool, double) = 0

template<typename **embedding_problem_t**>

class **pathfinder_serial** : public *find_embedding*::*pathfinder_base*<*embedding_problem_t*>
#include <pathfinder.hpp> A pathfinder where the Dijkstra-from-neighboring-chain passes are done serially.

Public Types

using **super** = *pathfinder_base*<*embedding_problem_t*>

using **embedding_t** = *embedding*<*embedding_problem_t*>

Public Functions

inline **pathfinder_serial**(*optional_parameters* &p_, int n_v, int n_f, int n_q, int n_r,
vector<vector<int>> &v_n, vector<vector<int>> &q_n)

inline virtual **~pathfinder_serial**()

inline virtual void **prepare_root_distances**(const *embedding_t* &emb, const int u) override
compute the distances from all neighbors of u to all qubits

File small_cliques.hpp

namespace **busclique**

Functions

bool **find_generic_1**(const vector<size_t> &nodes, vector<vector<size_t>> &emb)

bool **find_generic_2**(const vector<pair<size_t, size_t>> &edges, vector<vector<size_t>> &emb)

bool **find_generic_3**(const vector<pair<size_t, size_t>> &edges, vector<vector<size_t>> &emb)

bool **find_generic_4**(const vector<pair<size_t, size_t>> &edges, vector<vector<size_t>> &emb)

File topo_cache.hpp

namespace **busclique**

template<typename **topo_spec**>

class **topo_cache**
#include <topo_cache.hpp>

Public Functions

topo_cache(const *topo_cache*&) = delete

topo_cache(*topo_cache*&&) = delete

inline **~topo_cache**()

inline **topo_cache**(const *topo_spec* t, const vector<size_t> &nodes, const vector<pair<size_t, size_t>> &edges)

inline void **reset**()

template<typename **serialize_tag**>
inline size_t **serialize**(*serialize_tag*, uint8_t *output) const

inline vector<size_t> **fragment_nodes**() const

inline vector<pair<size_t, size_t>> **fragment_edges**() const

inline bool **next**()

Public Members

const *topo_spec* **topo**

const *cell_cache*<*topo_spec*> **cells**

Private Functions

inline *_initializer_tag* **_initialize**(const vector<size_t> &nodes, const vector<pair<size_t, size_t>> &edges)

inline void **compute_bad_edges**()

Private Members

fat_pointer<uint8_t> **nodemask**

fat_pointer<uint8_t> **edgemask**

fat_pointer<uint8_t> **badmask**

vector<pair<size_t, size_t>> **bad_edges**

uint8_t **mask_num**

fastrng **rng**

_initializer_tag **_init**

uint8_t ***child_nodemask**

uint8_t ***child_edgemask**

class **_initializer_tag**

Class list

Class busclique::biclique_cache

template<typename **topo_spec**>

class **biclique_cache**

Class busclique::biclique_yield_cache

template<typename **topo_spec**>

class *busclique*::**biclique_yield_cache**

class **iterator**

Class busclique::biclique_yield_cache::iterator

class **iterator**

Class busclique::bundle_cache

template<typename **topo_spec**>

class **bundle_cache**

Class busclique::cell_cache

template<typename **topo_spec**>

class **cell_cache**

Class busclique::chimera_spec_base

class **chimera_spec_base** : public *busclique*::*topo_spec_base*

Class busclique::clique_cache

template<typename **topo_spec**>

class **clique_cache**

Class busclique::clique_iterator

template<typename **topo_spec**>

class **clique_iterator**

Class busclique::clique_yield_cache

template<typename **topo_spec**>

class **clique_yield_cache**

Class busclique::craphash

class **craphash**

Class busclique::fat_pointer

template<typename **T**>

class **fat_pointer**

Class busclique::ignore_badmask

class **ignore_badmask**

Class busclique::maxcache

class **maxcache**

Class busclique::pegasus_spec_base

class **pegasus_spec_base** : public *busclique*::*topo_spec_base*

Class busclique::populate_badmask

class **populate_badmask**

Class busclique::serialize_size_tag

class **serialize_size_tag**

Class busclique::serialize_write_tag

class **serialize_write_tag**

Class busclique::topo_cache

template<typename **topo_spec**>

class **topo_cache**

Class busclique::topo_cache::_initializer_tag

class **_initializer_tag**

Class busclique::topo_spec_base

class **topo_spec_base**
Subclassed by *busclique::chimera_spec_base*, *busclique::pegasus_spec_base*, *busclique::zephyr_spec_base*

Class busclique::topo_spec_cellmask

template<typename **topo_spec**>

class **topo_spec_cellmask** : public *topo_spec*

Class busclique::yieldcache

class **yieldcache**

Class busclique::zephyr_spec_base

class **zephyr_spec_base** : public *busclique*::*topo_spec_base*

Class busclique::zerocache

class **zerocache**

Class fastrng::fastrng

class **fastrng**

Class find_embedding::BadInitializationException

class **BadInitializationException** : public *minorminer*::*MinorMinerException*

Class find_embedding::CorruptEmbeddingException

class **CorruptEmbeddingException** : public *minorminer*::*MinorMinerException*

Class find_embedding::CorruptParametersException

class **CorruptParametersException** : public *minorminer*::*MinorMinerException*

Class find_embedding::LocalInteraction

class *find_embedding*::**LocalInteraction**
Interface for communication between the library and various bindings.

Any bindings of this library need to provide a concrete subclass.

Public Functions

inline void **displayOutput**(int loglevel, const string &msg) const
Print a message through the local output method.

inline void **displayError**(int loglevel, const string &msg) const
Print an error through the local output method.

inline int **cancelled**(const *clock*::time_point stoptime) const
Check if someone is trying to cancel the embedding process.

Class find_embedding::ProblemCancelledException

class **ProblemCancelledException** : public *minorminer*::*MinorMinerException*

Class find_embedding::TimeoutException

class **TimeoutException** : public *minorminer*::*MinorMinerException*

Class find_embedding::chain

class *find_embedding*::**chain**

Public Functions

inline **chain**(vector<int> &w, int l)
construct this chain, linking it to the qubit_weight vector `w` (common to all chains in an embedding, typically) and setting its variable label `l`

inline *chain* &**operator=**(const vector<int> &c)
assign this to a vector of ints.

each incoming qubit will have itself as a parent.

inline *chain* &**operator=**(const *chain* &c)
assign this to another chain

inline size_t **size**() const
number of qubits in chain

inline size_t **count**(const int q) const
returns 0 if `q` is not contained in `this`, 1 otherwise

inline int **get_link**(const int x) const
get the qubit, in `this`, which links `this` to the chain of x (if x==label, interpret the linking qubit as the chain's root)

inline void **set_link**(const int x, const int q)
set the qubit, in `this`, which links `this` to the chain of x (if x==label, interpret the linking qubit as the chain's root)

inline int **drop_link**(const int x)
discard and return the linking qubit for `x`, or -1 if that link is not set

inline void **set_root**(const int q)
insert the qubit `q` into `this`, and set `q` to be the root (represented as the linking qubit for `label`)

inline void **clear**()
empty this data structure

inline void **add_leaf**(const int q, const int parent)
add the qubit `q` as a leaf, with `parent` as its parent

inline int **trim_branch**(int q)
try to delete the qubit `q` from this chain, and keep deleting until no more qubits are free to be deleted.

return the first ancestor which cannot be deleted

inline int **trim_leaf**(int q)
try to delete the qubit `q` from this chain.

if `q` cannot be deleted, return it; otherwise return its parent

inline int **parent**(const int q) const
the parent of `q` in this chain which might be `q` but otherwise cycles should be impossible

inline void **adopt**(const int p, const int q)
assign `p` to be the parent of `q`, on condition that both `p` and `q` are contained in `this`, `q` is its own parent, and `q` is not the root

inline int **refcount**(const int q) const
: return the number of references that `this` makes to the qubit `q` where a "reference" is an occurrence of `q` as a parent or an occurrence of `q` as a linking qubit / root

inline size_t **freeze**(vector<*chain*> &others, *frozen_chain* &keep)
: store this chain into a *frozen_chain*, unlink all chains from this, and *clear()*

inline void **thaw**(vector<*chain*> &others, *frozen_chain* &keep)
: restore a *frozen_chain* into this, re-establishing links from other chains.

  precondition: this is empty.

template<typename **embedding_problem_t**>
inline void **steal**(*chain* &other, *embedding_problem_t* &ep, int chainsize = 0)
: assumes `this` and `other` have links for eachother's labels steals all qubits from `other` which are available to be taken by `this`; starting with the qubit links and updating qubit links after all

inline void **link_path**(*chain* &other, int q, const vector<int> &parents)
: link this chain to another, following the path `q`, `parent[q]`, `parent[parent[q]]`, ...

  from `this` to `other` and intermediate nodes (all but the last) into `this` (preconditions: `this` and `other` are not linked, `q` is contained in `this`, and the parent-path is eventually contained in `other`)

inline *iterator* **begin**() const
: iterator pointing to the first qubit in this chain

inline *iterator* **end**() const
: iterator pointing to the end of this chain

inline void **diagnostic**()
: run the diagnostic, and if it fails, report the failure to the user and throw a *CorruptEmbeddingException*.

  the `last_op` argument is used in the error message

inline int **run_diagnostic**() const
: run the diagnostic and return a nonzero status `r` in case of failure if(`r`&1), then the parent of a qubit is not contained in this chain if(`r`&2), then there is a refcounting error in this chain

class **iterator**

Class find_embedding::chain::iterator

class **iterator**

Class find_embedding::domain_handler_masked

class **domain_handler_masked**
: this domain handler stores masks for each variable so that prepare_visited and prepare_distances are barely more expensive than a memcopy

Class find_embedding::domain_handler_universe

class **domain_handler_universe**
this is the trivial domain handler, where every variable is allowed to use every qubit

Class find_embedding::embedding

template<typename **embedding_problem_t**>

class *find_embedding*::**embedding**
This class is how we represent and manipulate embedding objects, using as much encapsulation as possible.

We provide methods to view and modify chains.

Public Functions

inline **embedding**(*embedding_problem_t* &e_p)
constructor for an empty embedding

inline **embedding**(*embedding_problem_t* &e_p, map<int, vector<int>> &fixed_chains, map<int, vector<int>> &initial_chains)
constructor for an initial embedding: accepts fixed and initial chains, populates the embedding based on them, and attempts to link adjacent chains together.

inline *embedding*<*embedding_problem_t*> &**operator=**(const *embedding*<*embedding_problem_t*> &other)
copy the data from `other.var_embedding` into `this.var_embedding`

inline const *chain* &**get_chain**(int v) const
Get the variables in a chain.

inline unsigned int **chainsize**(int v) const
Get the size of a chain.

inline int **weight**(int q) const
Get the weight of a qubit.

inline int **max_weight**() const
Get the maximum of all qubit weights.

inline int **max_weight**(const int start, const int stop) const
Get the maximum of all qubit weights in a range.

inline bool **has_qubit**(const int v, const int q) const
Check if variable v is includes qubit q in its chain.

inline void **set_chain**(const int u, const vector<int> &incoming)
Assign a chain for variable u.

inline void **fix_chain**(const int u, const vector<int> &incoming)
Permanently assign a chain for variable u.

NOTE: This must be done before any chain is assigned to u.

inline bool **operator==**(const *embedding* &other) const
check if `this` and `other` have the same chains (up to qubit containment per chain; linking and parent information is not checked)

inline void **construct_chain**(const int u, const int q, const vector<vector<int>> &parents)
construct the chain for u, rooted at q, with a vector of parent info, where for each neibor v of u, following q -> parents[v][q] -> parents[v][parents[v][q]] ...

terminates in the chain for v

inline void **construct_chain_steiner**(const int u, const int q, const vector<vector<int>> &parents, const vector<vector<*distance_t*>> &distances, vector<vector<int>> &visited_list)
construct the chain for u, rooted at q.

for the first neighbor v of u, we follow the parents until we terminate in the chain for v q -> parents[v][q] -> .... adding all but the last node to the chain of u. for each subsequent neighbor w, we pick a nearest Steiner node, qw, from the current chain of u, and add the path starting at qw, similar to the above... qw -> parents[w][qw] -> ... this has an opportunity to make shorter chains than construct_chain

inline void **flip_back**(int u, const int target_chainsize)
distribute path segments to the neighboring chains path segments are the qubits that are ONLY used to join link_qubit[u][v] to link_qubit[u][u] and aren't used for any other variable

- if the target chainsize is zero, dump the entire segment into the neighbor
- if the target chainsize is k, stop when the neighbor's size reaches k

inline void **tear_out**(int u)
short tearout procedure blank out the chain, its linking qubits, and account for the qubits being freed

inline int **freeze_out**(int u)
undo-able tearout procedure.

similar to tear_out(u), but can be undone with thaw_back(u). note that this embedding type has a space for a single frozen chain, and freeze_out(u) overwrites the previously-frozen chain consequently, freeze_out(u) can be called an arbitrary (nonzero) number of times before thaw_back(u), but thaw_back(u) MUST be preceeded by at least one freeze_out(u). returns the size of the chain being frozen

inline void **thaw_back**(int u)
undo for the freeze_out procedure: replaces the chain previously frozen, and destroys the data in the frozen chain thaw_back(u) must be preceeded by at least one freeze_out(u) and the chain for u must currently be empty (accomplished either by tear_out(u) or freeze_out(u))

inline void **steal_all**(int u)
grow the chain for u, stealing all available qubits from neighboring variables

inline int **statistics**(vector<int> &stats) const
compute statistics for this embedding and return 1 if no chains are overlapping when no chains are overlapping, populate stats with a chainlength histogram chains do overlap, populate stats with a qubit overfill histogram a histogram, in this case, is a vector of size (maximum attained value+1) where stats[i] is either the number of qubits contained in i+2 chains or the number of chains with size i

inline bool **linked**() const
check if the embedding is fully linked that is, if each pair of adjacent variables is known to correspond to a pair of adjacent qubits

inline bool **linked**(int u) const
check if a single variable is linked with all adjacent variables.

inline void **print**() const
print out this embedding to a level of detail that is useful for debugging purposes TODO describe the output format.

inline void **long_diagnostic**(std::string current_state)
run a long diagnostic, and if debugging is enabled, record `current_state` so that the error message has a little more context.

if an error is found, throw a *CorruptEmbeddingException*

inline void **run_long_diagnostic**(std::string current_state) const
run a long diagnostic to verify the integrity of this datastructure.

the guts of this function are its documentation, because this function only exists for debugging purposes

Class find_embedding::embedding_problem

template<class **fixed_handler**, class **domain_handler**, class **output_handler**>

class **embedding_problem** : public *find_embedding*::*embedding_problem_base*, public *fixed_handler*, public *domain_handler*, public *find_embedding*::*output_handler*<verbose>
A template to construct a complete embedding problem by combining *embedding_problem_base* with fixed/domain handlers.

Class find_embedding::embedding_problem_base

class *find_embedding*::**embedding_problem_base**
Common form for all embedding problems.

Needs to be extended with a fixed handler and domain handler to be complete.

Subclassed by *find_embedding::embedding_problem< fixed_handler, domain_handler, output_handler >*

Public Functions

inline void **reset_mood**()
resets some internal, ephemeral, variables to a default state

inline void **populate_weight_table**(int max_weight)
precomputes a table of weights corresponding to various overlap values `c`, for `c` from 0 to `max_weight`, inclusive.

inline *distance_t* **weight**(unsigned int c) const
returns the precomputed weight associated with an overlap value of `c`

inline const vector<int> &**var_neighbors**(int u) const
a vector of neighbors for the variable `u`

inline const vector<int> &**var_neighbors**(int u, *shuffle_first*)
a vector of neighbors for the variable `u`, pre-shuffling them

inline const vector<int> &**var_neighbors**(int u, *rndswap_first*)
a vector of neighbors for the variable `u`, applying a random transposition before returning the reference

inline const vector<int> &**qubit_neighbors**(int q) const
a vector of neighbors for the qubit `q`

inline int **num_vars**() const
number of variables which are not fixed

inline int **num_qubits**() const
number of qubits which are not reserved

inline int **num_fixed**() const
number of fixed variables

inline int **num_reserved**() const
number of reserved qubits

inline int **randint**(int a, int b)
make a random integer between 0 and `m-1`

template<typename **A**, typename **B**>
inline void **shuffle**(*A* a, *B* b)
shuffle the data bracketed by iterators `a` and `b`

inline void **qubit_component**(int q0, vector<int> &component, vector<int> &visited)
compute the connected component of the subset `component` of qubits, containing `q0`, and using`visited` as an indicator for which qubits have been explored

inline const vector<int> &**var_order**(*VARORDER* order = VARORDER_SHUFFLE)
compute a variable ordering according to the `order` strategy

inline void **dfs_component**(int x, const vector<vector<int>> &neighbors, vector<int> &component, vector<int> &visited)
Perform a depth first search.

Public Members

optional_parameters &**params**
A mutable reference to the user specified parameters.

Class find_embedding::fixed_handler_hival

class **fixed_handler_hival**
This fixed handler is used when the fixed variables are processed before instantiation and relabeled such that variables v >= num_v are fixed and qubits q >= num_q are reserved.

Class find_embedding::fixed_handler_none

class **fixed_handler_none**
This fixed handler is used when there are no fixed variables.

Class find_embedding::max_heap_tag

class **max_heap_tag**

Class find_embedding::min_heap_tag

class **min_heap_tag**

Class find_embedding::optional_parameters

class *find_embedding*::**optional_parameters**
Set of parameters used to control the embedding process.

Public Functions

inline **optional_parameters**(*optional_parameters* &p, map<int, vector<int>> fixed_chains, map<int, vector<int>> initial_chains, map<int, vector<int>> restrict_chains)
duplicate all parameters but chain hints, and seed a new rng.

this vaguely peculiar behavior is utilized to spawn parameters for component subproblems

Public Members

LocalInteractionPtr **localInteractionPtr**
actually not controlled by user, not initialized here, but initialized in Python, MATLAB, C wrappers level

double **timeout** = 1000
Number of seconds before the process unconditionally stops.

Class find_embedding::output_handler

template<bool **verbose**>

class *find_embedding*::**output_handler**
Output handlers are used to control output.

We provide two handlers one which only reports all errors (and optimizes away all other output) and another which provides full output. When verbose is zero, we recommend the errors-only handler and otherwise, the full handler Here's the full output handler

Subclassed by *find_embedding::embedding_problem< fixed_handler, domain_handler, output_handler >*

Public Functions

template<typename ...**Args**>
inline void **error**(const char *format, *Args*... args) const
printf regardless of the verbosity level

template<typename ...**Args**>
inline void **major_info**(const char *format, *Args*... args) const
printf at the major_info verbosity level

template<typename ...**Args**>

inline void **minor_info**(const char *format, *Args*... args) const
print at the minor_info verbosity level

template<typename ...**Args**>
inline void **extra_info**(const char *format, *Args*... args) const
print at the extra_info verbosity level

template<typename ...**Args**>
inline void **debug**(const char*, *Args*...) const
print at the debug verbosity level (only works when CPPDEBUG is set)

Class find_embedding::pairing_node

template<typename **N**>

class *find_embedding*::**pairing_node** : public *N*

Public Functions

inline *pairing_node*<*N*> ***merge_roots**(*pairing_node*<*N*> *other)
the basic operation of the pairing queue put `this` and `other` into heap-order

Class find_embedding::pairing_queue

template<typename **N**>

class **pairing_queue**

Class find_embedding::parameter_processor

class **parameter_processor**

Class find_embedding::pathfinder_base

template<typename **embedding_problem_t**>

class *find_embedding*::**pathfinder_base** : public *find_embedding*::*pathfinder_public_interface*
Subclassed by *find_embedding::pathfinder_parallel< embedding_problem_t >*, *find_embedding::pathfinder_serial< embedding_problem_t >*

Public Functions

inline virtual void **set_initial_chains**(map<int, vector<int>> chains) override
setter for the initial_chains parameter

inline bool **check_improvement**(const *embedding_t* &emb)
nonzero return if this is an improvement on our previous best embedding

inline virtual const *chain* &**get_chain**(int u) const override
chain accessor

inline virtual int **heuristicEmbedding**() override
perform the heuristic embedding, returning 1 if an embedding was found and 0 otherwise

Class find_embedding::pathfinder_parallel

template<typename **embedding_problem_t**>

class *find_embedding*::**pathfinder_parallel** : public *find_embedding*::*pathfinder_base*<*embedding_problem_t*>
A pathfinder where the Dijkstra-from-neighboring-chain passes are done serially.

Public Functions

inline virtual void **prepare_root_distances**(const *embedding_t* &emb, const int u) override
compute the distances from all neighbors of u to all qubits

Class find_embedding::pathfinder_public_interface

class **pathfinder_public_interface**
Subclassed by *find_embedding::pathfinder_base< embedding_problem_t >*

Class find_embedding::pathfinder_serial

template<typename **embedding_problem_t**>

class *find_embedding*::**pathfinder_serial** : public *find_embedding*::*pathfinder_base*<*embedding_problem_t*>
A pathfinder where the Dijkstra-from-neighboring-chain passes are done serially.

Public Functions

inline virtual void **prepare_root_distances**(const *embedding_t* &emb, const int u) override
compute the distances from all neighbors of u to all qubits

Class find_embedding::pathfinder_type

template<bool **parallel**, bool **fixed**, bool **restricted**, bool **verbose**>

class **pathfinder_type**

Class find_embedding::pathfinder_wrapper

class **pathfinder_wrapper**

Class find_embedding::priority_node

template<typename **P**, typename **heap_tag** = *min_heap_tag*>

class **priority_node**

Class graph::components

class *graph*::**components**

Represents a graph as a series of connected components.

The input graph may consist of many components, they will be separated in the construction.

Public Functions

inline const std::vector<int> &**nodes**(int c) const

Get the set of nodes in a component.

inline size_t **size**() const

Get the number of connected components in the graph.

inline size_t **num_reserved**(int c) const

returns the number of reserved nodes in a component

inline size_t **size**(int c) const

Get the size (in nodes) of a component.

inline const *input_graph* &**component_graph**(int c) const

Get a const reference to the graph object of a component.

inline std::vector<std::vector<int>> **component_neighbors**(int c) const

Construct a neighborhood list for component c, with reserved nodes as sources.

template<typename **T**>
inline bool **into_component**(const int c, *T* &nodes_in, std::vector<int> &nodes_out) const

translate nodes from the input graph, to their labels in component c

template<typename **T**>
inline void **from_component**(const int c, *T* &nodes_in, std::vector<int> &nodes_out) const

translate nodes from labels in component c, back to their original input labels

Class graph::input_graph

class *graph*::**input_graph**
Represents an undirected graph as a list of edges.

Provides methods to extract those edges into neighbor lists (with options to relabel and produce directed graphs).

As an input to the library this may be a disconnected graph, but when returned from components it is a connected sub graph.

Public Functions

inline **input_graph**()
Constructs an empty graph.

inline **input_graph**(int n_v, const std::vector<int> &aside, const std::vector<int> &bside)
Constructs a graph from the provided edges.

The ends of edge ii are aside[ii] and bside[ii].

Parameters

- **n_v** – Number of nodes in the graph.
- **aside** – List of nodes describing edges.
- **bside** – List of nodes describing edges.

inline void **clear**()
Remove all edges and nodes from a graph.

inline int **a**(const int i) const
Return the nodes on either end of edge `i`

inline int **b**(const int i) const
Return the nodes on either end of edge `i`

inline size_t **num_nodes**() const
Return the size of the graph in nodes.

inline size_t **num_edges**() const
Return the size of the graph in edges.

inline void **push_back**(int ai, int bi)
Add an edge to the graph.

template<typename **T1**, typename ...**Args**>
inline std::vector<std::vector<int>> **get_neighbors_sources**(const *T1* &sources, *Args*... args) const
produce a std::vector<std::vector<int>> of neigborhoods, with certain nodes marked as sources (inbound edges are omitted) sources is either a std::vector<int> (where non-sources x have sources[x] = 0), or another type for which we have a unaryint specialization optional arguments: relabel, mask (any type with a unaryint specialization) relabel is applied to the nodes as they are placed into the neighborhood list (and not used for checking sources / mask) mask is used to filter down to the induced graph on nodes x with mask[x] = 1

template<typename **T2**, typename ...**Args**>
inline std::vector<std::vector<int>> **get_neighbors_sinks**(const *T2* &sinks, *Args*... args) const
produce a std::vector<std::vector<int>> of neigborhoods, with certain nodes marked as sinks (outbound edges are omitted) sinks is either a std::vector<int> (where non-sinks x have sinks[x] = 0), or another type for which we have a unaryint specialization optional arguments: relabel, mask (any type with a unaryint

specialization) relabel is applied to the nodes as they are placed into the neighborhood list (and not used for checking sinks / mask) mask is used to filter down to the induced graph on nodes x with mask[x] = 1

template<typename ...**Args**>
inline std::vector<std::vector<int>> **get_neighbors**(*Args*... args) const
produce a std::vector<std::vector<int>> of neigborhoods optional arguments: relabel, mask (any type with a unaryint specialization) relabel is applied to the nodes as they are placed into the neighborhood list (and not used for checking mask) mask is used to filter down to the induced graph on nodes x with mask[x] = 1

Class graph::unaryint

template<typename **T**>

class **unaryint**

Class graph::unaryint< bool >

template<>

class **unaryint**<bool>

Class graph::unaryint< int >

template<>

class **unaryint**<int>

Class graph::unaryint< std::vector< int > >

template<>

class **unaryint**<std::vector<int>>

Class graph::unaryint< void * >

template<>

class **unaryint**<void*>
this one is a little weird construct a unaryint(nullptr) and get back the identity function f(x) -> x

Class minorminer::MinorMinerException

class **MinorMinerException** : public runtime_error
Subclassed by *find_embedding::BadInitializationException*, *find_embedding::CorruptEmbeddingException*, *find_embedding::CorruptParametersException*, *find_embedding::ProblemCancelledException*, *find_embedding::TimeoutException*

Struct list

Struct find_embedding::frozen_chain

struct **frozen_chain**

This class stores chains for embeddings, and performs qubit-use accounting.

The `label` is the index number for the variable represented by this chain. The `links` member of a chain is an unordered map storing the linking information for this chain. The `data` member of a chain stores the connectivity information for the chain.

Links: If `u` and `v` are variables which are connected by an edge, the following must be true: either chain_u or chain_v is empty,

or

chain_u.links[v] is a key in chain_u.data, chain_v.links[u] is a key in chain_v.data, and (chain_u.links[v], chain_v.links[u]) are adjacent in the qubit graph

Moreover, (chain_u.links[u]) must exist if chain_u is not empty, and this is considered the root of the chain.

Data: The `data` member stores the connectivity information. More precisely, `data` is a mapping `qubit->(parent, refs)` where: `parent` is also contained in the chain `refs` is the total number of references to `qubit`, counting both parents and links the chain root is its own parent.

Struct find_embedding::pathfinder_base::default_tag

struct **default_tag**

Struct find_embedding::pathfinder_base::embedded_tag

struct **embedded_tag**

Struct find_embedding::rndswap_first

struct **rndswap_first**

Struct find_embedding::shuffle_first

struct **shuffle_first**

1.3 Installation

1.3.1 Python

pip installation is recommended for platforms with precompiled wheels posted to pypi. Source distributions are provided as well.

```
pip install minorminer
```

To install from this repository, run the *setuptools* script.

```
pip install -r requirements.txt
python setup.py install
# optionally, run the tests to check your build
pip install -r tests/requirements.txt
python -m nose . --exe
```

1.3.2 C++

The *CMakeLists.txt* in the root of this repo will build the library and optionally run a series of tests. On Linux, the commands would be something like this:

```
mkdir build; cd build
cmake ..
make
```

To build the tests, turn the CMake option *MINORMINER_BUILD_TESTS* on. The command line option for CMake to do this would be *-DMINORMINER_BUILD_TESTS=ON*.

1.3.3 Library Usage

C++11 programs should be able to use this as a header-only library. If your project is using CMake, this library can be used fairly simply; if you have checked out this repo as *externals/minorminer* in your project, you would need to add the following lines to your *CMakeLists.txt*

```
add_subdirectory(externals/minorminer)

# After your target is defined
target_link_libraries(your_target minorminer pthread)
```

1.4 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.

PYTHON MODULE INDEX

m

INDEX

B

C

D

F

G

I

L

M

P