minorminer Documentation

Release 0.1.8

D-Wave Systems

Mar 07, 2019

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.

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

CHAPTER 1

Documentation

1.1 Introduction

1.1.1 Examples

This example minor embeds a triangular source K4 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
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]]
```

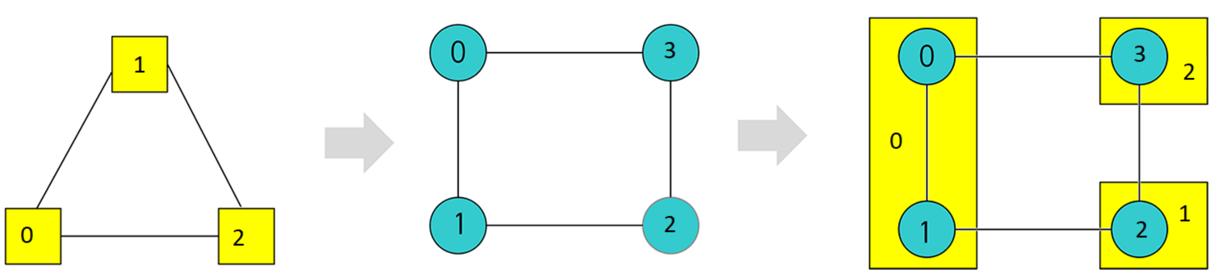

Fig. 1: Embedding a K_3 source graph into a square target graph by chaining two target nodes to represent one source node.

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_embedding(clique, target_graph)

print(embedding)
# 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]}
```

1.2 Reference Documentation

1.2.1 Python Interface

1.2.2 C++ Library

Namespace list

Namespace find_embedding

namespace find_embedding

Typedefs

using find_embedding::distance_t = typedef long long int

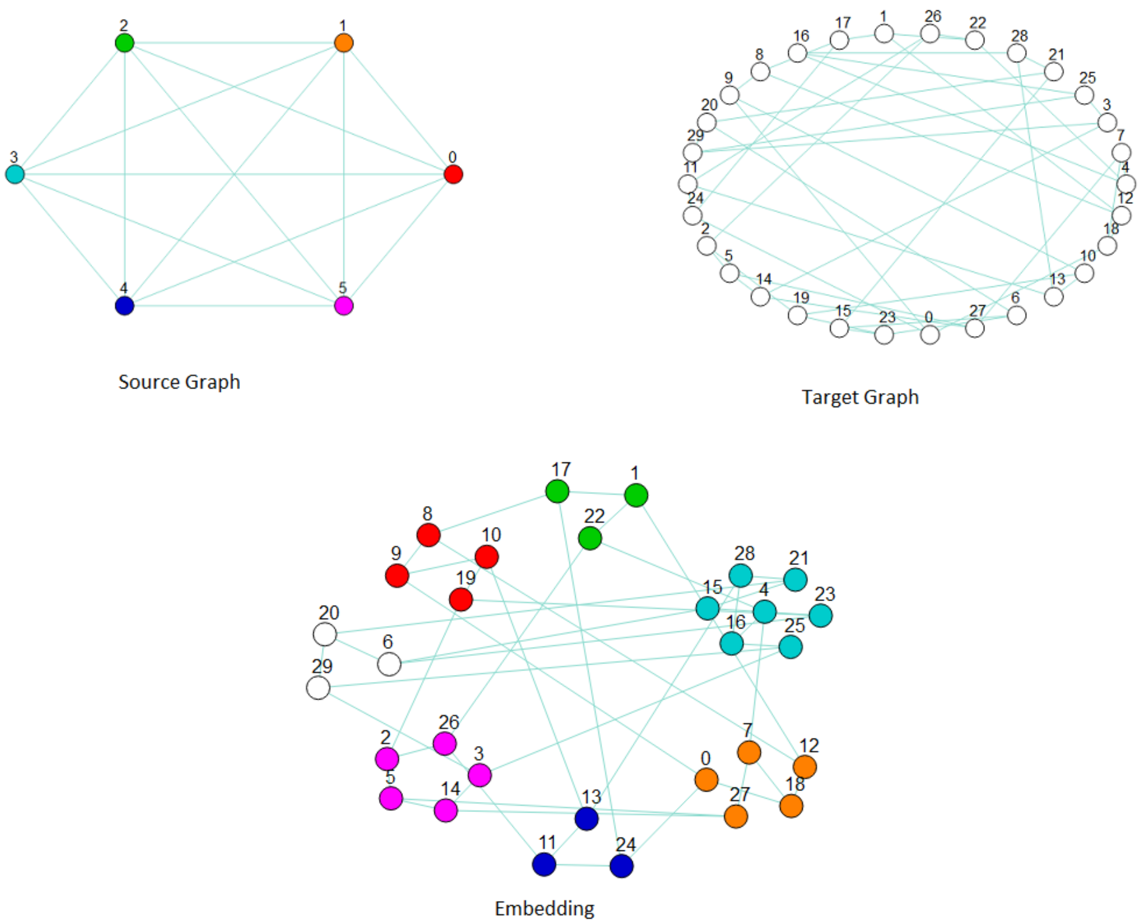

Fig. 2: Embedding a K_6 source graph (upper left) into a 30-node random target graph of degree 3 (upper right) by chaining several target nodes to represent one source node (bottom). The graphic of the embedding clusters chains representing nodes in the source graph: the cluster of red nodes is a chain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

using find_embedding::RANDOM = typedef fastrng

using find_embedding::clock = typedef std::chrono::high_resolution_clock

using find_embedding::min_queue = typedef std::priority_queue<priority_node<P, min_hea

using find_embedding::max_queue = typedef std::priority_queue<priority_node<P, max_hea

using find_embedding::distance_queue = typedef pairing_queue<priority_node<distance_t,

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

Enums

enum VARORDER

Values:

VARORDER_SHUFFLE

VARORDER_DFS

VARORDER_BFS

VARORDER_PFS

VARORDER_RPFS

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 chain

#include <chain.hpp>

Public Functions

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`

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

assign this to a vector of ints.

each incoming qubit will have itself as a parent.

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

assign this to another chain

int **size**() **const**

number of qubits in chain

int **count**(**const** int *q*) **const**

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

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)

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)

int **drop_link**(**const** int *x*)

discard and return the linking qubit for `x`, or -1 if that link is not set

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`)

void **clear**()

empty this data structure

void **add_leaf**(**const** int *q*, **const** int *parent*)

add the qubit `q` as a leaf, with `parent` as its parent

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

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

int **parent**(**const** int *q*) **const**

the parent of `q` in this chain which might be `q` but otherwise cycles should be impossible

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

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

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

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>
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

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`)

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

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

void **diagnostic** (char **last_op*)
: run the diagnostic, and if it fails, report the failure to the user and throw -1.

the `last_op` argument is used in the error message

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 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

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

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.

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

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

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

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

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

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

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

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

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.

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)

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`

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`

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

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

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

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)`)

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

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`

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

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

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

void **long_diagnostic** (char **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 -1

void **run_long_diagnostic** (char **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** dom

#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

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

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.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

class optional_parameters

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

Public Functions

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 output_handler_error

#include <embedding_problem.hpp> Here's the errors-only handler.

Public Functions

template <typename... *Args*>
void **error**(**const** char **format*, Args... *args*) **const**

printf regardless of the verbosity level

template <typename... *Args*>
void **major_info**(Args...) **const**

printf at the major_info verbosity level

template <typename... *Args*>
void **minor_info**(Args...) **const**

print at the minor_info verbosity level

template <typename... *Args*>
void **extra_info**(Args...) **const**

print at the extra_info verbosity level

template <typename... *Args*>
void **debug**(Args...) **const**

print at the debug verbosity level (only works when `CPPDEBUG` is set)

class output_handler_full

#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

Public Functions

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

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

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

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

template <typename... *Args*>
void **debug**(**const** char **ONDEBUGformat*, Args... *ONDEBUGargs*) **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

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** 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

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

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

virtual int **heuristicEmbedding**()
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

virtual void **prepare_root_distances** (**const** embedding_t &*emb*, **const** int *u*)
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

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

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

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

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

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

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

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

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>
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>

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

input_graph ()
Constructs an empty graph.

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.

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

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

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

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

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

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

template <**typename** T1, typename... *Args*>
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*>
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*>
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 <>
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

Class list

Class fastrng

class fastrng

Class find_embedding::BadInitializationException

class BadInitializationException : **public** *find_embedding*::*MinorMinerException*

Class find_embedding::CorruptEmbeddingException

class CorruptEmbeddingException : **public** *find_embedding*::*MinorMinerException*

Class find_embedding::CorruptParametersException

class CorruptParametersException : **public** *find_embedding*::*MinorMinerException*

Class find_embedding::LocalInteraction

class LocalInteraction
Interface for communication between the library and various bindings.

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

Public Functions

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

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

Class find_embedding::MinorMinerException

class MinorMinerException : **public** runtime_error

Subclassed by *find_embedding::BadInitializationException*, *find_embedding::CorruptEmbeddingException*, *find_embedding::CorruptParametersException*, *find_embedding::ProblemCancelledException*, *find_embedding::TimeoutException*

Class find_embedding::ProblemCancelledException

class ProblemCancelledException : **public** *find_embedding*::*MinorMinerException*

Class find_embedding::TimeoutException

class TimeoutException : **public** *find_embedding*::*MinorMinerException*

Class find_embedding::chain

class chain

Public Functions

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`

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

assign this to a vector of ints.

each incoming qubit will have itself as a parent.

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

assign this to another chain

int **size**() **const**

number of qubits in chain

int **count**(**const** int *q*) **const**

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

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)

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)

int **drop_link**(**const** int *x*)

discard and return the linking qubit for `x`, or -1 if that link is not set

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`)

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

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

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

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

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

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

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

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

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>
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

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`)

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

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

void **diagnostic**(char *last_op)
: run the diagnostic, and if it fails, report the failure to the user and throw -1.

  the `last_op` argument is used in the error message

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 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 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

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

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.

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

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

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

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

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

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

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

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

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.

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)

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`

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`

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

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

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

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)`)

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

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`

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

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

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

void **long_diagnostic** (char **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 -1

void **run_long_diagnostic** (char **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_ha
A template to construct a complete embedding problem by combining `embedding_problem_base` with fixed/domain handlers.

Class find_embedding::embedding_problem_base

class 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

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

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.

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

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

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

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

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

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

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

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

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

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

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

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

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

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 optional_parameters
Set of parameters used to control the embedding process.

Public Functions

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_error

class output_handler_error
Here's the errors-only handler.

Public Functions

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

template <typename… *Args*>
void **major_info**(Args…) **const**
printf at the major_info verbosity level

template <typename... *Args*>
void **minor_info**(Args...) **const**
print at the minor_info verbosity level

template <typename... *Args*>
void **extra_info**(Args...) **const**
print at the extra_info verbosity level

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

Class find_embedding::output_handler_full

class output_handler_full
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

Public Functions

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

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

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

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

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

Class find_embedding::pairing_node

template <typename *N*>
class pairing_node : **public** N

Public Functions

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 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

int **check_improvement** (**const** embedding_t &*emb*)

nonzero return if this is an improvement on our previous best embedding

virtual const *chain* &**get_chain** (int *u*) **const**

chain accessor

virtual int **heuristicEmbedding** ()

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 pathfinder_parallel : **public** *find_embedding*::pathfinder_base<embedding_problem_t>

A pathfinder where the Dijkstra-from-neighboring-chain passes are done serially.

Public Functions

virtual void **prepare_root_distances** (**const** embedding_t &*emb*, **const** int *u*)

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

Public Functions

virtual void **prepare_root_distances**(**const** embedding_t &*emb*, **const** int *u*)
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 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

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

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

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

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

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

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>
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>
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 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

input_graph ()
Constructs an empty graph.

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.

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

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

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

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

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

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

template <**typename** T1, typename... *Args*>

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*>
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*>
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 <>
template<>
class unaryint<bool>

Class graph::unaryint< int >

template <>
template<>
class unaryint<int>

Class graph::unaryint< std::vector< int > >

template <>
template<>
class unaryint<std::vector<int>>

Class graph::unaryint< void * >

template <>
template<>

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

File list

File chain.hpp

Defines

DIAGNOSE2 (other, X)

DIAGNOSE (X)

namespace find_embedding

class chain
#include <chain.hpp>

Public Functions

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`

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

each incoming qubit will have itself as a parent.

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

int **size** () **const**
number of qubits in chain

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

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)

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)

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

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`)

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

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

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

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

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

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

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

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

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>
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

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`)

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

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

void **diagnostic** (char **last_op*)
run the diagnostic, and if it fails, report the failure to the user and throw -1.

the `last_op` argument is used in the error message

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

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

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

find_embedding::chain::iterator::iterator(typename decltype(data)::const_iterato

iterator **operator++**()

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

decltype(data) const ::key_type& find_embedding::chain::iterator::operator*() co

Private Members

decltype(data) ::const_iterator find_embedding::chain::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

void **clear**()

Public Members

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

unordered_map<int, int> **links**

File debug.hpp

Defines

minorminer_assert(X)

ONDEBUG(X)

File embedding.hpp

Defines

DIAGNOSE(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

embedding(embedding_problem_t *&e_p*)

constructor for an empty embedding

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.

embedding<embedding_problem_t> &**operator=**(**const** *embedding*<embedding_problem_t> *&other*)

copy the data from `other.var_embedding` into `this.var_embedding`

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

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

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

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

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

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

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

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.

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)

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`

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`

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

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

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

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)`)

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

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`

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

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

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

void **long_diagnostic** (char **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 -1

void **run_long_diagnostic** (char **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

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:

VARORDER_SHUFFLE

VARORDER_DFS

VARORDER_BFS

VARORDER_PFS

VARORDER_RPFS

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

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

virtual ~domain_handler_masked()

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

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

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

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

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

virtual ~domain_handler_universe()

Public Static Functions

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

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

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

static 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** dom

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

Public Functions

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*)

virtual ~embedding_problem()

Private Types

template<>
using ep_t = *embedding_problem_base*

template<>
using fh_t = fixed_handler

template<>
using dh_t = domain_handler

template<>
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

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*)

virtual ~embedding_problem_base()

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

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.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

template <**typename** queue_t>
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)

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

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

virtual ~fixed_handler_hival()

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

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

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

virtual ~fixed_handler_none()

Public Static Functions

static bool **fixed**(int)

static bool **reserved**(int)

class output_handler_error
#include <embedding_problem.hpp> Here's the errors-only handler.

Public Functions

output_handler_error(*optional_parameters* &*p*)

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

template <typename... *Args*>
void **major_info**(Args...) **const**
printf at the major_info verbosity level

template <typename... *Args*>
void **minor_info**(Args...) **const**
print at the minor_info verbosity level

template <typename... *Args*>
void **extra_info**(Args...) **const**
print at the extra_info verbosity level

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

Private Members

optional_parameters &**params**

class output_handler_full
#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

Public Functions

output_handler_full(*optional_parameters* &*p*)

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

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

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

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

template <typename... *Args*>

void **debug**(**const** char **ONDEBUGformat*, Args... *ONDEBUGargs*) **const**
print at the debug verbosity level (only works when `CPPDEBUG` is set)

Private Members

optional_parameters &**params**

File fastrng.hpp

class fastrng
#include <fastrng.hpp>

Public Types

typedef uint64_t **result_type**

Public Functions

fastrng()

fastrng(uint64_t *x*)

void **seed**(uint32_t *x*)

void **seed**(uint64_t *x*)

uint64_t **operator()**()

void **discard**(int *n*)

Public Static Functions

static constexpr uint64_t **min**()

static constexpr uint64_t **max**()

Private Members

uint64_t **S0**

uint64_t **S1**

Private Static Functions

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

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

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

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

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

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

Public Members

int **num_vars**

int **num_qubits**

vector<int> **qub_reserved_unscrewed**

vector<int> **var_fixed_unscrewed**

int **num_reserved**

graph::components **qub_components**

int **problem_qubits**

int **problem_reserved**

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

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

vector<int> **_filter_fixed_vars**()

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 std::conditional<verbose, *output_handler_full*, *output_handler_error*>::type **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*

class pathfinder_wrapper
#include <find_embedding.hpp>

Public Functions

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

~pathfinder_wrapper()

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

int **heuristicEmbedding**()

int **num_vars**()

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

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

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*>
std::unique_ptr<*pathfinder_public_interface*> **_pf_parse4**(Args&&... *args*)

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

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

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

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

Public Functions

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

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

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

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

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

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

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

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

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

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>
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>
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

int **__init_find**(int *x*)

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

input_graph()
Constructs an empty graph.

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.

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

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

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

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

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

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

template <**typename** T1, typename... *Args*>
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*>
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*>
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

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>
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>
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 throgh unaryint

Private Members

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

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

int **_num_nodes**

template <>
template<>
class unaryint<bool>

#include <graph.hpp>

Public Functions

unaryint (**const** bool *x*)

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

Private Members

const bool **b**

template <>
template<>
class unaryint<int>

#include <graph.hpp>

Public Functions

unaryint (int *m*)

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

Private Members

const int **b**

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

Public Functions

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

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

Private Members

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

template <>
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

unaryint (void ***const** &)

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

File pairing_queue.hpp

namespace find_embedding

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

Public Functions

pairing_node()

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

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*>
void **refresh**(Args... *args*)

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

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

Private Functions

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

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

pairing_queue(int *n*)

pairing_queue(*pairing_queue* &&*other*)

~pairing_queue()

void **reset**()

bool **empty**()

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

N **top**()

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

priority_node()

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

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

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

Public Functions

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*)

void **set_initial_chains**(map<int, vector<int>> *chains*)

virtual ~pathfinder_base()

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

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

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

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

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

Protected Functions

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

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)

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

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

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

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`

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`

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

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

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

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

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

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>
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 find_embedding::pathfinder_serial< embedding_problem_t >

friend find_embedding::pathfinder_parallel< embedding_problem_t >

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

template<>
using super = pathfinder_base<embedding_problem_t>

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

Public Functions

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*)

virtual ~pathfinder_parallel()

virtual void **prepare_root_distances**(**const** embedding_t &*emb*, **const** int *u*)

compute the distances from all neighbors of `u` to all qubits

Private Functions

void **run_in_thread**(**const** embedding_t &*emb*, **const** int *u*)

template <**typename** C>
void **exec_chunked**(C *e_chunk*)

template <**typename** C>
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

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

template<>

using super = pathfinder_base<embedding_problem_t>

template<>

using embedding_t = *embedding*<embedding_problem_t>

Public Functions

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*)

virtual ~pathfinder_serial()

virtual void **prepare_root_distances**(**const** embedding_t &*emb*, **const** int *u*)

compute the distances from all neighbors of `u` to all qubits

File util.hpp

namespace find_embedding

Typedefs

using find_embedding::distance_t = typedef long long int

using find_embedding::RANDOM = typedef fastrng

using find_embedding::clock = typedef std::chrono::high_resolution_clock

using find_embedding::min_queue = typedef std::priority_queue<priority_node<P, min_hea

using find_embedding::max_queue = typedef std::priority_queue<priority_node<P, max_hea

using find_embedding::distance_queue = typedef pairing_queue<priority_node<distance_t,

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

Functions

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** *find_embedding*::*MinorMinerException*

#include <util.hpp>

Public Functions

BadInitializationException (**const** string &*m* = "bad embedding used with skip_initialization")

class CorruptEmbeddingException : **public** *find_embedding*::*MinorMinerException*

#include <util.hpp>

Public Functions

CorruptEmbeddingException (**const** string &*m* = "chains may be invalid")

class CorruptParametersException : **public** *find_embedding*::*MinorMinerException*

#include <util.hpp>

Public Functions

CorruptParametersException (**const** string &*m* = "chain inputs are corrupted")

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

virtual ~LocalInteraction()

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

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

Private Functions

virtual void **displayOutputImpl**(**const** string&) **const** = 0
Print the string to a binding specified sink.

virtual bool **timedOutImpl**(**const** clock::time_point *stoptime*) **const**
Check if the embedding process has timed out.

virtual bool **cancelledImpl**() **const** = 0
Check if someone has tried to cancel the embedding process.

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

Public Functions

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

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

Public Functions

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

template <typename... *Args*>
void **printx**(**const** char **format*, Args... *args*) **const**

template <typename... *Args*>
void **error**(**const** char **format*, Args... *args*) **const**

template <typename... *Args*>
void **major_info**(**const** char **format*, Args... *args*) **const**

template <typename... *Args*>
void **minor_info**(**const** char **format*, Args... *args*) **const**

template <typename... *Args*>

void **extra_info** (**const** char **format*, Args... *args*) **const**

template <typename... *Args*>
void **debug** (**const** char **format*, Args... *args*) **const**

optional_parameters ()

void **seed** (uint64_t *randomSeed*)

Public Members

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

int **max_no_improvement** = 10

RANDOM **rng**

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

double **max_beta** = numeric_limits<double>::max()

int **tries** = 10

int **verbose** = 0

int **inner_rounds** = numeric_limits<int>::max()

int **max_fill** = numeric_limits<int>::max()

bool **return_overlap** = false

int **chainlength_patience** = 2

int **threads** = 1

bool **skip_initialization** = false

map<int, vector<int>> **fixed_chains**

map<int, vector<int>> **initial_chains**

map<int, vector<int>> **restrict_chains**

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

Public Functions

ProblemCancelledException (**const** string &*m* = "embedding cancelled by keyboard interrupt")

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

Public Functions

TimeoutException (**const** string &*m* = "embedding timed out")

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 cython==0.27
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

"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.

1.5 Ocean Overview

D-Wave Ocean includes various projects/repositories on Github that help solve problems on the D-Wave system.

Learn about D-Wave's Ocean and how its projects work together at D-Wave Ocean on Read the Docs.

1.6 Contributing to Ocean

D-Wave welcomes contributions to Ocean projects.

See how to contribute at Ocean Contributors.

1.7 Glossary

Quantum computing and the types of problems it addresses use many domain-specific terms.

Learn the relevant terminology at Ocean Glossary.

1.8 D-Wave

D-Wave Systems is the leader in the development and delivery of quantum computing systems and software, and the world's only commercial supplier of quantum computers.

Learn more about D-Wave at D-Wave Systems.

1.9 Leap

Leap, launched in 2018, is the real-time Quantum Application Environment from D-Wave Systems Inc. Leap brings quantum computing to the real world by providing cloud access to our systems, for free, to anyone who wants to give it a try. Learn about the types of problems that the D-Wave quantum computer can solve, run interactive demos and coding examples on the quantum computer, contribute your coding ideas, and join the growing conversation in our community of like-minded users at Leap.

1.10 D-Wave System Documentation

D-Wave System Documentation describes the D-Wave system, its properties and parameters, and provides information on solving problems using a D-Wave system. Learn about the D-Wave quantum computer at D-Wave System Documentation.

Index

D

F

G

M

O