cliquematch

Release 3.0.0

Gautham Venkatasubramanian

Jan 31, 2022

CONTENTS:

CHAPTER

ONE

FINDING CORRESPONDENCE VIA MAXIMUM CLIQUES IN LARGE GRAPHS

The *cliquematch* package aims to do two specific things:

1. Find maximum cliques in large sparse undirected graphs, as quickly and efficiently as possible. *cliquematch* v3 also enables finding vertex-weighted maximum cliques via the *NWGraph* class.
2. Construct large sparse undirected graphs in-memory for the various applications of the maximum clique/clique enumeration problem.

That's it. *cliquematch* does not provide a way to modify a large graph once it has been loaded, or any other general capability, because:

- the internal data structures are designed to optimize the clique search
- sparse graphs constructed for applications of the maximum clique problem are rarely modified by hand (constructed anew instead)
- there are better packages (`networkx` and `igraph`) for general graph analysis.

Note: This is *cliquematch* v3, which has a simpler API than v1 and also provides the capability of clique enumeration. You view the v1 docs here.

1.1 Installing `cliquematch`

cliquematch uses `pybind11` to provide Python wrappers. Internally, the core clique search algorithm is implemented in `C++11`, simple template classes are used to provide flexibility for applications, and `Eigen` is used to provide fast access to numpy arrays.

1.1.1 Installing from a wheel

PyPI wheels are available for Linux and Windows.

```
pip install cliquematch>=3.0.0
```

1.1.2 Installing from source

1. *cliquematch* requires pybind11 (v2.2 or newer) for its setup:

```
pip3 install pybind11
```

2. *cliquematch* requires Eigen (v3.3.7 or newer) as part of its setup.
   - You can clone the Github repo via `git clone --recursive` to get Eigen.
   - If you already have an existing version of Eigen or have downloaded it separately, set the `EIGEN_DIR` environment variable to the folder containing Eigen before compilation.
3. A C++11 compatible compiler must be available for the installation:
   - On Linux, gcc is called with `--std=c++11` (builds with `gcc 4.8.2` for `manylinux1` wheels).
   - On Windows, Visual Studio 2015 Update 3 (MSVC 14.0 runtime) or later is needed.
   - **Note:** Installing under Windows+MinGW has not been tested.

1.2 User Guide

As described in the homepage, *cliquematch* aims to do two things:

1. Find maximum cliques in large sparse undirected graphs, as quickly and efficiently as possible.
2. Construct large sparse undirected graphs in-memory for the various applications of the maximum clique/clique enumeration problem.

Let's look at how *cliquematch* can be used as described above.

1.2.1 Finding a Maximum Clique

Note: The below examples use python. If you're using R and like to call python packages via reticulate, you can view the R Markdown file from the examples in the Github repo.

Finding a maximum clique in an undirected graph is a well known problem. Most exact algorithms, including the one used in *cliquematch*, use some form of a depth-first search (DFS), along with some pruning techniques to speed up the search.

Finding a maximum clique using *cliquematch* is simple (and *fast*): load a *Graph* from a file (or from an adjacency matrix, an adjacency list, or a list of edges), set some search properties, and find a maximum clique in it.

For example, we load a graph from a file: cond-mat-2003.mtx and find a maximum clique.

```
import cliquematch
G = cliquematch.Graph.from_file("cond-mat-2003.mtx")
```

We can also load a *Graph* using an adjacency matrix (`from_adj_matrix`), an adjacency list (`from_adj_list`), or a list of edges (*from_edgelist*). Once the *Graph* object has been loaded, we can provide various parameters to search for a clique with *get_max_clique*:

```
answer = G.get_max_clique(
    lower_bound=1,
    upper_bound=1729,
    time_limit=100,
    use_heuristic=True,
    use_dfs=True,
)
print(answer)
# [9986, 9987, 10066, 10068, 10071, 10072, 10074, 10076,
# 10077, 10078, 10079, 10080, 10081, 10082, 10083, 10085,
# 10287, 10902, 10903, 10904, 10905, 10906, 10907, 10908, 10909]
```

- A `lower_bound` and an `upper_bound` can be set for the size for clique.
- A `time_limit` can be set for the clique search. If time limits are not needed, set `0` as the limit.
- In case the graph is very large, to get a large clique quickly via a heuristic method, `use_heuristic` can be set to `True`, and `use_dfs` can be set to `False` to skip the depth-first search.

Warning: Both `use_heuristic` and `use_dfs` cannot be set to `False`. *cliquematch* will then skip the clique search entirely and raise a `RuntimeError`.

The *Graph* object has some read-only properties:

- *n_vertices* and *n_edges* provide the number of vertices and edges in the graph.
- *search_done* is `True` if the depth-first search has been completed.

The user can print the *Graph* object to view the properties:

```
print(G)
# cliquematch.core.Graph object at 0x559e7da730c0
# (n_vertices=31163,n_edges=120029,search_done=False)
```

What is the *search_done* property for? In case the clique search was interrupted due to `time_limit`, the user can provide a `continue_search` parameter within a loop like this:

```
while not G.search_done:
    answer = G.get_max_clique(
        lower_bound=1,
        upper_bound=1729,
        time_limit=100,
        use_heuristic=True,
        use_dfs=True,
        continue_search=True,
    )
```

If the entire *Graph* needs to be searched again (say for a clique of larger size), the user can call the *reset_search* method:

```
G.reset_search()
G.get_max_clique(
    lower_bound=1, upper_bound=31, time_limit=100, use_heuristic=True, use_dfs=True
)
```

1.2.2 Finding Multiple cliques : clique enumeration

The above *Graph* methods deal with finding *one* large clique. A related use case is to find *multiple* large cliques in a given graph,and iterate through them in some order.

The *Graph* class has an *all_cliques* method for finding all cliques of a given size. Taking the *Graph* object loaded as per the above section:

```
import cliquematch
G = cliquematch.Graph.from_file("cond-mat-2003.mtx")
# we know there exists a maximum clique of size 25
# so let's find cliques of size 24
for clique in G.all_cliques(size=24):
    print(clique)
```

1.2.3 Applications of the maximum clique problem

Applications of the maximum clique problem primarily involve:

1. the construction of a graph from a different kind of dataset,
2. writing the graph to a file,
3. reading the file again,
4. finding a maximum clique, and then
5. conversion of the clique back into the existing dataset.

This process is usually repeated with tweaks to underlying dataset, leading to different graphs and cliques. For such use cases the primary bottlenecks are the construction of the graph, reading/processing the graph data in a clique-friendly manner, and finding the maximum clique. *cliquematch* aims to solve these issues by keeping the graph construction in memory, and having an optimized clique search algorithm.

Graph construction for maximum clique problems mostly involve one of the two ways below:

- A graph is constructed using an *edge indication function* on all pairs of elements belonging to a dataset X.
  - *cliquematch* does not provide any specific code for this; an edge list can constructed from the data using a nested loop, following which a *Graph* object can be loaded,and the maximum clique (ie the largest group of related elements in the dataset X) can be computed.
- A *correspondence graph* is constructed using the elements of *two* datasets P and Q; the vertices of the graph refer to pairs of elements (p_i, q_j), and an edge between two vertices implies some common relationship between the elements from P and the elements from Q.

Yeah... that's a little dense. Let's try again with some math.

Assume you have two sets $P = p_1, p_2, ...p_M$ and $Q = q_1, q_2, ...q_N$, and we want to find the **largest** subsets $P^* \in P$ and $Q^* \in Q$ such that there exists a *one-to-one correspondence* between P^* and Q^*.

This means the elements of P^* are related to each other similar to how the elements of Q^* are related to each other. Suppose the elements of P^* (and similarly Q^*) have a *pairwise* relationship, then we can say that for all pairs

$$\begin{aligned}((p_{i_1}, p_{i_2}), (q_{j_1}, q_{j_2})) \in P^* \times P^* \times Q^* \times Q^* \\ i_1 \neq i_2 \\ j_1 \neq j_2\end{aligned}$$

there exists some boolean *condition function* f such that`

$$\begin{aligned} f(p_{i_1}, p_{i_2}, q_{j_1}, q_{j_2}) &= 1 \\ \forall (p_{i_1}, p_{i_2}, q_{j_1}, q_{j_2}) \in P^* \times P^* \times Q^* \times Q^* \\ i_1 &\neq i_2 \\ j_1 &\neq j_2 \end{aligned}$$

What does this have to do with maximum cliques? Well, P^* and Q^* are the **largest** such subsets, so maybe finding them can be done by converting the problem to a maximum clique problem. This is where we bring in a *correspondence graph*: an undirected graph $G(V, E)$, where $V = P \times Q$ ie the vertices indicate a mapping (p_i, q_j). As for edges, that's where f comes in: an edge exists between $v_1 = (p_{i_1}, q_{j_1})$ and $v_2 = (p_{i_2}, q_{j_2})$ if and only if:

$$f(p_{i_1}, p_{i_2}, q_{j_1}, q_{j_2}) = 1$$

It can be proved that finding a maximum clique in the correspondence graph G is the same as finding the largest subsets P^* and Q^* that have a one-to-one correspondence.

cliquematch provides classes the implement the above functionality of correspondence graphs. The user has to provide are the sets P and Q, along with a condition function f. A common use case is to have f expressed using *distance metrics*:

$$f(p_{i_1}, p_{i_2}, q_{j_1}, q_{j_2}) = 1 \iff ||d_P(p_{i_1}, p_{i_2}) - d_Q(q_{j_1}, q_{j_2})|| \leq \epsilon$$

The correspondence graph classes are all subclasses of *Graph*, they all expose the same methods:

- An `__init__` method that accepts the sets P (or `S1`), Q (or `S2`), and the distance functions for `S1` and `S2` respectively.
- A *build_edges* method that constructs the correspondence graph using only the distance metrics.
- A *build_edges_with_condition* method that accepts a condition function `cf`, and a boolean `use_condition_only`:
  - If `use_cfunc_only` is `True`, the graph is constructed using only `cf` (slower)
  - Otherwise the graph is constructed using the distance metrics and pruned with `cf` (faster)
- A *get_correspondence* method that returns the largest corresponding subsets P^* and Q^*, or the indices of the elements in the subsets.
- A *all_correspondences* method similar to the *cliquematch.Graph.all_cliques* that works similar to *get_correspondence*.

The correspondence graph classes available are:

- *cliquematch.A2AGraph* where `S1` and `S2` are 2-D `numpy.ndarray`s
- *cliquematch.L2LGraph* where `S1` and `S2` are `list`s (or any list-like object)
- *cliquematch.A2LGraph* where `S1` is a 2-D `numpy.ndarray` and `S2` is a `list`
- *cliquematch.L2AGraph* where `S2` is a `list` and `S2` is a 2-D `numpy.ndarray`
- *cliquematch.IsoGraph* where `S1` and `S2` are *Graph*s (subgraph isomorphism).
- *cliquematch.AlignGraph* which is a special case of *A2AGraph* used for image alignment.

The concept of correspondence graphs enables applying maximum cliques to many fields. Here are a couple of examples from the Github repo:

1. This image matching algorithm can be implemented using *cliquematch* like this .
2. Simple molecular alignment can be implemented like this .

1.3 API Documentation

class cliquematch.**Graph**

search_done
Whether the search has been completed (Readonly)

Type bool

n_vertices
Number of vertices in the graph (Readonly)

Type int

n_edges
Number of edges in the graph (Readonly)

Type int

get_max_clique()
Finds a maximum clique in graph within the given bounds

Parameters

- **lower_bound** (*int*) – set a lower bound on the clique size. default is **1**.
- **upper_bound** (*int*) – set an upper bound on the clique size. default is **65535**.
- **time_limit** (*float*) – set a time limit for the search: a nonpositive value implies there is no time limit (use in conjunction with continue_search). default is **-1**.
- **use_heuristic** (*bool*) – if True, use the heuristic-based search to obtain a large clique quickly. Good for obtaining an initial lower bound. default is True.
- **use_dfs** (*bool*) – if True, use the depth-first to obtain the clique. default is True.
- **continue_search** (*bool*) – set as True to continue a clique search interrupted by time_limit. default is False.

Returns the vertices in the maximum clique

Return type list

Raises **RuntimeError** – if the graph is empty or a clique could not be found

reset_search()
Reset the search space for *get_max_clique*.

Raises **RuntimeError** – if the graph is empty

all_cliques(*size*)
Iterate through all cliques of a given size in the *Graph*.

Parameters **size** (*int*) – size of a clique to search for.

Return type *CliqueIterator*

Raises **RuntimeError** – if the graph is empty

static from_file()
Constructs *Graph* instance from reading a Matrix Market file

Parameters **filename** (*str*) –

Returns the loaded *Graph*

Raises **RuntimeError** – if the file could not be read

static from_edgelist()
Constructs *Graph* instance from the given edge list. Note that vertex indices must start from 1.

Parameters

- **edgelist** (*numpy.ndarray*) – shape `(n,2)`
- **num_vertices** (*int*) –

Returns the loaded *Graph*

Raises

- **RuntimeError** – if any value in `edgelist` is greater than `num_vertices`
- **RuntimeError** – if value in `edgelist` is `0`

static from_matrix()
Constructs *Graph* instance from the given boolean adjacency matrix

Parameters **adjmat** (*numpy.ndarray*) – `bool` square matrix

Returns the loaded *Graph*

Raises **RuntimeError** – if `adjmat` is not square or the edges could not be constructed

static from_adjlist()
Constructs *Graph* instance from the given adjacency list. Note that the first element of the list must be an empty `set`, vertex indices start at 1.

Parameters

- **num_vertices** (*int*) –
- **num_edges** (*int*) –
- **edges** (*list*) – `list` of `set`s

Returns the loaded *Graph*

Raises **RuntimeError** – if first element in `edges` is nonempty, or there are invalid vertices/edges

to_file()
Exports *Graph* instance to a Matrix Market file

Parameters **filename** (*str*) –

Raises **RuntimeError** – if the file could not be opened, or if the graph is empty

to_edgelist()
Exports *Graph* instance to an edge list

Returns `(n,2)` `numpy.ndarray` of edges

Raises **RuntimeError** – if the graph is empty

to_matrix()
Exports *Graph* instance to a boolean matrix

Returns square `numpy.ndarray` of `bool`s

Raises **RuntimeError** – if the graph is empty

to_adjlist()
Exports *Graph* instance to an adjacency list

Returns list of sets

Raises **RuntimeError** – if the graph is empty

class cliquematch.**NWGraph**

search_done
Whether the search has been completed (Readonly)

Type bool

n_vertices
Number of vertices in the graph (Readonly)

Type int

n_edges
Number of edges in the graph (Readonly)

Type int

get_max_clique()
Finds a maximum clique in graph within the given bounds

Parameters

- **lower_bound** (*float*) – set a lower bound on the clique size. default is **1**.
- **upper_bound** (*float*) – set an upper bound on the clique size. default is **65535**.
- **use_heuristic** (*bool*) – if True, use the heuristic-based search to obtain a large clique quickly. Good for obtaining an initial lower bound. default is True.
- **use_dfs** (*bool*) – if True, use the depth-first to obtain the clique. default is True.

Returns the vertices in the maximum clique

Return type list

Raises **RuntimeError** – if the graph is empty or a clique could not be found

reset_search()
Reset the search space for *get_max_clique*.

Raises **RuntimeError** – if the graph is empty

all_cliques(*size*)
Iterate through all cliques of a given size in the *NWGraph*.

Parameters **size** (*float*) – size of a clique to search for.

Return type *NWCliqueIterator*

Raises **RuntimeError** – if the graph is empty

get_clique_weight(*clique*)
Return the weight of the provided clique.

Parameters **clique** (*list*) – a clique obtained via *get_max_clique*.

Return type float

static from_edgelist()
Constructs *NWGraph* instance from the given edge list. Note that vertex indices must start from 1.

Parameters

- **edgelist** (*numpy.ndarray*) – shape `(n, 2)`
- **num_vertices** (*int*) –

Returns the loaded *NWGraph*

Raises

- **RuntimeError** – if any value in `edgelist` is greater than `num_vertices`
- **RuntimeError** – if value in `edgelist` is `0`

static from_matrix()

Constructs *NWGraph* instance from the given boolean adjacency matrix

Parameters **adjmat** (*numpy.ndarray*) – `bool` square matrix

Returns the loaded *NWGraph*

Raises **RuntimeError** – if `adjmat` is not square or the edges could not be constructed

static from_adjlist()

Constructs *NWGraph* instance from the given adjacency list. Note that the first element of the list must be an empty `set`, vertex indices start at 1.

Parameters

- **num_vertices** (*int*) –
- **num_edges** (*int*) –
- **edges** (*list*) – `list` of `set`s

Returns the loaded *NWGraph*

Raises **RuntimeError** – if first element in `edges` is nonempty, or there are invalid vertices/edges

to_edgelist()

Exports *NWGraph* instance to an edge list

Returns `(n, 2)` `numpy.ndarray` of edges

Raises **RuntimeError** – if the graph is empty

to_matrix()

Exports *NWGraph* instance to a boolean matrix

Returns square `numpy.ndarray` of `bool`s

Raises **RuntimeError** – if the graph is empty

to_adjlist()

Exports *NWGraph* instance to an adjacency list

Returns `list` of `set`s

Raises **RuntimeError** – if the graph is empty

class cliquematch.**A2AGraph**(*set1*, *set2*, *d1=None*, *d2=None*, *is_d1_symmetric=True*, *is_d2_symmetric=True*)

Correspondence Graph wrapper for array-to-array mappings.

S1

array elements are converted to `numpy.float64`

Type `numpy.ndarray`

S2
array elements are converted to `numpy.float64`

Type `numpy.ndarray`

d1
distance metric for elements in *S1*, defaults to Euclidean metric if `None`.

Type `callable` (`numpy.ndarray`, `int`, `int`) -> `float`

d2
distance metric for elements in *S2*, defaults to Euclidean metric if `None`

Type `callable` (`numpy.ndarray`, `int`, `int`) -> `float`

is_d1_symmetric

Type `bool`

is_d2_symmetric

Type `bool`

all_correspondences(*size*, *return_indices=True*)
Find all correspondences of a given size.

Parameters

- **size** (`int`) – size of the corresponding subsets.
- **return_indices** (`bool`) – if `True` return the indices of the corresponding elements, else return the elements

Returns a wrapped *CorrespondenceIterator* object, which yields correspondences as per `return_indices`.

Return type *_WrappedIterator*

Raises **RuntimeError** – if called before edges have been constructed

build_edges()
Build edges of the correspondence graph using distance metrics.

Checks *d1* and *d2* for defaults before passing to base class.

Raises **RuntimeError** – if *d1*or *d2* are invalid functions

Returns `True` if construction was successful

Return type bool

build_edges_with_condition(*condition_func*, *use_cfunc_only*)
Build edges of the correspondence graph using a given condition function.

Parameters

- **condition_func** (`callable`) – must take parameters corresponding to *S1*, `int`, `int`, *S2*, `int`, `int`, and return `bool`
- **use_cfunc_only** (`bool`) – if `True`, the distance metrics will not be used to filter out edges (slower)

Returns `True` if construction was successful

Raises **RuntimeError** – if *d1*, *d2*, or `condition_func` are invalid functions

get_correspondence(*lower_bound=1*, *upper_bound=65535*, *time_limit=-1.0*, *use_heuristic=True*, *use_dfs=True*, *continue_search=False*, *return_indices=True*)

Get corresponding subsets between the *S1* and *S2*. Calls *get_max_clique* internally.

Parameters

- **lower_bound** (int) – set a lower bound for the size
- **upper_bound** (int) – set an upper bound for the size
- **time_limit** (float) – set a time limit for the search: a nonpositive value implies there is no time limit (use in conjunction with continue_search).
- **use_heuristic** (bool) – if True, use the heuristic-based search to obtain a large clique quickly. Good for obtaining an initial lower bound.
- **use_dfs** (bool) – if True, use the depth-first to obtain the clique. default is True.
- **continue_search** (bool) – set as True to continue a clique search interrupted by time_limit.
- **return_indices** (bool) – if True return the indices of the corresponding elements, else return the elements

Raises

- **RuntimeError** – if called before edges are constructed
- **RuntimeError** – if search parameters are invalid / clique is not found
- **RuntimeError** – if obtained correspondence is invalid

class cliquematch.**L2LGraph**(*set1*, *set2*, *d1=None*, *d2=None*, *is_d1_symmetric=True*, *is_d2_symmetric=True*)

Correspondence Graph wrapper for list-to-list mappings.

Any object can be passed for *S1* and *S2*; the user is required to define how the elements are accessed.

S1

array elements are converted to numpy.float64

Type object

S2

array elements are converted to numpy.float64

Type object

d1

distance metric for elements in *S1*

Type callable (list, int, int) -> float

d2

distance metric for elements in *S2*

Type callable (list, int, int) -> float

is_d1_symmetric

Type bool

is_d2_symmetric

Type bool

all_correspondences(*size*, *return_indices=True*)
Find all correspondences of a given size.

Parameters

- **size** (int) – size of the corresponding subsets.
- **return_indices** (bool) – if True return the indices of the corresponding elements, else return the elements

Returns a wrapped *CorrespondenceIterator* object, which yields correspondences as per return_indices.

Return type *_WrappedIterator*

Raises **RuntimeError** – if called before edges have been constructed

build_edges()
Build edges of the correspondence graph using distance metrics.

Checks *d1* and *d2* for defaults before passing to base class.

Raises **RuntimeError** – if *d1*or *d2* are invalid functions

Returns True if construction was successful

Return type bool

build_edges_with_condition(*condition_func*, *use_cfunc_only*)
Build edges of the correspondence graph using a given condition function.

Parameters

- **condition_func** (callable) – must take parameters corresponding to *S1*, int, int, *S2*, int, int, and return bool
- **use_cfunc_only** (bool) – if True, the distance metrics will not be used to filter out edges (slower)

Returns True if construction was successful

Raises **RuntimeError** – if *d1*, *d2*, or condition_func are invalid functions

get_correspondence(*lower_bound=1*, *upper_bound=65535*, *time_limit=-1.0*, *use_heuristic=True*, *use_dfs=True*, *continue_search=False*, *return_indices=True*)
Get corresponding subsets between the *S1* and *S2*. Calls *get_max_clique* internally.

Parameters

- **lower_bound** (int) – set a lower bound for the size
- **upper_bound** (int) – set an upper bound for the size
- **time_limit** (float) – set a time limit for the search: a nonpositive value implies there is no time limit (use in conjunction with continue_search).
- **use_heuristic** (bool) – if True, use the heuristic-based search to obtain a large clique quickly. Good for obtaining an initial lower bound.
- **use_dfs** (bool) – if True, use the depth-first to obtain the clique. default is True.
- **continue_search** (bool) – set as True to continue a clique search interrupted by time_limit.
- **return_indices** (bool) – if True return the indices of the corresponding elements, else return the elements

Raises

- **RuntimeError** – if called before edges are constructed
- **RuntimeError** – if search parameters are invalid / clique is not found
- **RuntimeError** – if obtained correspondence is invalid

class cliquematch.**L2AGraph**(*set1*, *set2*, *d1=None*, *d2=None*, *is_d1_symmetric=True*, *is_d2_symmetric=True*)

Correspondence Graph wrapper for list-to-array mappings.

Any general object can be passed for *S1*; the user is required to define how elements are accessed.

S1

Type object

S2

array elements are converted to numpy.float64

Type numpy.ndarray

d1

distance metric for elements in *S1*

Type callable (list, int, int) -> float

d2

distance metric for elements in *S2*, defaults to Euclidean metric if None

Type callable (numpy.ndarray, int, int) -> float

is_d1_symmetric

Type bool

is_d2_symmetric

Type bool

all_correspondences(*size*, *return_indices=True*)

Find all correspondences of a given size.

Parameters

- **size** (int) – size of the corresponding subsets.
- **return_indices** (bool) – if True return the indices of the corresponding elements, else return the elements

Returns a wrapped *CorrespondenceIterator* object, which yields correspondences as per return_indices.

Return type *_WrappedIterator*

Raises **RuntimeError** – if called before edges have been constructed

build_edges()

Build edges of the correspondence graph using distance metrics.

Checks *d1* and *d2* for defaults before passing to base class.

Raises **RuntimeError** – if *d1*or *d2* are invalid functions

Returns True if construction was successful

Return type bool

build_edges_with_condition(*condition_func*, *use_cfunc_only*)
Build edges of the correspondence graph using a given condition function.

Parameters

- **condition_func** (callable) – must take parameters corresponding to *S1*, int, int, *S2*, int, int, and return bool
- **use_cfunc_only** (bool) – if True, the distance metrics will not be used to filter out edges (slower)

Returns True if construction was successful

Raises **RuntimeError** – if *d1*, *d2*, or condition_func are invalid functions

get_correspondence(*lower_bound=1*, *upper_bound=65535*, *time_limit=-1.0*, *use_heuristic=True*, *use_dfs=True*, *continue_search=False*, *return_indices=True*)
Get corresponding subsets between the *S1* and *S2*. Calls *get_max_clique* internally.

Parameters

- **lower_bound** (int) – set a lower bound for the size
- **upper_bound** (int) – set an upper bound for the size
- **time_limit** (float) – set a time limit for the search: a nonpositive value implies there is no time limit (use in conjunction with continue_search).
- **use_heuristic** (bool) – if True, use the heuristic-based search to obtain a large clique quickly. Good for obtaining an initial lower bound.
- **use_dfs** (bool) – if True, use the depth-first to obtain the clique. default is True.
- **continue_search** (bool) – set as True to continue a clique search interrupted by time_limit.
- **return_indices** (bool) – if True return the indices of the corresponding elements, else return the elements

Raises

- **RuntimeError** – if called before edges are constructed
- **RuntimeError** – if search parameters are invalid / clique is not found
- **RuntimeError** – if obtained correspondence is invalid

class cliquematch.**A2LGraph**(*set1*, *set2*, *d1=None*, *d2=None*, *is_d1_symmetric=True*, *is_d2_symmetric=True*)
Correspondence Graph wrapper for array-to-list mappings.

Any general object can be passed for *S2*; the user is required to define how elements are accessed.

S1
array elements are converted to numpy.float64

Type numpy.ndarray

S2

Type object

d1
distance metric for elements in *S1*, defaults to Euclidean metric if None

Type callable (numpy.ndarray, int, int) -> float

d2
distance metric for elements in *S2*

Type callable (list, int, int) -> float

is_d1_symmetric

Type bool

is_d2_symmetric

Type bool

all_correspondences(*size*, *return_indices=True*)
Find all correspondences of a given size.

Parameters

- **size** (int) – size of the corresponding subsets.
- **return_indices** (bool) – if True return the indices of the corresponding elements, else return the elements

Returns a wrapped *CorrespondenceIterator* object, which yields correspondences as per return_indices.

Return type *_WrappedIterator*

Raises **RuntimeError** – if called before edges have been constructed

build_edges()
Build edges of the correspondence graph using distance metrics.

Checks *d1* and *d2* for defaults before passing to base class.

Raises **RuntimeError** – if *d1*or *d2* are invalid functions

Returns True if construction was successful

Return type bool

build_edges_with_condition(*condition_func*, *use_cfunc_only*)
Build edges of the correspondence graph using a given condition function.

Parameters

- **condition_func** (callable) – must take parameters corresponding to *S1*, int, int, *S2*, int, int, and return bool
- **use_cfunc_only** (bool) – if True, the distance metrics will not be used to filter out edges (slower)

Returns True if construction was successful

Raises **RuntimeError** – if *d1*, *d2*, or condition_func are invalid functions

get_correspondence(*lower_bound=1*, *upper_bound=65535*, *time_limit=-1.0*, *use_heuristic=True*, *use_dfs=True*, *continue_search=False*, *return_indices=True*)
Get corresponding subsets between the *S1* and *S2*. Calls *get_max_clique* internally.

Parameters

- **lower_bound** (int) – set a lower bound for the size
- **upper_bound** (int) – set an upper bound for the size

- **time_limit** (float) – set a time limit for the search: a nonpositive value implies there is no time limit (use in conjunction with continue_search).
- **use_heuristic** (bool) – if True, use the heuristic-based search to obtain a large clique quickly. Good for obtaining an initial lower bound.
- **use_dfs** (bool) – if True, use the depth-first to obtain the clique. default is True.
- **continue_search** (bool) – set as True to continue a clique search interrupted by time_limit.
- **return_indices** (bool) – if True return the indices of the corresponding elements, else return the elements

Raises

- **RuntimeError** – if called before edges are constructed
- **RuntimeError** – if search parameters are invalid / clique is not found
- **RuntimeError** – if obtained correspondence is invalid

class cliquematch.**IsoGraph**(*set1*, *set2*)

Correspondence graph for finding subgraph isomorphisms.

S1

Type *cliquematch.Graph*

S2

Type *cliquematch.Graph*

build_edges()

Build edges of the correspondence graph.

get_correspondence(*lower_bound=1*, *upper_bound=65535*, *time_limit=-1.0*, *use_heuristic=True*, *use_dfs=True*, *continue_search=False*, *return_indices=True*)

Obtain the corresponding vertices in the subgraph isomorphism.

Parameters

- **lower_bound** (int) – set a lower bound for the size
- **upper_bound** (int) – set an upper bound for the size
- **time_limit** (float) – set a time limit for the search: a nonpositive value implies there is no time limit (use in conjunction with continue_search).
- **use_heuristic** (bool) – if True, use the heuristic-based search to obtain a large clique quickly. Good for obtaining an initial lower bound.
- **use_dfs** (bool) – if True, use the depth-first to obtain the clique. default is True.
- **continue_search** (bool) – set as True to continue a clique search interrupted by time_limit.
- **return_indices** (bool) – if True return the vertices of the corresponding subgraphs, else return dicts for each corresponding subgraph and a dict mapping the vertices.

Raises

- **RuntimeError** – if called before edges are constructed
- **RuntimeError** – if search parameters are invalid / clique is not found

- **RuntimeError** – if obtained correspondence is invalid

all_correspondences(*size*, *return_indices=True*)
Find all correspondences of a given size.

Parameters

- **size** (int) – size of the corresponding subsets.
- **return_indices** (bool) – if True return the vertices of the corresponding subgraphs, else return dicts for each corresponding subgraph and a dict mapping the vertices.

Returns a wrapped *CorrespondenceIterator* object, which yields correspondences as per return_indices.

Return type *_WrappedIterator*

Raises **RuntimeError** – if called before edges are constructed

class cliquematch.**AlignGraph**(*set1*, *set2*)
Correspondence graph for aligning images using obtained interest points.

Uses a mask-based filtering method as a conditon function during construction of the graph. Default Euclidean metrics are used as distance metrics.

S1
array elements are converted to numpy.float64

Type numpy.ndarray

S2
array elements are converted to numpy.float64

Type numpy.ndarray

build_edges_with_filter(*control_points*, *filter_mask*, *percentage*)
Uses control points and a binary mask to filter out invalid mappings and construct a correspondence graph.

Parameters

- **control_points** (numpy.ndarray) – control points to use in every alignment test
- **filter_mask** (numpy.ndarray) – a boolean mask showing valid regions in the target image
- **percentage** (float) – an alignment is valid if the number of control points that fall within the mask are greater than this value

get_correspondence(*lower_bound=1*, *upper_bound=65535*, *time_limit=-1.0*, *use_heuristic=True*, *use_dfs=True*, *continue_search=False*, *return_indices=True*)
Find correspondence between the sets of points S1 and S2. Calls *get_max_clique* internally.

Parameters

- **lower_bound** (int) – set a lower bound for the size
- **upper_bound** (int) – set an upper bound for the size
- **time_limit** (float) – set a time limit for the search: a nonpositive value implies there is no time limit (use in conjunction with continue_search).
- **use_heuristic** (bool) – if True, use the heuristic-based search to obtain a large clique quickly. Good for obtaining an initial lower bound.
- **use_dfs** (bool) – if True, use the depth-first to obtain the clique. default is True.

- **continue_search** (bool) – set as True to continue a clique search interrupted by time_limit.
- **return_indices** (bool) – if True return the indices of the corresponding elements, else return the below dict

Returns The two sets of corresponding points and the rotation/translation required to transform *S1* to *S2* (obtained via Kabsch Algorithm)

Return type dict

Raises

- **RuntimeError** – if called before edges are constructed
- **RuntimeError** – if search parameters are invalid / clique is not found
- **RuntimeError** – if obtained correspondence is invalid

class cliquematch.**_WrappedIterator**
Wrapper for a *CorrespondenceIterator* object to provide results as per return_indices. Satisfies the iter/next protocol.

class cliquematch.core.**CliqueIterator**
A class satisfying the iter/next protocol to produces cliques of a given size from a *Graph*.

class cliquematch.core.**CorrespondenceIterator**
A class satisfying the iter/next protocol to produces correspondences of a given size from a *Graph*.

class cliquematch.core.**NWCliqueIterator**
A class satisfying the iter/next protocol to produces cliques of a given weight from a *NWGraph*.

1.4 How cliquematch works

cliquematch implements an exact algorithm to find maximum cliques, with a lot of pruning optimizations for large sparse graphs. The algorithm used is similar to FMC [1] and PMC [2], but also uses the concept of bitset-compression from BBMC [3] to save memory. It is implemented in C++11, does not use any additional libraries apart from the C++ STL, and is single-threaded.

1.4.1 Avoiding memory allocations

In earlier versions of *cliquematch*, the clique search methods were occasionally slowed due to many requests of small amounts from heap memory. In version 2.1, *cliquematch* does exactly *one* heap allocation – a std::vector large enough to hold a maximum clique – at the start of every search method, be it the heuristic search, finding one clique or enumerating cliques. During the clique search, it performs *zero* memory allocations. After a clique is found, it requests memory to return the clique to the user. Thus the number of heap allocations is mostly constant: a set number of allocations to initialize the Graph object (changes slightly depending on size of the graph), zero allocations during the search, and one allocation per clique returned.

This is possible because clique size always has an upper bound (degree, core-number, or the measure used in *cliquematch* all provide such a bound) and because bitsets are used for representing branches of the clique search. Thus *cliquematch* computes an upper bound for clique size (ie how deep a search can go), the maximum bitset space required for a vertex ($\lceil \frac{d_{max}}{64} \rceil$) and allocates space accordingly. The allocated space is reused throughout the search, thus avoiding any heap allocations during the most used part of the program.

1.4.2 Benchmarking the `cliquematch` algorithm

The below table shows how cliquematch compares to FMC and PMC.

name	V	E	w	t_{cm}	t_{fmc}	t_{pmc}	t_{cm_heur}	w_{cm_heur}	t_{fmc_heur}	w_{fmc_heur}
Erdos02	6927	8472	7	**0.0001**	0.0015	0.0022	0.0001	6	0.0009	7
Erdos972	5488	7085	7	**0.0001**	0.0004	0.0016	0.0001	7	0.0002	6
Erdos982	5822	7375	7	**0.0001**	0.0004	0.0018	0.0	7	0.0002	7
Erdos992	6100	7515	8	**0.0001**	0.0004	0.0018	0.0	8	0.0002	8
Fault_639	638802	14626683	18	**2.0622**	13.8244	•	0.5864	18	2.4625	18
brock200_2	200	9876	12	0.217	0.6421	**0.0016**	0.0019	10	0.0023	9
c-fat200-5	200	8473	58	0.0023	0.4169	**0.0003**	0.0001	58	0.0102	58
ca-AstroPh	18772	198110	57	**0.0004**	0.0789	0.0132	0.0003	56	0.0277	57
ca-CondMat	23133	93497	26	**0.0003**	0.0048	0.0083	0.0002	26	0.0042	26
ca-GrQc	5242	14496	44	**0.0001**	0.0005	0.0017	0.0001	43	0.0012	44
ca-HepPh	12008	118521	239	**0.0034**	0.0131	0.0149	0.0012	239	0.2518	239
ca-HepTh	9877	25998	32	**0.0001**	0.0007	0.0035	0.0	32	0.0001	32
caidaRouterLevel	192244	609066	17	**0.0257**	0.2098	0.0774	0.0084	17	0.073	15
coPapersCitesseer	434102	16036720	845	**0.0248**	0.8344	1.4019	0.0144	845	14.6734	845
com-Youtube	1134890	2987624	17	**1.1393**	9.618	•	0.1412	16	0.3515	13
cond-mat-2003	31163	120029	25	**0.0007**	0.0118	0.0096	0.0004	25	0.0038	25
cti	16840	48232	3	**0.0014**	0.0036	0.0046	0.0009	3	0.0013	3
hamming6-4	64	704	4	**0.0002**	0.0003	8.5115	0.0	4	5.6982	4
johnson8-4-4	70	1855	14	0.0405	0.1459	**0.0003**	0.0001	11	0.0005	14
keller4	171	9435	11	**3.4486**	15.4211	•	0.0016	9	0.0027	9
loc-Brightkite	58228	214078	37	5.0651	2.7146	**0.0277**	0.0038	36	0.0151	31

- `fmc -t 0 -p` was used to run the FMC branch-and-bound algorithm.
- `pmc -t 1 -r 1 -a 0 -h 0 -d` (single CPU thread, reduce wait time of 1 second, full algorithm, skip heuristic, search in descending order) was used to run the PMC branch-and-bound algorithm.
- `fmc -t 1` was used to run the FMC heuristic algorithm.
- A small script was used to run the *`cliquematch`* algorithm. *`cliquematch`* was compiled with `BENCHMARKING=1`.

The benchmark graphs were taken from the UFSparse [4], SNAP [5], and DIMACS [6] collections. $|V|$ and $|E|$ denote the number of nodes and edges in the graph. ω denotes the size of the maximum clique found. All the

branch-and-bound methods agreed on the maximum clique size in every benchmark. t_{cm}, t_{fmc}, and t_{pmc} denote the time taken by `cliquematch`, FMC, and PMC respectively in the branch-and-bound search: the least time is in bold text. $w_{cm-heur}$, $t_{cm-heur}$ denote the size of clique and time taken by the heuristic method in `cliquematch`; and similarly for $w_{fmc-heur}$ and $t_{fmc-heur}$. A minus sign (−) indicates that the program returned an error without completing the calculation.

1.4.3 Correspondence Graphs in C++

The correspondence graph classes are generated using simple C++ template programming (simple because it's basically a typed macro substitution). `cliquematch` can be extended with custom correspondence graphs: they can be prototyped using the existing classes, and/or implemented in C++ for performance. The source code of `IsoGraph` or `AlignGraph` are good places to start if you're trying to implement a custom correspondence graph class.

References

BIBLIOGRAPHY

[1] Bharath Pattabiraman, Md Mostofa Ali Patwary, Assefaw H Gebremedhin, Wei-keng Liao, and Alok Choudhary. Fast algorithms for the maximum clique problem on massive graphs with applications to overlapping community detection. *Internet Mathematics*, 11(4-5):421–448, 2015.

[2] Ryan A Rossi, David F Gleich, and Assefaw H Gebremedhin. Parallel maximum clique algorithms with applications to network analysis. *SIAM Journal on Scientific Computing*, 37(5):C589–C616, 2015.

[3] Pablo San Segundo, Diego Rodríguez-Losada, and Agustín Jiménez. An exact bit-parallel algorithm for the maximum clique problem. *Computers & Operations Research*, 38(2):571–581, 2011.

[4] Timothy A Davis and Yifan Hu. The university of florida sparse matrix collection. *ACM Transactions on Mathematical Software (TOMS)*, 38(1):1–25, 2011.

[5] Jure Leskovec and Andrej Krevl. SNAP Datasets: Stanford large network dataset collection. https://snap.stanford.edu/data, June 2014.

[6] David S Johnson and Michael A Trick. *Cliques, coloring, and satisfiability: second DIMACS implementation challenge, October 11-13, 1993*. Volume 26. American Mathematical Soc., 1996.

PYTHON MODULE INDEX

C

INDEX

Symbols

A

B

C

D

F

G

I

L

M

N

R

S

T