TMAP

Release 0.2.0

Jul 03, 2019

CONTENTS

CHAPTER

ONE

TABLE OF CONTENTS

1.1 Getting Started

1.1.1 Installation

TMAP can be installed using the conda package manager that is distributed with miniconda (or anaconda).

```
conda install tmap
```

The module is then best imported using a shorter identifier.

```
import tmap as tm
```

1.1.2 Laying out a Simple Graph

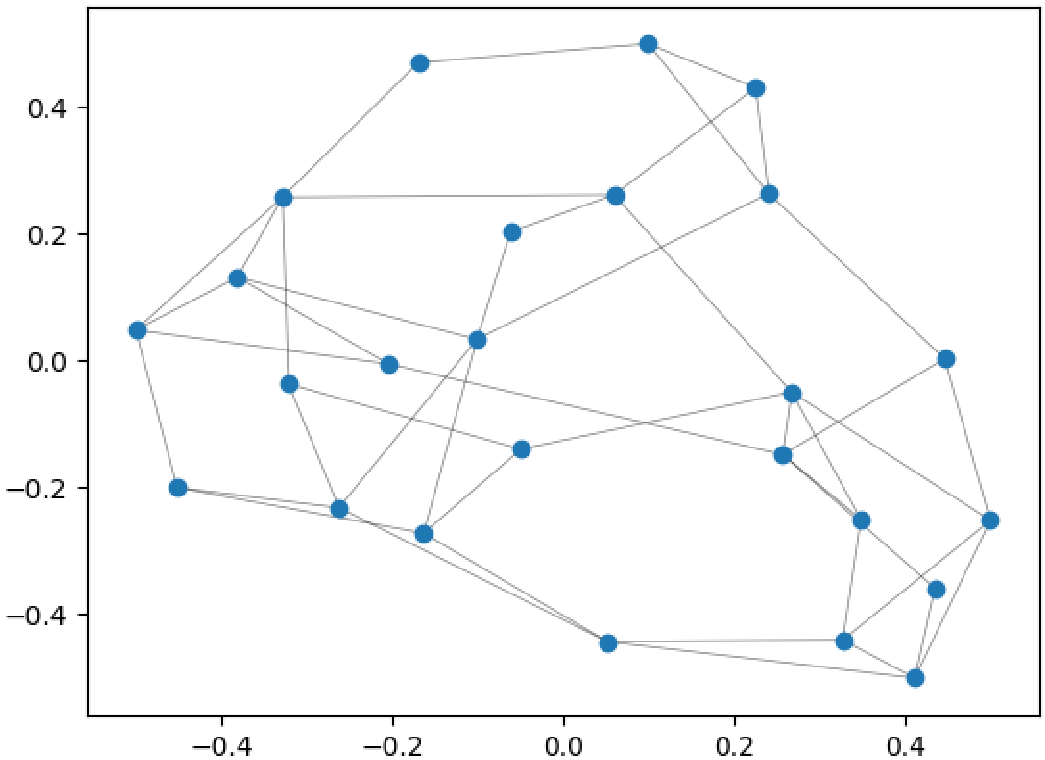
Even though TMAP is mainly targeted at tasks consisting of laying out very large data sets, the simplest usage example is laying out a graph.

```
import tmap as tm
import numpy as np
from matplotlib import pyplot as plt

n = 25
edge_list = []

# Create a random graph
for i in range(n):
    for j in np.random.randint(0, high=n, size=2):
        edge_list.append([i, j, np.random.rand(1)])

# Set the initial randomized positioning to True
# Otherwise, OGDF tends to segfault
cfg = tm.LayoutConfiguration()
cfg.fme_randomize = True

# Compute the layout
x, y, s, t, gp = tm.layout_from_edge_list(n, edge_list, config=cfg,
                                          create_mst=False)

# Plot the edges
for i in range(len(s)):
```

(continues on next page)

(continued from previous page)

```
    plt.plot([x[s[i]], x[t[i]]], [y[s[i]], y[t[i]]], 'k-',
             linewidth=0.5, alpha=0.5, zorder=1)

# Plot the vertices
plt.scatter(x, y, zorder=2)

plt.savefig('simple_graph.png')
```



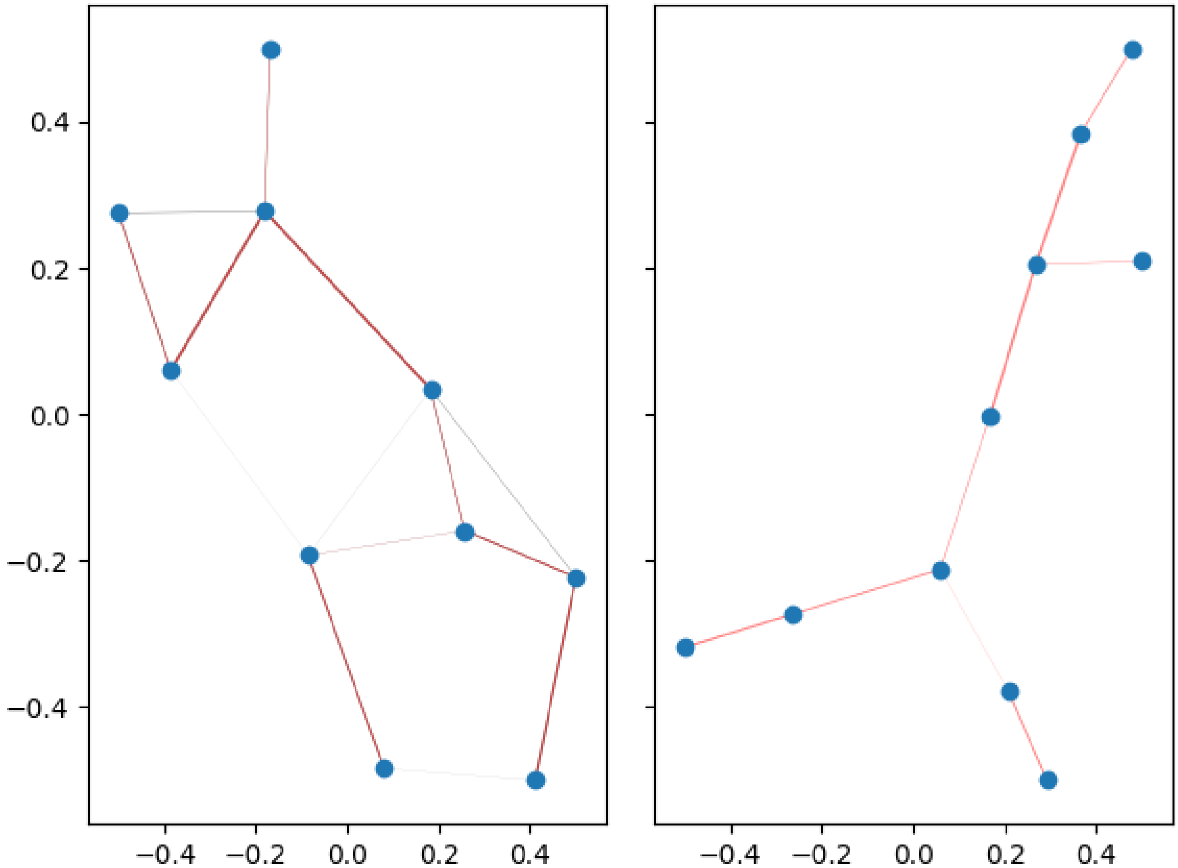
When laying out large graphs, it might be useful to discard some edges in order to create a more interpretable and visually pleasing layout. This is achieved using the (default) argument `create_mst=True`. Following, this is exemplified on a small graph.

```
n = 10
edge_list = []
weights = {}

# Create a random graph
for i in range(n):
    for j in np.random.randint(0, high=n, size=2):
        # Do not add parallel edges here, to be sure
        # to have the right weight later
        if i in weights and j in weights[i] or j in weights and i in weights[j]:
            continue
```

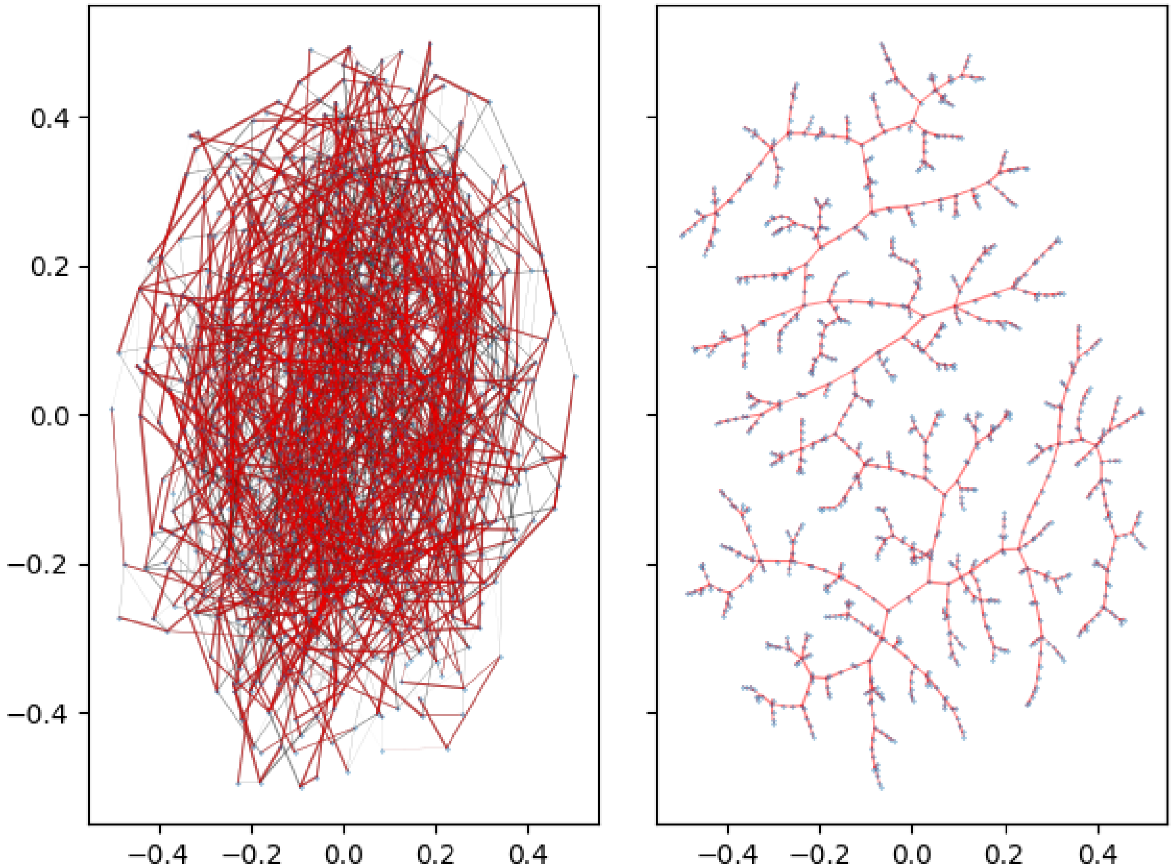
(continues on next page)

(continued from previous page)

```
        weight = np.random.rand(1)
        edge_list.append([i, j, weight])

        # Store the weights in 2d map for easy access
        if i not in weights:
            weights[i] = {}
        if j not in weights:
            weights[j] = {}

        # Invert weights to make lower ones more visible in the plot
        weights[i][j] = 1.0 - weight
        weights[j][i] = 1.0 - weight


# Set the initial randomized positioning to True
# Otherwise, OGDF tends to segfault
cfg = tm.LayoutConfiguration()
cfg.fme_randomize = True

# Compute the layout
x, y, s, t, _ = tm.layout_from_edge_list(n, edge_list, config=cfg,
                                         create_mst=False)
x_mst, y_mst, s_mst, t_mst, _ = tm.layout_from_edge_list(n, edge_list,
                                                         create_mst=True)

_, (ax1, ax2) = plt.subplots(ncols=2, sharey=True)

# Plot graph layout with spanning tree superimposed in red
for i in range(len(s)):
    ax1.plot([x[s[i]], x[t[i]]], [y[s[i]], y[t[i]]], 'k-',
             linewidth=weights[s[i]][t[i]], alpha=0.5, zorder=1)

for i in range(len(s_mst)):
    ax1.plot([x[s_mst[i]], x[t_mst[i]]], [y[s_mst[i]], y[t_mst[i]]], 'r-',
             linewidth=weights[s_mst[i]][t_mst[i]], alpha=0.5, zorder=2)

ax1.scatter(x, y, zorder=3)


# Plot spanning tree layout
for i in range(len(s_mst)):
    ax2.plot([x_mst[s_mst[i]], x_mst[t_mst[i]]], [y_mst[s_mst[i]], y_mst[t_mst[i]]],
↪'r-',
             linewidth=weights[s_mst[i]][t_mst[i]], alpha=0.5, zorder=1)

ax2.scatter(x_mst, y_mst, zorder=2)

plt.tight_layout()
plt.savefig('spanning_tree.png')
```



On a highly connected graph with 1000 vertices, the advantages of the tree visualizaton method applied by TMAP become obvious.



There are a wide array of options to tune the final tree layout to your linking. See *Layout* for the descriptions of all available parameters.

1.1.3 MinHash

In order to enable the visualization of larger data sets, it is necessary to speed up the k-nearest neighbor graph generation. While in general, any approach can be used to create this nearest neighbor graph (see Laying out a Simple Graph), TMAP provides a built-in LSH Forest data structure, which enables extremely fast k-nearest neighbor queries.

In order to index data in the LSH forest data structure, it has to be hashed using a locality sensitive scheme such as MinHash.

TMAP includes the two classes `MinHash` and `LSHForest` for fast k-nearest neighbor search.

The following example shows how to use the `MinHash` class to estimate Jaccard distances.

```
import tmap as tm

enc = tm.Minhash()

mh_a = enc.from_binary_array(tm.VectorUchar([1, 1, 1, 1, 0, 1, 0, 1, 1, 0]))
mh_b = enc.from_binary_array(tm.VectorUchar([1, 0, 1, 1, 0, 1, 1, 0, 1, 0]))
mh_c = enc.from_binary_array(tm.VectorUchar([1, 0, 1, 1, 1, 1, 1, 0, 1, 0]))

dist_a_b = enc.get_distance(mh_a, mh_b)
```

(continues on next page)

(continued from previous page)

```
dist_b_c = enc.get_distance(mh_b, mh_c)

print(dist_a_b)
print(dist_b_c)
```

```
>>> 0.390625
>>> 0.140625
```

An in-depth explanation of MinHash can be found in this video by Jeffry D Ullman.

`Minhash` also supports encoding strings, indexed binary arrays, and `int` and `float` weighted arrays. See *MinHash* for details.

1.1.4 LSH Forest

The hashes generated by `Minhash` can be indexed using `LSHForest` for fast k-nearest neighbor retreival.

```
from timeit import default_timer as timer

import numpy as np
import tmap as tm

# Use 128 permutations to create the MinHash
enc = tm.Minhash(128)
lf = tm.LSHForest(128)

d = 1000
n = 1000000

data = []

# Generating some random data
start = timer()
for i in range(n):
    data.append(tm.VectorUchar(np.random.randint(0, high=2, size=d)))
print(f'Generating the data took {(timer() - start) * 1000}ms.')

# Use batch_add to parallelize the insertion of the arrays
start = timer()
lf.batch_add(enc.batch_from_binary_array(data))
print(f'Adding the data took {(timer() - start) * 1000}ms.')

# Index the added data
start = timer()
lf.index()
print(f'Indexing took {(timer() - start) * 1000}ms.')

# Find the 10 nearest neighbors of the first entry
start = timer()
result = lf.query_linear_scan_by_id(0, 10)
print(f'The kNN search took {(timer() - start) * 1000}ms.')
```

```
>>> Generating the data took 118498.04133399994ms.
>>> Adding the data took 55051.067827000224ms.
```

(continues on next page)

(continued from previous page)

```
>>> Indexing took 2059.1810410005564ms.
>>> The kNN search took 0.32151699997484684ms.
```

After indexing the data, the 10 nearest neighbor search on a million 1,000-dimensional vectors took ~0.32ms. In addition, the `LSHForest` class also supports the parallelized generation of a k-nearest neighbor graph using the method `get_knn_graph()`.

```
# ...

# Construct the k-nearest neighbour graph
start = timer()
knng_from = tm.VectorUint()
knng_to = tm.VectorUint()
knng_weight = tm.VectorFloat()

result = lf.get_knn_graph(knng_from, knng_to, knng_weight, 10)
print(f'The kNN search took {(timer() - start) * 1000}ms.')
```

```
>>> The kNN search took 37519.07863999986ms.
```

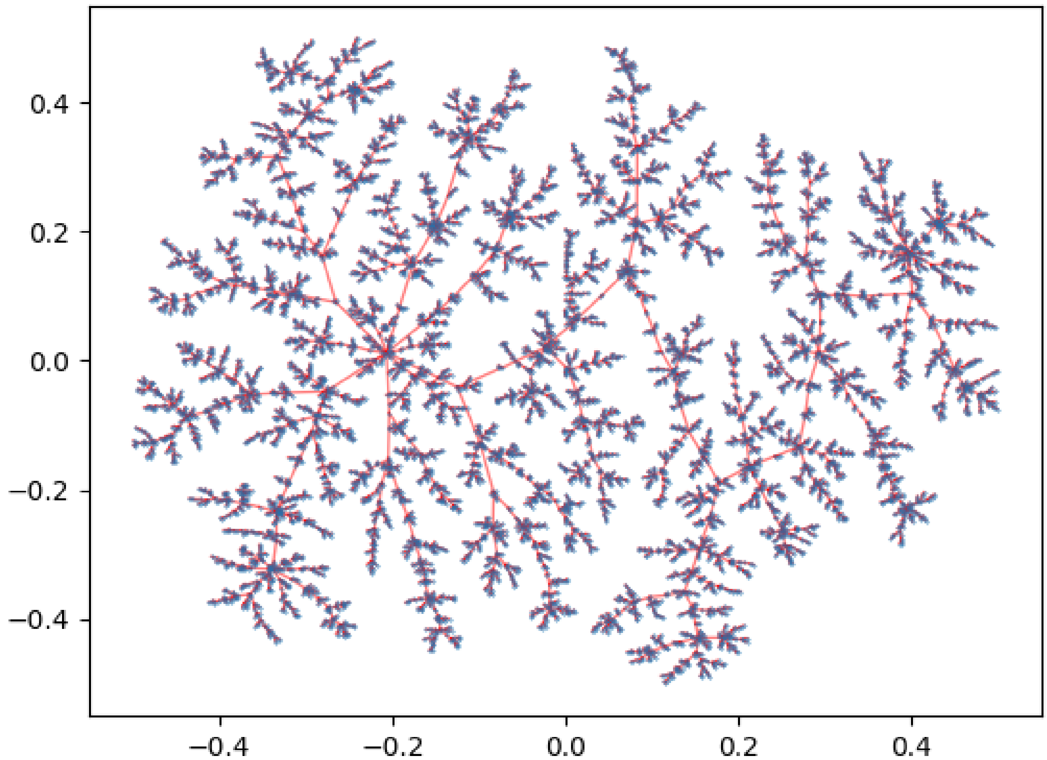
1.1.5 Layout

TMAP ships with the function `layout_from_lsh_forest()` which creates a graph / tree layout directly from an `LSHForest` instance.

The resulting layout can then be plotted using matplotlib / pyplot using its `plot()` and `scatter` methods.

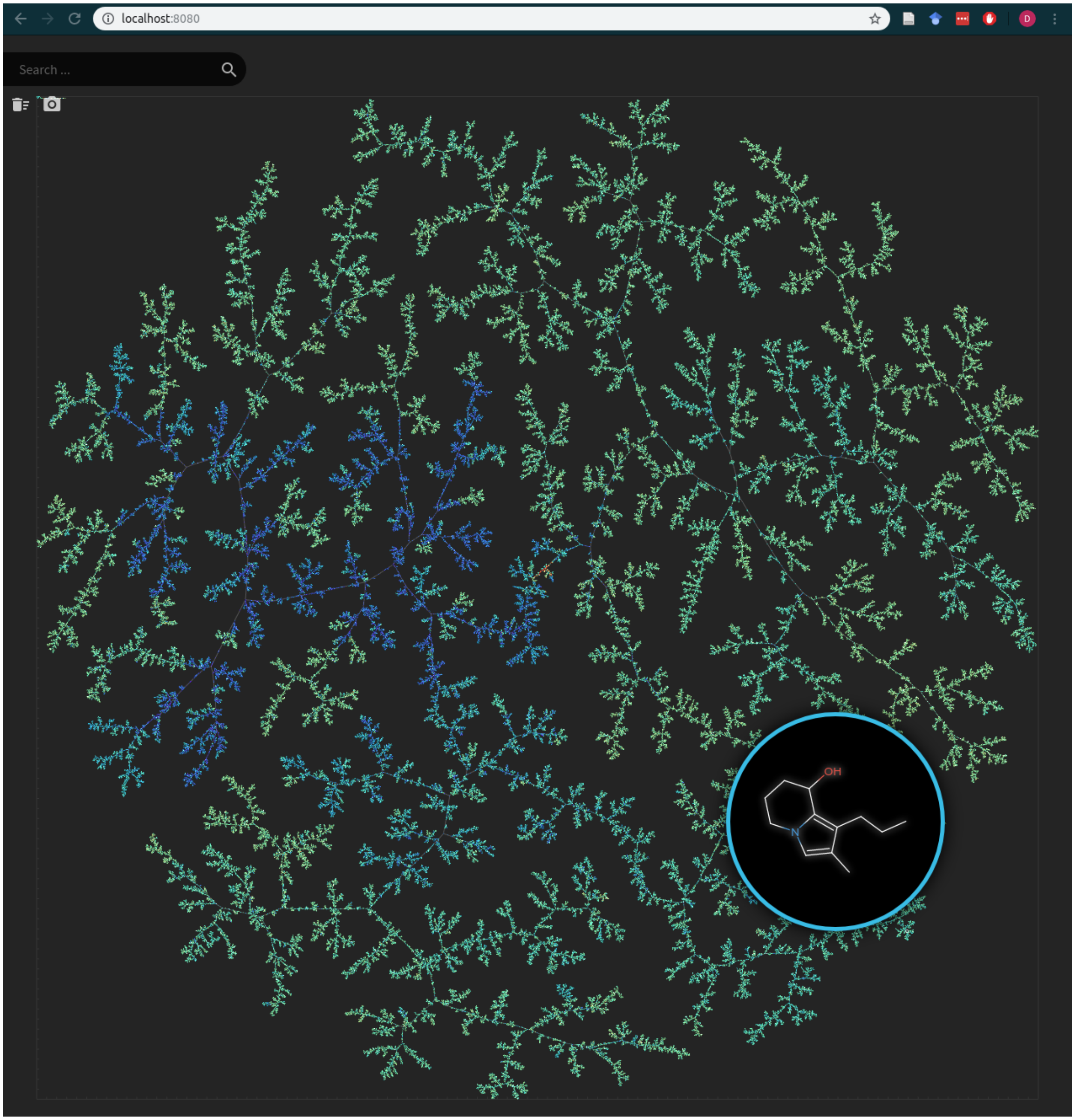
```
# ...

# The configuration for the MST plot
# Distribute the tree more evenly
cfg = tm.LayoutConfiguration()
cfg.sl_scaling_min = 1
cfg.sl_scaling_max = 1
cfg.node_size = 1 / 50

# Construct the k-nearest neighbour graph
start = timer()
x, y, s, t, _ = tm.layout_from_lsh_forest(lf, config=cfg)
print(f'layout_from_lsh_forest took {(timer() - start) * 1000}ms.')

# Plot spanning tree layout
start = timer()
for i in range(len(s)):
    plt.plot([x[s[i]], x[t[i]]], [y[s[i]], y[t[i]]], 'r-',
             linewidth=1.0, alpha=0.5, zorder=1)

plt.scatter(x, y, s=0.1, zorder=2)

plt.savefig('lsh_forest_knng_mpl.png')
print(f'Plotting using matplotlib took {(timer() - start) * 1000}ms.')
```

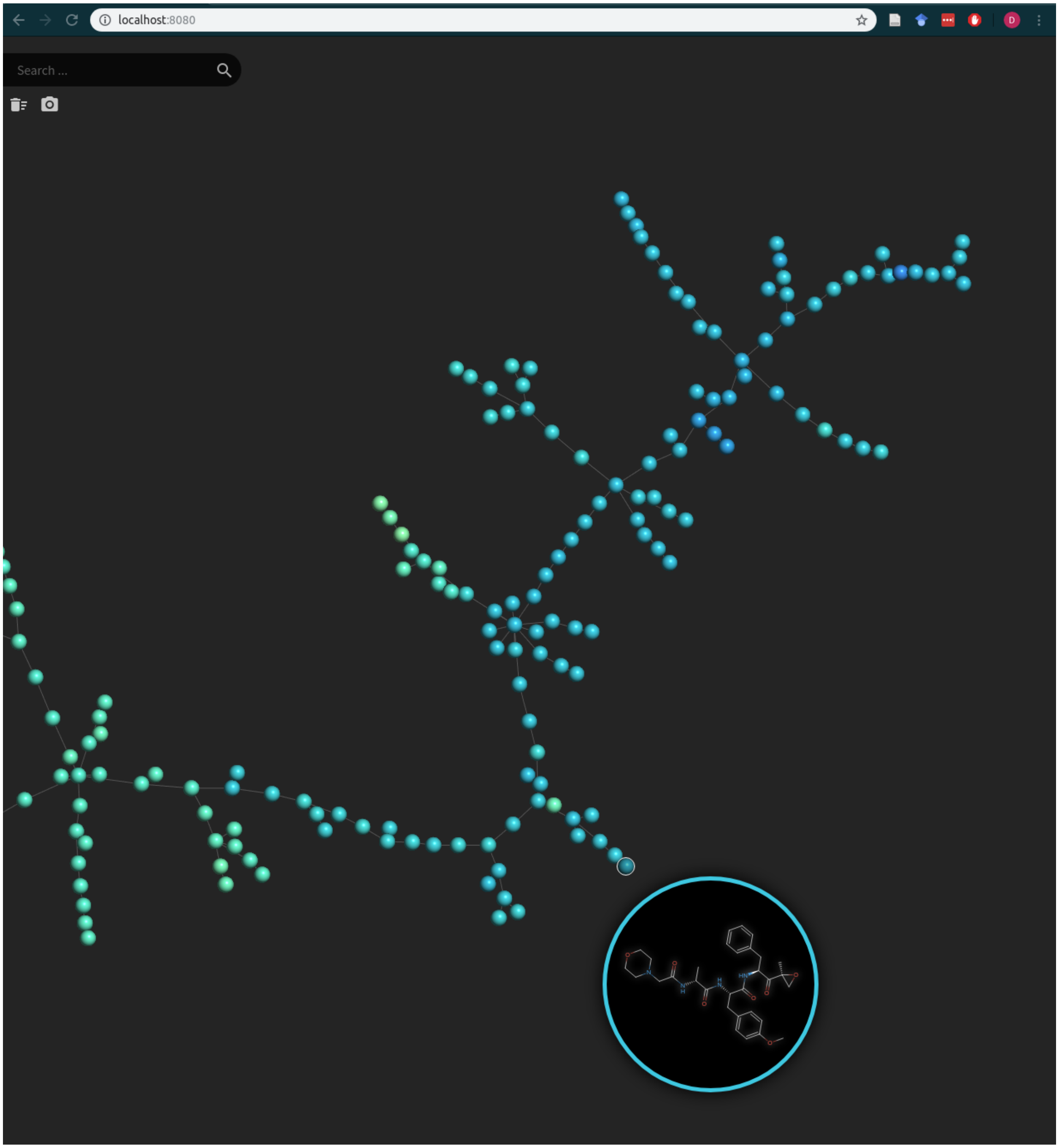
```
>>> layout_from_lsh_forest took 1218.4765429992694ms.
>>> Plotting using matplotlib took 35739.334431000316ms.
```



Using matplotlib / pyplot has tow main disadvantages: It is slow and does not yield interactive plots. For this reason, we suggest to use the Python package Faerun for large scale data sets. Faerun supports millions of data points in web-based visualizations.

Together with TMAP, Faerun can easily create visualizations of more than 10 million data points including associated web links and structure drawings for high dimensional chemical data sets within an hour.


localhost:8080
Search ...
OH
N



1.2 Documentation

1.2.1 MinHash

class tmap.**Minhash**(*self: tmap.Minhash, d: int=128, seed: int=42, sample_size: int=128*) → None

A generator for MinHash vectors that supports binary, indexed, string and also `int` and `float` weighted vectors as input.

Constructor for the class *Minhash*.

Keyword Arguments

- **d** (int) – The number of permutations used for hashing
- **seed** (int) – The seed used for the random number generator(s)
- **sample_size** (int) – The sample size when generating a weighted MinHash

__init__(*self: tmap.Minhash*, *d: int=128*, *seed: int=42*, *sample_size: int=128*) → None
Constructor for the class *Minhash*.

Keyword Arguments

- **d** (int) – The number of permutations used for hashing
- **seed** (int) – The seed used for the random number generator(s)
- **sample_size** (int) – The sample size when generating a weighted MinHash

batch_from_binary_array(*self: tmap.Minhash, arg0: List[tmap.VectorUchar]*) → List[tmap.VectorUint]
Create MinHash vectors from binary arrays (parallelized).

Parameters **vec** (List of VectorUchar) – A list of vectors containing binary values

Returns A list of MinHash vectors

Return type List of VectorUint

batch_from_int_weight_array(*self: tmap.Minhash, arg0: List[tmap.VectorUint]*) → List[tmap.VectorUint]
Create MinHash vectors from int arrays, where entries are weights rather than indices of ones (parallelized).

Parameters **vec** (List of VectorUint) – A list of vectors containing int values

Returns A list of MinHash vectors

Return type List of VectorUint

batch_from_sparse_binary_array(*self: tmap.Minhash, arg0: List[tmap.VectorUint]*) → List[tmap.VectorUint]
Create MinHash vectors from sparse binary arrays (parallelized).

Parameters **vec** (List of VectorUint) – A list of vectors containing indices of ones in a binary array

Returns A list of MinHash vectors

Return type List of VectorUint

batch_from_string_array(*self: tmap.Minhash, arg0: List[List[str]]*) → List[tmap.VectorUint]
Create MinHash vectors from string arrays (parallelized).

Parameters **vec** (List of List of str) – A list of list of strings

Returns A list of MinHash vectors

Return type List of VectorUint

batch_from_weight_array(*self: tmap.Minhash, arg0: List[tmap.VectorFloat]*) → List[tmap.VectorUint]
Create MinHash vectors from float arrays (parallelized).

Parameters **vec** (List of VectorFloat) – A list of vectors containing float values

Returns A list of MinHash vectors

Return type List of VectorUint

from_binary_array(*self: tmap.Minhash*, *arg0: tmap.VectorUchar*) → tmap.VectorUint

Create a MinHash vector from a binary array.

Parameters **vec** (`VectorUchar`) – A vector containing binary values

Returns A MinHash vector

Return type `VectorUint`

from_sparse_binary_array(*self: tmap.Minhash*, *arg0: tmap.VectorUint*) → tmap.VectorUint

Create a MinHash vector from a sparse binary array.

Parameters **vec** (`VectorUint`) – A vector containing indices of ones in a binary array

Returns A MinHash vector

Return type `VectorUint`

from_string_array(*self: tmap.Minhash, arg0: List[str]*) → tmap.VectorUint

Create a MinHash vector from a string array.

Parameters **vec** (`List` of `str`) – A vector containing strings

Returns A MinHash vector

Return type `VectorUint`

from_weight_array(*self: tmap.Minhash*, *arg0: tmap.VectorFloat*) → tmap.VectorUint

Create a MinHash vector from a `float` array.

Parameters **vec** (`VectorFloat`) – A vector containing `float` values

Returns A MinHash vector

Return type `VectorUint`

get_distance(*self: tmap.Minhash*, *arg0: tmap.VectorUint*, *arg1: tmap.VectorUint*) → float

Calculate the Jaccard distance between two MinHash vectors.

Parameters

- **vec_a** (`VectorUint`) – A MinHash vector
- **vec_b** (`VectorUint`) – A MinHash vector

Returns `float` The Jaccard distance

get_weighted_distance(*self: tmap.Minhash*, *arg0: tmap.VectorUint*, *arg1: tmap.VectorUint*) → float

Calculate the weighted Jaccard distance between two MinHash vectors.

Parameters

- **vec_a** (`VectorUint`) – A weighted MinHash vector
- **vec_b** (`VectorUint`) – A weighted MinHash vector

Returns `float` The Jaccard distance

1.2.2 LSH Forest

class tmap.**LSHForest**(*self: tmap.LSHForest*, *d: int=128*, *l: int=8*, *store: bool=True*, *file_backed: bool=False*) → None

A LSH forest data structure which incorporates optional linear scan to increase the recovery performance. Most query methods are available in parallelized versions named with a `batch_` prefix.

Constructor for the class *`LSHForest`*.

Keyword Arguments

- **d** (int) – The dimensionality of the MinHashe vectors to be added
- **l** (int) – The number of prefix trees used when indexing data
- **store** (bool) –
- **file_backed** (bool) Whether to store the data on disk rather than in main memory (experimental) –

__init__(*self: tmap.LSHForest*, *d: int=128*, *l: int=8*, *store: bool=True*, *file_backed: bool=False*) → None
Constructor for the class *LSHForest*.

Keyword Arguments

- **d** (int) – The dimensionality of the MinHashe vectors to be added
- **l** (int) – The number of prefix trees used when indexing data
- **store** (bool) –
- **file_backed** (bool) Whether to store the data on disk rather than in main memory (experimental) –

add(*self: tmap.LSHForest*, *arg0: tmap.VectorUint*) → None
Add a MinHash vector to the LSH forest.

Parameters **vecs** (VectorUint) – A MinHash vector that is to be added to the LSH forest

batch_add(*self: tmap.LSHForest, arg0: List[tmap.VectorUint]*) → None
Add a list MinHash vectors to the LSH forest (parallelized).

Parameters **vecs** (List of VectorUint) – A list of MinHash vectors that is to be added to the LSH forest

batch_query(*self: tmap.LSHForest, arg0: List[tmap.VectorUint], arg1: int*) → List[tmap.VectorUint]
Query the LSH forest for k-nearest neighbors (parallelized).

Parameters

- **vecs** (List of VectorUint) – The query MinHash vectors
- **k** (int) – The number of nearest neighbors to be retrieved

Returns The results of the queries

Return type List of VectorUint

clear(*self: tmap.LSHForest*) → None
Clears all the added data and computed indices from this *LSHForest* instance.

get_all_distances(*self: tmap.LSHForest*, *arg0: tmap.VectorUint*) → tmap.VectorFloat
Calculate the Jaccard distances of a MinHash vector to all indexed MinHash vectors.

Parameters **vec** (VectorUint) – The query MinHash vector

Returns The Jaccard distances

Return type List of float

get_all_nearest_neighbors(*self: tmap.LSHForest*, *k: int*, *kc: int=10*, *weighted: bool=False*) → tmap.VectorUint
Get the k-nearest neighbors of all indexed MinHash vectors.

Parameters **k** (int) – The number of nearest neighbors to be retrieved

Keyword Arguments

- **kc** (int) – The factor by which k is multiplied for LSH forest retreival
- **weighted** (bool) – Whether the MinHash vectors in this *LSHForest* instance are weighted

Returns VectorUint The ids of all k-nearest neighbors

get_distance (*self: tmap.LSHForest*, *arg0: tmap.VectorUint*, *arg1: tmap.VectorUint*) → float
Calculate the Jaccard distance between two MinHash vectors.

Parameters

- **vec_a** (VectorUint) – A MinHash vector
- **vec_b** (VectorUint) – A MinHash vector

Returns float The Jaccard distance

get_distance_by_id (*self: tmap.LSHForest*, *arg0: int*, *arg1: int*) → float
Calculate the Jaccard distance between two indexed MinHash vectors.

Parameters

- **a** (int) – The id of an indexed MinHash vector
- **b** (int) – The id of an indexed MinHash vector

Returns float The Jaccard distance

get_hash (*self: tmap.LSHForest*, *arg0: int*) → tmap.VectorUint
Retrieve the MinHash vector of an indexed entry given its index. The index is defined by order of insertion.

Parameters **a** (int) – The id of an indexed MinHash vector

Returns VectorUint The MinHash vector

get_knn_graph (*self: tmap.LSHForest*, *from: tmap.VectorUint*, *to: tmap.VectorUint*, *weight: tmap.VectorFloat*, *k: int*, *kc: int=10*, *weighted: bool=False*) → None
Construct the k-nearest neighbor graph of the indexed MinHash vectors. It will be written to out parameters from, to, and weight as an edge list.

Parameters

- **from** (VectorUint) – A vector to which the ids for the from vertices are written
- **to** (VectorUint) – A vector to which the ids for the to vertices are written
- **weight** (VectorFloat) – A vector to which the edge weights are written
- **k** (int) – The number of nearest neighbors to be retrieved during the construction of the k-nearest neighbor graph

Keyword Arguments

- **kc** (int) – The factor by which k is multiplied for LSH forest retreival
- **weighted** (bool) – Whether the MinHash vectors in this *LSHForest* instance are weighted

get_weighted_distance (*self: tmap.LSHForest*, *arg0: tmap.VectorUint*, *arg1: tmap.VectorUint*) → float
Calculate the weighted Jaccard distance between two MinHash vectors.

Parameters

- **vec_a** (VectorUint) – A weighted MinHash vector

- **vec_b** (`VectorUint`) – A weighted MinHash vector

Returns `float` The Jaccard distance

get_weighted_distance_by_id(*self: tmap.LSHForest*, *arg0: int*, *arg1: int*) → float
Calculate the Jaccard distance between two indexed weighted MinHash vectors.

Parameters

- **a** (`int`) – The id of an indexed weighted MinHash vector
- **b** (`int`) – The id of an indexed weighted MinHash vector

Returns `float` The weighted Jaccard distance

index(*self: tmap.LSHForest*) → None
Index the LSH forest. This has to be run after each time new MinHashes were added.

is_clean(*self: tmap.LSHForest*) → bool
Returns a boolean indicating whether or not the LSH forest has been indexed after the last MinHash vector was added.

Returns `True` if *`index()`* has been run since MinHash vectors have last been added using *`add()`* or *`batch_add()`*. `False` otherwise

Return type `bool`

linear_scan(*self: tmap.LSHForest*, *vec: tmap.VectorUint*, *indices: tmap.VectorUint*, *k: int=10*, *weighted: bool=False*) → List[Tuple[float, int]]
Query a subset of indexed MinHash vectors using linear scan.

Parameters

- **vec** (`VectorUint`) – The query MinHash vector
- **indices** (`VectorUint`) –

Keyword Arguments

- **k** (`int`) – The number of nearest neighbors to be retrieved
- **weighted** (`bool`) – Whether the MinHash vectors in this *`LSHForest`* instance are weighted

Returns The results of the query

Return type `List` of `Tuple[float, int]`

query(*self: tmap.LSHForest*, *arg0: tmap.VectorUint*, *arg1: int*) → tmap.VectorUint
Query the LSH forest for k-nearest neighbors.

Parameters

- **vec** (`VectorUint`) – The query MinHash vector
- **k** (`int`) – The number of nearest neighbors to be retrieved

Returns The results of the query

Return type `VectorUint`

query_by_id(*self: tmap.LSHForest*, *arg0: int*, *arg1: int*) → tmap.VectorUint
Query the LSH forest for k-nearest neighbors.

Parameters

- **id** (`int`) – The id of an indexed MinHash vector
- **k** (`int`) – The number of nearest neighbors to be retrieved

Returns The results of the query

Return type `VectorUint`

query_exclude (*self: tmap.LSHForest*, *arg0: tmap.VectorUint*, *arg1: tmap.VectorUint*, *arg2: int*) → tmap.VectorUint

Query the LSH forest for k-nearest neighbors.

Parameters

- **vec** (`VectorUint`) – The query MinHash vector
- **exclude** (`List` of `VectorUint`) –
- **k** (`int`) – The number of nearest neighbors to be retrieved

Returns The results of the query

Return type `VectorUint`

query_exclude_by_id (*self: tmap.LSHForest*, *arg0: int*, *arg1: tmap.VectorUint*, *arg2: int*) → tmap.VectorUint

Query the LSH forest for k-nearest neighbors.

Parameters

- **id** (`int`) – The id of an indexed MinHash vector
- **exclude** (`List` of `VectorUint`) –
- **k** (`int`) – The number of nearest neighbors to be retrieved

Returns The results of the query

Return type `VectorUint`

query_linear_scan (*self: tmap.LSHForest*, *vec: tmap.VectorUint*, *k: int*, *kc: int=10*, *weighted: bool=False*) → List[Tuple[float, int]]

Query k-nearest neighbors with a LSH forest / linear scan combination. `k`*:obj:`kc` nearest neighbors are searched for using LSH forest; from these, the `k` nearest neighbors are retrieved using linear scan.

Parameters

- **vec** (`VectorUint`) – The query MinHash vector
- **k** (`int`) – The number of nearest neighbors to be retrieved

Keyword Arguments

- **kc** (`int`) – The factor by which `k` is multiplied for LSH forest retreival
- **weighted** (`bool`) – Whether the MinHash vectors in this *`LSHForest`* instance are weighted

Returns The results of the query

Return type `List` of `Tuple[float, int]`

query_linear_scan_by_id (*self: tmap.LSHForest*, *id: int*, *k: int*, *kc: int=10*, *weighted: bool=False*) → List[Tuple[float, int]]

Query k-nearest neighbors with a LSH forest / linear scan combination. `k`*:obj:`kc` nearest neighbors are searched for using LSH forest; from these, the `k` nearest neighbors are retrieved using linear scan.

Parameters

- **id** (`int`) – The id of an indexed MinHash vector
- **k** (`int`) – The number of nearest neighbors to be retrieved

Keyword Arguments

- **kc** (int) – The factor by which k is multiplied for LSH forest retreival
- **weighted** (bool) – Whether the MinHash vectors in this *LSHForest* instance are weighted

Returns The results of the query

Return type List of Tuple[float, int]

query_linear_scan_exclude (*self: tmap.LSHForest*, *vec: tmap.VectorUint*, *k: int*, *exclude: tmap.VectorUint*, *kc: int=10*, *weighted: bool=False*) → List[Tuple[float, int]]

Query k-nearest neighbors with a LSH forest / linear scan combination. k`*:obj:`kc nearest neighbors are searched for using LSH forest; from these, the k nearest neighbors are retrieved using linear scan.

Parameters

- **vec** (VectorUint) – The query MinHash vector
- **k** (int) – The number of nearest neighbors to be retrieved

Keyword Arguments

- **exclude** (List of VectorUint) –
- **kc** (int) – The factor by which k is multiplied for LSH forest retreival
- **weighted** (bool) – Whether the MinHash vectors in this *LSHForest* instance are weighted

Returns The results of the query

Return type List of Tuple[float, int]

query_linear_scan_exclude_by_id (*self: tmap.LSHForest*, *id: int*, *k: int*, *exclude: tmap.VectorUint*, *kc: int=10*, *weighted: bool=False*) → List[Tuple[float, int]]

Query k-nearest neighbors with a LSH forest / linear scan combination. k`*:obj:`kc nearest neighbors are searched for using LSH forest; from these, the k nearest neighbors are retrieved using linear scan.

Parameters

- **id** (int) – The id of an indexed MinHash vector
- **k** (int) – The number of nearest neighbors to be retrieved

Keyword Arguments

- **exclude** (List of VectorUint) –
- **kc** (int) – The factor by which k is multiplied for LSH forest retreival
- **weighted** (bool) – Whether the MinHash vectors in this *LSHForest* instance are weighted

Returns The results of the query

Return type List of Tuple[float, int]

restore (*self: tmap.LSHForest*, *arg0: str*) → None

Deserializes a previously serialized (using *store()*) state into this instance of *LSHForest* and recreates the index.

Parameters **path** (str) – The path to the file which is deserialized

size (*self: tmap.LSHForest*) → int
Returns the number of MinHash vectors in this LSHForest instance.

Returns The number of MinHash vectors

Return type `int`

store (*self: tmap.LSHForest*, *arg0: str*) → None
Serializes the current state of this instance of *LSHForest* to the disk in binary format. The index is not serialized and has to be rebuilt after deserialization.

Parameters **path** (`str`) – The path to which to searialize the file

1.2.3 Layout

tmap.**layout_from_lsh_forest**()
layout_from_lsh_forest(lsh_forest: tmap::LSHForest, config: tmap.LayoutConfiguration=k: 10 kc: 10 fme_iterations: 1000 fme_randomize: 0 fme_threads: 4 fme_precision: 4 sl_repeats: 1 sl_extra_scaling_steps: 1 sl_scaling_x: 5.000000 sl_scaling_y: 20.000000 sl_scaling_type: RelativeToDrawing mmm_repeats: 1 placer: Barycenter merger: LocalBiconnected merger_factor: 2.000000 merger_adjustment: 0 node_size1.000000, create_mst: bool=True, clear_lsh_forest: bool=False, weighted: bool=False) -> Tuple[tmap.VectorFloat, tmap.VectorFloat, tmap.VectorUint, tmap.VectorUint, tmap.GraphProperties]

Create minimum spanning tree or k-nearest neighbor graph coordinates and topology from an *LSHForest* instance.

Arguments: lsh_forest (*LSHForest*): An *LSHForest* instance

Keyword Arguments: config (*LayoutConfiguration*, optional): An *LayoutConfiguration* instance create_mst (`bool`): Whether to create a minimum spanning tree or to return coordinates and topology for the k-nearest neighbor graph clear_lsh_forest (`bool`): Whether to run `clear()` on the *LSHForest* instance after k-nearest negihbor graph and MST creation and before layout weighted (`bool`): Whether the MinHash vectors in the *LSHForest* instance are weighted

Returns: `Tuple[VectorFloat, VectorFloat, VectorUint, VectorUint, GraphProperties]` The x and y coordinates of the vertices, the ids of the vertices spanning the edges, and information on the graph

tmap.**layout_from_edge_list**()
layout_from_edge_list(vertex_count: int, edges: List[Tuple[int, int, float]], config: tmap.LayoutConfiguration=k: 10 kc: 10 fme_iterations: 1000 fme_randomize: 0 fme_threads: 4 fme_precision: 4 sl_repeats: 1 sl_extra_scaling_steps: 1 sl_scaling_x: 5.000000 sl_scaling_y: 20.000000 sl_scaling_type: RelativeToDrawing mmm_repeats: 1 placer: Barycenter merger: LocalBiconnected merger_factor: 2.000000 merger_adjustment: 0 node_size1.000000, create_mst: bool=True) -> Tuple[tmap.VectorFloat, tmap.VectorFloat, tmap.VectorUint, tmap.VectorUint, tmap.GraphProperties]

Create minimum spanning tree or k-nearest neighbor graph coordinates and topology from an edge list.

Arguments: vertex_count (`int`): The number of vertices in the edge list edges (`List of Tuple[int, int, float]`): An edge list defining a graph

Keyword Arguments: config (*LayoutConfiguration*, optional): An *LayoutConfiguration* instance create_mst (`bool`): Whether to create a minimum spanning tree or to return coordinates and topology for the k-nearest neighbor graph

Returns: `Tuple[VectorFloat, VectorFloat, VectorUint, VectorUint, GraphProperties]`: The x and y coordinates of the vertices, the ids of the vertices spanning the edges, and information on the graph tmap.**mst_from_lsh_forest**(*lsh_forest: tmap::LSHForest*, *k: int*, *kc: int=10*, *weighted: bool=False*) → Tuple[tmap.VectorUint, tmap.VectorUint]

Create minimum spanning tree topology from an `LSHForest` instance.

Parameters

- **lsh_forest** (`LSHForest`) – An `LSHForest` instance
- **k** (`int`) – The number of nearest neighbors used to create the k-nearest neighbor graph

Keyword Arguments

- **kc** (`int`) – The scalar by which k is multiplied before querying the LSH forest. The results are then ordered decreasing based on linear-scan distances and the top k results returned
- **weighted** (`bool`) – Whether the MinHash vectors in the `LSHForest` instance are weighted

Returns the topology of the minimum spanning tree of the data indexed in the LSH forest

Return type `Tuple[VectorUint, VectorUint]`

class tmap.**ScalingType**(*self: tmap.ScalingType*, *arg0: int*) → None

The scaling types available in OGDF. The class is to be used as an enum.

Notes

The available values are

`ScalingType.Absolute`: Absolute factor, can be used to scale relative to level size change.

`ScalingType.RelativeToAvgLength`: Scales by a factor relative to the average edge weights.

`ScalingType.RelativeToDesiredLength`: Scales by a factor relative to the disired edge length.

`ScalingType.RelativeToDrawing`: Scales by a factor relative to the drawing.

class tmap.**Placer**(*self: tmap.Placer*, *arg0: int*) → None

The places available in OGDF. The class is to be used as an enum.

Notes

The available values are

`Placer.Barycenter`: Places a vertex at the barycenter of its neighbors' position.

`Placer.Solar`: Uses information of the merging phase of the solar merger. Places a new vertex on the direct line between two suns.

`Placer.Circle`: Places the vertices in a circle around the barycenter and outside of the current drawing

`Placer.Median`: Places a vertex at the median position of the neighbor nodes for each coordinate axis.

`Placer.Random`: Places a vertex at a random position within the smallest circle containing all vertices around the barycenter of the current drawing.

`Placer.Zero`: Places a vertex at the same position as its representative in the previous level.

class tmap.**Merger**(*self: tmap.Merger*, *arg0: int*) → None

The mergers available in OGDF. The class is to be used as an enum.

Notes

The available values are

`Merger.EdgeCover`: Based on the matching merger. Computes an edge cover such that each contained edge is incident to at least one unmatched vertex. The cover edges are then used to merge their adjacent vertices.

`Merger.LocalBiconnected`: Based on the edge cover merger. Avoids distortions by checking whether biconnectivity will be lost in the local neighborhood around the potential merging position.

`Merger.Solar`: Vertices are partitioned into solar systems, consisting of sun, planets and moons. The systems are then merged into the sun vertices.

`Merger.IndependentSet`: Uses a maximal independent set filtration. See GRIP for details.

class tmap.**LayoutConfiguration**(*self: tmap.LayoutConfiguration*) → None

A container for configuration options for *layout_from_lsh_forest()* and *layout_from_edge_list()*.

int k

The number of nearest neighbors used to create the k-nearest neighbor graph.

Type `int`

int kc

The scalar by which k is multiplied before querying the LSH forest. The results are then ordered decreasing based on linear-scan distances and the top k results returned.

Type `int`

int fme_iterations

Maximum number of iterations of the fast multipole embedder.

Type `int`

bool fme_randomize

Whether or not to randomize the layout at the start.

Type `bool`

int fme_threads

The number of threads for the fast multipole embedder.

Type `int`

int fme_precision

The number of coefficients of the multipole expansion.

Type `int`

int sl_repeats

The number of repeats of the scaling layout algorithm.

Type `int`

int sl_extra_scaling_steps

Sets the number of repeats of the scaling.

Type `int`

double sl_scaling_min

The minimum scaling factor.

Type `float`

double sl_scaling_max
The maximum scaling factor.

Type `float`

ScalingType sl_scaling_type
Defines the (relative) scale of the graph.

Type *ScalingType*

int mmm_repeats
Number of repeats of the per-level layout algorithm.

Type `int`

Placer placer
The method by which the initial positons of the vertices at eachlevel are defined.

Type *Placer*

Merger merger
The vertex merging strategy applied during the coarsening phaseof the multilevel algorithm.

Type *Merger*

double merger_factor
The ratio of the sizes between two levels up to which the mergingis run. Does not apply to all merging strategies.

Type `float`

int merger_adjustment
The edge length adjustment of the merging algorithm. Does notapply to all merging strategies.

Type `int`

float node_size
The size of the nodes, which affects the magnitude of their repellingforce. Decreasing this value generally resolves overlaps in a verycrowded tree.

Type `float`

Constructor for the class *LayoutConfiguration*.

__init__(*self: tmap.LayoutConfiguration*) → None
Constructor for the class *LayoutConfiguration*.

class tmap.**GraphProperties**(*self: tmap.GraphProperties*) → None
Contains properties of the minimum spanning tree (or forest) generated by *layout_from_lsh_forest()* and *layout_from_edge_list()*.

mst_weight
The total weight of the minimum spanning tree.

Type `float`

n_connected_components
The number of connected components in the minimum spanning forest.

Type `int`

n_isolated_vertices

Type `int`

degrees
The degrees of all vertices in the minimum spanning tree (or forest).

Type `VectorUint`

adjacency_list
The adjaceny lists for all vertices in the minimum spanning tree (or forest).

Type `List` of `VectorUint`

Constructor for the class *GraphProperties*.

__init__(*self: tmap.GraphProperties*) → None
Constructor for the class *GraphProperties*.

INDEX

Symbols

A

B

C

D

F

G

I

L

M

N

P

Q

R

S