Package ‘quitefastmst’

July 23, 2025

Type Package

Title Euclidean and Mutual Reachability Minimum Spanning Trees

Version 0.9.0

Date 2025-07-22

Description Functions to compute Euclidean minimum spanning trees using single-, sesqui-, and dual-tree Boruvka algorithms. Thanks to K-d trees, they are fast in spaces of low intrinsic dimensionality. Mutual reachability distances (used in the definition of the 'HDBSCAN*' algorithm) are also supported. The package also features relatively fast fallback minimum spanning tree and nearest-neighbours algorithms for spaces of higher dimensionality. The 'Python' version of 'quitefastmst' is available via 'PyPI'.

BugReports `https://github.com/gagolews/quitefastmst/issues`

URL `https://quitefastmst.gagolewski.com/`,
`https://github.com/gagolews/quitefastmst`

License AGPL-3

Imports Rcpp

Suggests datasets

LinkingTo Rcpp

Encoding UTF-8

SystemRequirements OpenMP, C++17

RoxygenNote 7.3.2

NeedsCompilation yes

Author Marek Gagolewski [aut, cre, cph] (ORCID: <`https://orcid.org/0000-0003-0637-6028`>)

Maintainer Marek Gagolewski <`marek@gagolewski.com`>

Repository CRAN

Date/Publication 2025-07-23 19:00:11 UTC

Contents

`knn_euclid` *Euclidean Nearest Neighbours*

Description

If `Y` is `NULL`, then the function determines the first `k` nearest neighbours of each point in `X` with respect to the Euclidean distance. It is assumed that each query point is not its own neighbour.

Otherwise, for each point in `Y`, this function determines the `k` nearest points thereto from `X`.

Usage

```
knn_euclid(
  X,
  k = 1L,
  Y = NULL,
  algorithm = "auto",
  max_leaf_size = 0L,
  squared = FALSE,
  verbose = FALSE
)
```

Arguments

`X`	the "database"; a matrix of shape $n \times d$
`k`	requested number of nearest neighbours (should be rather small)
`Y`	the "query points"; `NULL` or a matrix of shape $m \times d$; note that setting `Y=X`, contrary to `NULL`, will include the query points themselves amongst their own neighbours
`algorithm`	`"auto"`, `"kd_tree"` or `"brute"`; K-d trees can be used for `d` between 2 and 20 only; `"auto"` selects `"kd_tree"` in low-dimensional spaces
`max_leaf_size`	maximal number of points in the K-d tree leaves; smaller leaves use more memory, yet are not necessarily faster; use `0` to select the default value, currently set to 32
`squared`	whether the output `nn.dist` should be based on the squared Euclidean distance
`verbose`	whether to print diagnostic messages

Details

The implemented algorithms, see the `algorithm` parameter, assume that k is rather small, say, $k \leq 20$.

Our implementation of K-d trees (Bentley, 1975) has been quite optimised; amongst others, it has good locality of reference (at the cost of making a copy of the input dataset), features the sliding midpoint (midrange) rule suggested by Maneewongvatana and Mound (1999), node pruning strategies inspired by some ideas from (Sample et al., 2001), and a couple of further tuneups proposed by the current author. Still, it is well-known that K-d trees perform well only in spaces of low intrinsic dimensionality. Thus, due to the so-called curse of dimensionality, for high `d`, the brute-force algorithm is recommended.

The number of threads used is controlled via the `OMP_NUM_THREADS` environment variable or via the `omp_set_num_threads` function at runtime. For best speed, consider building the package from sources using, e.g., `-O3 -march=native` compiler flags.

Value

A list with two elements, `nn.index` and `nn.dist`, is returned.

`nn.dist` and `nn.index` have shape $n \times k$ or $m \times k$, depending whether `Y` is given.

`nn.index[i,j]` is the index (between 1 and n) of the j-th nearest neighbour of i.

`nn.dist[i,j]` gives the weight of the edge `{i, nn.index[i,j]}`, i.e., the distance between the i-th point and its j-th nearest neighbour, $j = 1, \dots, k$. `nn.dist[i,]` is sorted nondecreasingly for all i.

Author(s)

Marek Gagolewski

References

J.L. Bentley, Multidimensional binary search trees used for associative searching, *Communications of the ACM* 18(9), 509–517, 1975, doi:10.1145/361002.361007.

S. Maneewongvatana, D.M. Mount, It's okay to be skinny, if your friends are fat, *4th CGC Workshop on Computational Geometry*, 1999.

N. Sample, M. Haines, M. Arnold, T. Purcell, Optimizing search strategies in K-d Trees, *5th WSES/IEEE Conf. on Circuits, Systems, Communications & Computers* (CSCC'01), 2001.

See Also

The official online manual of **quitefastmst** at `https://quitefastmst.gagolewski.com/`

`mst_euclid`

Examples

```
library("datasets")
data("iris")
X <- jitter(as.matrix(iris[1:2]))  # some data
neighbours <- knn_euclid(X, 1)  # 1-NNs of each point
```

```
plot(X, asp=1, las=1)
segments(X[,1], X[,2], X[neighbours$nn.index,1], X[neighbours$nn.index,2])

knn_euclid(X, 5, matrix(c(6, 4), nrow=1))  # five closest points to (6, 4)
```

`mst_euclid` *Euclidean and Mutual Reachability Minimum Spanning Trees*

Description

The function determines the/a(*) minimum spanning tree (MST) of a set of n points, i.e., an acyclic undirected connected graph whose vertices represent the points, edges are weighted by the distances between point pairs, and have minimal total weight.

MSTs have many uses in, amongst others, topological data analysis (clustering, density estimation, dimensionality reduction, outlier detection, etc.).

In clustering and density estimation, the parameter `M` plays the role of a smoothing factor; for discussion, see (Campello et al., 2015) and the references therein. `M` corresponds to the **hdbscan** Python package's `min_samples=M-1`.

For $M \leq 2$, we get a spanning tree that minimises the sum of Euclidean distances between the points, i.e., the classic Euclidean minimum spanning tree (EMST). If $M = 2$, the function additionally returns the distance to each point's nearest neighbour.

If $M > 2$, the spanning tree is the smallest with respect to the degree-M mutual reachability distance (Campello et al., 2013) given by $d_M(i,j) = \max\{c_M(i), c_M(j), d(i,j)\}$, where $d(i,j)$ is the standard Euclidean distance between the i-th and the j-th point, and $c_M(i)$ is the i-th M-core distance defined as the distance between the i-th point and its $(M-1)$-th nearest neighbour (not including the query point itself).

Usage

```
mst_euclid(
  X,
  M = 1L,
  algorithm = "auto",
  max_leaf_size = 0L,
  first_pass_max_brute_size = 0L,
  mutreach_adj = -1.00000011920929,
  verbose = FALSE
)
```

Arguments

`X`	the "database"; a matrix of shape $n \times d$
`M`	the degree of the mutual reachability distance (should be rather small). $M \leq 2$ denotes the ordinary Euclidean distance

`algorithm`	`"auto"`, `"single_kd_tree"`, `"sesqui_kd_tree"`, `"dual_kd_tree"`, or `"brute"`; K-d trees can only be used for d between 2 and 20 only; `"auto"` selects `"sesqui_kd_tree"` for $d \leq 20$. `"brute"` is used otherwise
`max_leaf_size`	maximal number of points in the K-d tree leaves; smaller leaves use more memory, yet are not necessarily faster; use `0` to select the default value, currently set to 32 for the single-tree and sesqui-tree and 8 for the dual-tree Borůvka algorithm
`first_pass_max_brute_size`	minimal number of points in a node to treat it as a leaf (unless it's actually a leaf) in the first iteration of the algorithm; use `0` to select the default value, currently set to 32
`mutreach_adj`	adjustment for mutual reachability distance ambiguity (for $M > 2$) whose fractional part should be close to 0: values in $(-1, 0)$ prefer connecting to farther nearest neighbours, values in $(0, 1)$ fall for closer NNs (which is what many other implementations provide), values in $(-2, -1)$ prefer connecting to points with smaller core distances, values in $(1, 2)$ favour larger core distances; see below for more details
`verbose`	whether to print diagnostic messages

Details

(*) We note that if there are many pairs of equidistant points, there can be many minimum spanning trees. In particular, it is likely that there are point pairs with the same mutual reachability distances.

To make the definition less ambiguous (albeit with no guarantees), internally, the brute-force algorithm relies on the adjusted distance: $d_M(i,j) = \max\{c_M(i), c_M(j), d(i,j)\} + \varepsilon d(i,j)$ or $d_M(i,j) = \max\{c_M(i), c_M(j), d(i,j)\} - \varepsilon \min\{c_M(i), c_M(j)\}$, where ε is close to 0. |`mutreach_adj`|<1 selects the former formula (ε=`mutreach_adj`) whilst 1<|`mutreach_adj`|<2 chooses the latter (ε=`mutreach_adj`±1).

For the K-d tree-based methods, on the other hand, `mutreach_adj` indicates the preference towards connecting to farther/closer points with respect to the original metric or having smaller/larger core distances if a point i has multiple nearest-neighbour candidates j', j'' with $c_M(i) \geq \max\{d(i,j'), c_M(j')\}$ and $c_M(i) \geq \max\{d(i,j''), c_M(j'')\}$.

Generally, the smaller the `mutreach_adj`, the more leaves should be in the tree (note that there are only four types of adjustments, though).

The implemented algorithms, see the `algorithm` parameter, assume that M is rather small; say, $M \leq 20$.

Our implementation of K-d trees (Bentley, 1975) has been quite optimised; amongst others, it has good locality of reference (at the cost of making a copy of the input dataset), features the sliding midpoint (midrange) rule suggested by Maneewongvatana and Mound (1999), node pruning strategies inspired by some ideas from (Sample et al., 2001), and a couple of further tuneups proposed by the current author.

The "single-tree" version of the Borůvka algorithm is parallelised: in every iteration, it seeks each point's nearest "alien", i.e., the nearest point thereto from another cluster. The "dual-tree" Borůvka version of the algorithm is, in principle, based on (March et al., 2010). As far as our implementation is concerned, the dual-tree approach is often only faster in 2- and 3-dimensional spaces, for $M \leq 2$, and in a single-threaded setting. For another (approximate) adaptation of the dual-tree algorithm to mutual reachability distances, see (McInnes and Healy, 2017).

The "sesqui-tree" variant (by the current author) is a mixture of the two approaches: it compares leaves against the full tree and can be run in parallel. It is usually faster than the single- and dual-tree methods in very low dimensional spaces and usually not much slower than the single-tree variant otherwise.

Nevertheless, it is well-known that K-d trees perform well only in spaces of low intrinsic dimensionality (the "curse"). For high d, the "brute-force" algorithm is recommended. Here, we provided a parallelised (see Olson, 1995) version of the Jarník (1930) (a.k.a. Prim, 1957) algorithm, where the distances are computed on the fly (only once for $M \leq 2$).

The number of threads used is controlled via the `OMP_NUM_THREADS` environment variable or via the `omp_set_num_threads` function at runtime. For best speed, consider building the package from sources using, e.g., `-O3 -march=native` compiler flags.

Value

A list with two $(M=1)$ or four $(M>1)$ elements, `mst.index` and `mst.dist`, and additionally `nn.index` and `nn.dist`.

`mst.index` is a matrix with $n-1$ rows and 2 columns, whose rows define the tree edges.

`mst.dist` is a vector of length $n-1$ giving the weights of the corresponding edges.

The tree edges are ordered with respect to weights nondecreasingly, and then by the indexes (lexicographic ordering of the `(weight, index1, index2)` triples). For each `i`, it holds `mst_ind[i,1]<mst_ind[i,2]`.

`nn.index` is an n by $M-1$ matrix giving the indexes of each point's nearest neighbours with respect to the Euclidean distance. `nn.dist` provides the corresponding distances.

Author(s)

Marek Gagolewski

References

V. Jarník, O jistém problému minimálním, *Práce Moravské Přírodovědecké Společnosti* 6, 1930, 57–63.

C.F. Olson, Parallel algorithms for hierarchical clustering, Parallel Computing 21(8), 1995, 1313–1325.

R. Prim, Shortest connection networks and some generalizations, *The Bell System Technical Journal* 36(6), 1957, 1389–1401.

O. Borůvka, O jistém problému minimálním, *Práce Moravské Přírodovědecké Společnosti* 3, 1926, 37–58.

W.B. March, R. Parikshit, A.G. Gray, Fast Euclidean minimum spanning tree: Algorithm, analysis, and applications, *Proc. 16th ACM SIGKDD Intl. Conf. Knowledge Discovery and Data Mining (KDD '10)*, 2010, 603–612.

J.L. Bentley, Multidimensional binary search trees used for associative searching, *Communications of the ACM* 18(9), 509–517, 1975, doi:10.1145/361002.361007.

S. Maneewongvatana, D.M. Mount, It's okay to be skinny, if your friends are fat, *4th CGC Workshop on Computational Geometry*, 1999.

N. Sample, M. Haines, M. Arnold, T. Purcell, Optimizing search strategies in K-d Trees, *5th WSES/IEEE Conf. on Circuits, Systems, Communications & Computers* (CSCC'01), 2001.

R.J.G.B. Campello, D. Moulavi, J. Sander, Density-based clustering based on hierarchical density estimates, *Lecture Notes in Computer Science* 7819, 2013, 160–172. doi:10.1007/978364237456-2_14.

R.J.G.B. Campello, D. Moulavi, A. Zimek, J. Sander, Hierarchical density estimates for data clustering, visualization, and outlier detection, *ACM Transactions on Knowledge Discovery from Data (TKDD)* 10(1), 2015, 1–51, doi:10.1145/2733381.

L. McInnes, J. Healy, Accelerated hierarchical density-based clustering, *IEEE Intl. Conf. Data Mining Workshops (ICMDW)*, 2017, 33–42, doi:10.1109/ICDMW.2017.12.

See Also

The official online manual of **quitefastmst** at https://quitefastmst.gagolewski.com/

knn_euclid

Examples

```
library("datasets")
data("iris")
X <- jitter(as.matrix(iris[1:2]))  # some data
T <- mst_euclid(X)                 # Euclidean MST of X
plot(X, asp=1, las=1)
segments(X[T$mst.index[, 1], 1], X[T$mst.index[, 1], 2],
         X[T$mst.index[, 2], 1], X[T$mst.index[, 2], 2])
```

---

omp_set_num_threads   *Get or Set the Number of Threads*

Description

These functions get or set the maximal number of OpenMP threads that can be used by knn_euclid and mst_euclid, amongst others.

Usage

```
omp_set_num_threads(n_threads)

omp_get_max_threads()
```

Arguments

n_threads	maximum number of threads to use

Value

omp_get_max_threads returns the maximal number of threads that will be used during the next call to a parallelised function, not the maximal number of threads possibly available. It there is no built-in support for OpenMP, 1 is always returned.

For omp_set_num_threads, the previous value of max_threads is returned.

Author(s)

Marek Gagolewski

See Also

The official online manual of **quitefastmst** at `https://quitefastmst.gagolewski.com/`

Index