Package 'seqtrie'

October 20, 2025

Title Radix Tree and Trie-Based String Distances

Version 0.3.5

Date 2025-10-19

Description A collection of Radix Tree and Trie algorithms for finding similar sequences and calculating sequence distances (Levenshtein and other distance metrics). This work was inspired by a trie implementation in Python: ``Fast and Easy Levenshtein distance using a Trie." Hanov (2011) <https://stevehanov.ca/blog/index.php?id=114>.

License GPL-3

Biarch true

Encoding UTF-8

Depends R (>= 3.5.0)

LazyData true

SystemRequirements GNU make

LinkingTo Rcpp, RcppParallel, BH

Imports Rcpp (>= 0.12.18.3), RcppParallel (>= 5.1.3), R6, rlang, dplyr, stringi

Suggests knitr, rmarkdown, stringdist, pwalign, igraph, ggplot2

VignetteBuilder knitr

RoxygenNote 7.3.2

Copyright This package includes code from the 'span-lite' library owned by Martin Moene under Boost Software License 1.0. This package includes code from the 'ankerl' library owned by Martin Leitner-Ankerl under MIT License. This package contains data derived from Adaptive Biotechnologies ``ImmuneCODE'' dataset under Creative Commons Attribution 4.0.

URL https://github.com/traversc/seqtrie

BugReports https://github.com/traversc/seqtrie/issues

NeedsCompilation yes

Author Travers Ching [aut, cre, cph],
Martin Moene [ctb, cph] (span-lite C++ library),
Steve Hanov [ctb] (Trie levenshtein implementation in Python),
Martin Leitner-Ankerl [ctb] (Ankerl unordered dense hashmap)

Maintainer Travers Ching <traversc@gmail.com>

Repository CRAN

Date/Publication 2025-10-20 08:00:02 UTC

Contents

`covid_cdr3` *Adaptive COVID TCRB CDR3 data*

Description

Unique TCRB CDR3 sequences from the Nolan et al. 2020. CDR3s were extracted via IgBLAST. The license for this data is Creative Commons Attribution 4.0 International License.

Usage

```
data(covid_cdr3)
```

Format

A character vector of length 133,034.

References

Nolan, Sean, et al. "A large-scale database of T-cell receptor beta (TCRB) sequences and binding associations from natural and synthetic exposure to SARS-CoV-2." (2020). doi: 10.21203/rs.3.rs-51964/v1.

Examples

```
data(covid_cdr3)
# Average CDR3 length
mean(nchar(covid_cdr3)) # [1] 43.56821
```

`dist_matrix` *Compute distances between all combinations of two sets of sequences*

Description

Compute distances between all combinations of query and target sequences

Usage

```
dist_matrix(
  query,
  target,
  mode,
  cost_matrix = NULL,
  gap_cost = NA_integer_,
  gap_open_cost = NA_integer_,
  nthreads = 1,
  show_progress = FALSE
)
```

Arguments

`query`	A character vector of query sequences.
`target`	A character vector of target sequences.
`mode`	The distance metric to use. One of hamming (hm), global (gb) or anchored (an).
`cost_matrix`	A custom cost matrix for use with the "global" or "anchored" distance metrics. See details.
`gap_cost`	The cost of a gap for use with the "global" or "anchored" distance metrics. See details.
`gap_open_cost`	The cost of a gap opening. See details.
`nthreads`	The number of threads to use for parallel computation.
`show_progress`	Whether to show a progress bar.

Details

This function calculates all combinations of pairwise distances based on Hamming, Levenshtein or Anchored algorithms. The output is a NxM matrix where N = length(query) and M = length(target). Note: this can take a *really* long time; be careful with input size.

Three types of distance metrics are supported, based on the form of alignment performed. These are: Hamming, Global (Levenshtein) and Anchored.

An anchored alignment is a form of semi-global alignment, where the query sequence is "anchored" (global) to the beginning of both the query and target sequences, but is semi-global in that the end of the either the query sequence or target sequence (but not both) can be unaligned. This type of alignment is sometimes called an "extension" alignment in literature.

In contrast a global alignment must align the entire query and target sequences. When mismatch and indel costs are equal to 1, this is also known as the Levenshtein distance.

By default, if mode == "global" or "anchored", all mismatches and indels are given a cost of 1. However, you can define your own distance metric by setting the substitution cost_matrix and separate gap parameters. The cost_matrix is a strictly positive square integer matrix of substitution costs and should include all characters in query and target as column- and rownames. Any rows/columns named "gap" or "gap_open" are ignored. To set the cost of a gap (insertion or deletion), use the gap_cost parameter (a single positive integer). To enable affine gaps, provide the gap_open_cost parameter (a single positive integer) in addition to gap_cost. If affine alignment is used, the total cost of a gap of length L is defined as: TOTAL_GAP_COST = gap_open_cost + (gap_cost * gap_length).

If mode == "hamming" all alignment parameters are ignored; mismatch is given a distance of 1 and gaps are not allowed.

Value

The output is a distance matrix between all query (rows) and target (columns) sequences. For anchored searches, the output also includes attributes "query_size" and "target_size" which are matrices containing the lengths of the query and target sequences that are aligned.

Examples

```
dist_matrix(c("ACGT", "AAAA"), c("ACG", "ACGT"), mode = "global")
```

`dist_pairwise` *Pairwise distance between two sets of sequences*

Description

Compute the pairwise distance between two sets of sequences

Usage

```
dist_pairwise(
  query,
  target,
  mode,
  cost_matrix = NULL,
  gap_cost = NA_integer_,
  gap_open_cost = NA_integer_,
  nthreads = 1,
  show_progress = FALSE
)
```

Arguments

query	A character vector of query sequences.
target	A character vector of target sequences.. Must be the same length as query.
mode	The distance metric to use. One of hamming (hm), global (gb) or anchored (an).
cost_matrix	A custom cost matrix for use with the "global" or "anchored" distance metrics. See details.
gap_cost	The cost of a gap for use with the "global" or "anchored" distance metrics. See details.
gap_open_cost	The cost of a gap opening. See details.
nthreads	The number of threads to use for parallel computation.
show_progress	Whether to show a progress bar.

Details

This function calculates pairwise distances based on Hamming, Levenshtein or Anchored algorithms. *query* and *target* must be the same length.

Three types of distance metrics are supported, based on the form of alignment performed. These are: Hamming, Global (Levenshtein) and Anchored.

An anchored alignment is a form of semi-global alignment, where the query sequence is "anchored" (global) to the beginning of both the query and target sequences, but is semi-global in that the end of the either the query sequence or target sequence (but not both) can be unaligned. This type of alignment is sometimes called an "extension" alignment in literature.

In contrast a global alignment must align the entire query and target sequences. When mismatch and indel costs are equal to 1, this is also known as the Levenshtein distance.

By default, if mode == "global" or "anchored", all mismatches and indels are given a cost of 1. However, you can define your own distance metric by setting the substitution cost_matrix and separate gap parameters. The cost_matrix is a strictly positive square integer matrix of substitution costs and should include all characters in query and target as column- and rownames. Any rows/columns named "gap" or "gap_open" are ignored. To set the cost of a gap (insertion or deletion), use the gap_cost parameter (a single positive integer). To enable affine gaps, provide the gap_open_cost parameter (a single positive integer) in addition to gap_cost. If affine alignment is used, the total cost of a gap of length L is defined as: TOTAL_GAP_COST = gap_open_cost + (gap_cost * gap_length).

If mode == "hamming" all alignment parameters are ignored; mismatch is given a distance of 1 and gaps are not allowed.

Value

The output of this function is a vector of distances. If mode == "anchored" then the output also includes attributes "query_size" and "target_size" which are vectors containing the lengths of the query and target sequences that are aligned.

Examples

```
dist_pairwise(c("ACGT", "AAAA"), c("ACG", "ACGT"), mode = "global")
```

---

dist_search *Distance search for similar sequences*

Description

Find similar sequences within a distance threshold

Usage

```
dist_search(
  query,
  target,
  max_distance = NULL,
  max_fraction = NULL,
  mode = "levenshtein",
  cost_matrix = NULL,
  gap_cost = NA_integer_,
  gap_open_cost = NA_integer_,
  tree_class = "RadixTree",
  nthreads = 1,
  show_progress = FALSE
)
```

Arguments

query	A character vector of query sequences.
target	A character vector of target sequences.
max_distance	how far to search in units of absolute distance. Can be a single value or a vector. Mutually exclusive with max_fraction.
max_fraction	how far to search in units of relative distance to each query sequence length. Can be a single value or a vector. Mutually exclusive with max_distance.
mode	The distance metric to use. One of hamming (hm), global (gb) or anchored (an).

cost_matrix	A custom cost matrix for use with the "global" or "anchored" distance metrics. See details.
gap_cost	The cost of a gap for use with the "global" or "anchored" distance metrics. See details.
gap_open_cost	The cost of a gap opening. See details.
tree_class	Which R6 class to use. Either RadixTree or RadixForest (default: RadixTree)
nthreads	The number of threads to use for parallel computation.
show_progress	Whether to show a progress bar.

Details

This function finds all sequences in *target* that are within a distance threshold of any sequence in *query*. This function uses either a RadixTree or RadixForest to store *target* sequences. See the R6 class documentation for additional details.

Three types of distance metrics are supported, based on the form of alignment performed. These are: Hamming, Global (Levenshtein) and Anchored.

An anchored alignment is a form of semi-global alignment, where the query sequence is "anchored" (global) to the beginning of both the query and target sequences, but is semi-global in that the end of the either the query sequence or target sequence (but not both) can be unaligned. This type of alignment is sometimes called an "extension" alignment in literature.

In contrast a global alignment must align the entire query and target sequences. When mismatch and indel costs are equal to 1, this is also known as the Levenshtein distance.

By default, if mode == "global" or "anchored", all mismatches and indels are given a cost of 1. However, you can define your own distance metric by setting the substitution cost_matrix and separate gap parameters. The cost_matrix is a strictly positive square integer matrix of substitution costs and should include all characters in query and target as column- and rownames. Any rows/columns named "gap" or "gap_open" are ignored. To set the cost of a gap (insertion or deletion), use the gap_cost parameter (a single positive integer). To enable affine gaps, provide the gap_open_cost parameter (a single positive integer) in addition to gap_cost. If affine alignment is used, the total cost of a gap of length L is defined as: TOTAL_GAP_COST = gap_open_cost + (gap_cost * gap_length).

If mode == "hamming" all alignment parameters are ignored; mismatch is given a distance of 1 and gaps are not allowed.

Value

The output is a data.frame of all matches with columns "query" and "target". For anchored searches, the output also includes attributes "query_size" and "target_size" which are vectors containing the portion of the query and target sequences that are aligned.

Examples

```
dist_search(c("ACGT", "AAAA"), c("ACG", "ACGT"), max_distance = 1, mode = "levenshtein")
```

generate_cost_matrix *Generate a simple cost matrix*

Description

Generate a cost matrix for use with the search method.

Usage

```
generate_cost_matrix(charset, ambiguity_base = NULL, match = 0L, mismatch = 1L)
```

Arguments

charset	A string of all allowed characters in both query and target sequences (e.g. "ACGT").
ambiguity_base	A single character (e.g. "N") that will match any character in charset at the cost of match. Defaults to NULL.
match	Integer cost of a match.
mismatch	Integer cost of a mismatch.

Value

A square cost matrix with row- and column-names given by charset, plus the optional ambiguity_base. Gap costs are no longer included here; pass gap_cost and gap_open_cost to distance/search functions.

Examples

```
generate_cost_matrix("ACGT", match = 0, mismatch = 1)
generate_cost_matrix("ACGT", ambiguity_base = "N", match = 0, mismatch = 1)
```

RadixForest *RadixForest*

Description

Radix Forest class implementation

Details

The RadixForest class is a specialization of the RadixTree implementation. Instead of putting sequences into a single tree, the RadixForest class puts sequences into separate trees based on sequence length. This allows for faster searching of similar sequences based on Hamming or Levenshtein distance metrics. Unlike the RadixTree class, the RadixForest class does not support anchored searches or a custom cost matrix. See *RadixTree* for additional details.

Public fields

`forest_pointer` Map of sequence length to RadixTree

`char_counter_pointer` Character count data for the purpose of validating input

Methods

Public methods:

- `RadixForest$new()`
- `RadixForest$show()`
- `RadixForest$to_string()`
- `RadixForest$graph()`
- `RadixForest$to_vector()`
- `RadixForest$size()`
- `RadixForest$insert()`
- `RadixForest$erase()`
- `RadixForest$find()`
- `RadixForest$prefix_search()`
- `RadixForest$search()`
- `RadixForest$validate()`

Method `new()`**:** Create a new RadixForest object

Usage - new:

```
RadixForest$new(sequences = NULL)
```

Arguments - new:

`sequences` A character vector of sequences to insert into the forest

Method `show()`**:** Print the forest to screen

Usage - show:

```
RadixForest$show()
```

Method `to_string()`**:** Print the forest to a string

Usage - to_string:

```
RadixForest$to_string()
```

Method `graph()`**:** Plot of the forest using igraph

Usage - graph:

```
RadixForest$graph(depth = -1, root_label = "root", plot = TRUE)
```

Arguments - graph:

`depth` The tree depth to plot for each tree in the forest.

`root_label` The label of the root node(s) in the plot.

`plot` Whether to create a plot or return the data used to generate the plot.

Returns - graph: A data frame of parent-child relationships used to generate the igraph plot OR a ggplot2 object

Method `to_vector()`: Output all sequences held by the forest as a character vector

Usage - to_vector:

```
RadixForest$to_vector()
```

Returns - to_vector: A character vector of all sequences contained in the forest.

Method `size()`: Output the size of the forest (i.e. how many sequences are contained)

Usage - size:

```
RadixForest$size()
```

Returns - size: The size of the forest

Method `insert()`: Insert new sequences into the forest

Usage - insert:

```
RadixForest$insert(sequences)
```

Arguments - insert:

`sequences` A character vector of sequences to insert into the forest

Returns - insert: A logical vector indicating whether the sequence was inserted (TRUE) or already existing in the forest (FALSE)

Method `erase()`: Erase sequences from the forest

Usage - erase:

```
RadixForest$erase(sequences)
```

Arguments - erase:

`sequences` A character vector of sequences to erase from the forest

Returns - erase: A logical vector indicating whether the sequence was erased (TRUE) or not found in the forest (FALSE)

Method `find()`: Find sequences in the forest

Usage - find:

```
RadixForest$find(query)
```

Arguments - find:

`query` A character vector of sequences to find in the forest

Returns - find: A logical vector indicating whether the sequence was found (TRUE) or not found in the forest (FALSE)

Method `prefix_search()`: Search for sequences in the forest that start with a specified prefix. E.g.: a query of "CAR" will find "CART", "CARBON", "CARROT", etc. but not "CATS".

Usage - prefix_search:

```
RadixForest$prefix_search(query)
```

Arguments - prefix_search:

`query` A character vector of sequences to search for in the forest

Returns - prefix_search: A data frame of all matches with columns "query" and "target".

Method `search()`**:** Search for sequences in the forest that are with a specified distance metric to a specified query.

Usage - search:

```
RadixForest$search(
  query,
  max_distance = NULL,
  max_fraction = NULL,
  mode = "levenshtein",
  nthreads = 1,
  show_progress = FALSE
)
```

Arguments - search:

`query` A character vector of query sequences.

`max_distance` how far to search in units of absolute distance. Can be a single value or a vector. Mutually exclusive with max_fraction.

`max_fraction` how far to search in units of relative distance to each query sequence length. Can be a single value or a vector. Mutually exclusive with max_distance.

`mode` The distance metric to use. One of hamming (hm), global (gb) or anchored (an).

`nthreads` The number of threads to use for parallel computation.

`show_progress` Whether to show a progress bar.

Returns - search: The output is a data.frame of all matches with columns "query" and "target".

Method `validate()`**:** Validate the forest

Usage - validate:

```
RadixForest$validate()
```

Returns - validate: A logical indicating whether the forest is valid (TRUE) or not (FALSE). This is mostly an internal function for debugging purposes and should always return TRUE.

Examples

```
forest <- RadixForest$new()
forest$insert(c("ACGT", "AAAA"))
forest$erase("AAAA")
forest$search("ACG", max_distance = 1, mode = "levenshtein")
 #   query target distance
 # 1   ACG   ACGT        1

forest$search("ACG", max_distance = 1, mode = "hamming")
 # query    target   distance
 # <0 rows> (or 0-length row.names)
```

RadixTree *RadixTree*

Description

Radix Tree (trie) class implementation

Details

The RadixTree class is a trie implementation. The primary usage is to be able to search of similar sequences based on a dynamic programming framework. This can be done using the *search* method which searches for similar sequences based on the Global, Anchored or Hamming distance metrics.

Three types of distance metrics are supported, based on the form of alignment performed. These are: Hamming, Global (Levenshtein) and Anchored.

An anchored alignment is a form of semi-global alignment, where the query sequence is "anchored" (global) to the beginning of both the query and target sequences, but is semi-global in that the end of the either the query sequence or target sequence (but not both) can be unaligned. This type of alignment is sometimes called an "extension" alignment in literature.

In contrast a global alignment must align the entire query and target sequences. When mismatch and indel costs are equal to 1, this is also known as the Levenshtein distance.

By default, if mode == "global" or "anchored", all mismatches and indels are given a cost of 1. However, you can define your own distance metric by setting the substitution cost_matrix and separate gap parameters. The cost_matrix is a strictly positive square integer matrix of substitution costs and should include all characters in query and target as column- and rownames. Any rows/columns named "gap" or "gap_open" are ignored. To set the cost of a gap (insertion or deletion), use the gap_cost parameter (a single positive integer). To enable affine gaps, provide the gap_open_cost parameter (a single positive integer) in addition to gap_cost. If affine alignment is used, the total cost of a gap of length L is defined as: TOTAL_GAP_COST = gap_open_cost + (gap_cost * gap_length).

If mode == "hamming" all alignment parameters are ignored; mismatch is given a distance of 1 and gaps are not allowed.

Public fields

`root_pointer` Root of the RadixTree

`char_counter_pointer` Character count data for the purpose of validating input

Methods

Public methods:

- `RadixTree$new()`
- `RadixTree$show()`
- `RadixTree$to_string()`
- `RadixTree$graph()`

- RadixTree$to_vector()
- RadixTree$size()
- RadixTree$insert()
- RadixTree$erase()
- RadixTree$find()
- RadixTree$prefix_search()
- RadixTree$search()
- RadixTree$single_gap_search()
- RadixTree$validate()

Method `new()`**:** Create a new RadixTree object

Usage - new:

```
RadixTree$new(sequences = NULL)
```

Arguments - new:

`sequences` A character vector of sequences to insert into the tree

Method `show()`**:** Print the tree to screen

Usage - show:

```
RadixTree$show()
```

Method `to_string()`**:** Print the tree to a string

Usage - to_string:

```
RadixTree$to_string()
```

Returns - to_string: A string representation of the tree

Method `graph()`**:** Plot of the tree using igraph (needs to be installed separately)

Usage - graph:

```
RadixTree$graph(depth = -1, root_label = "root", plot = TRUE)
```

Arguments - graph:

`depth` The tree depth to plot. If -1 (default), plot the entire tree.

`root_label` The label of the root node in the plot.

`plot` Whether to create a plot or return the data used to generate the plot.

Returns - graph: A data frame of parent-child relationships used to generate the igraph plot OR a ggplot2 object

Method `to_vector()`**:** Output all sequences held by the tree as a character vector

Usage - to_vector:

```
RadixTree$to_vector()
```

Returns - to_vector: A character vector of all sequences contained in the tree. Return order is not guaranteed.

Method `size()`**:** Output the size of the tree (i.e. how many sequences are contained)

Usage - size:

```
RadixTree$size()
```

Returns - size: The size of the tree

Method `insert()`**:** Insert new sequences into the tree

Usage - insert:

```
RadixTree$insert(sequences)
```

Arguments - insert:

`sequences` A character vector of sequences to insert into the tree

Returns - insert: A logical vector indicating whether the sequence was inserted (TRUE) or already existing in the tree (FALSE)

Method `erase()`**:** Erase sequences from the tree

Usage - erase:

```
RadixTree$erase(sequences)
```

Arguments - erase:

`sequences` A character vector of sequences to erase from the tree

Returns - erase: A logical vector indicating whether the sequence was erased (TRUE) or not found in the tree (FALSE)

Method `find()`**:** Find sequences in the tree

Usage - find:

```
RadixTree$find(query)
```

Arguments - find:

`query` A character vector of sequences to find in the tree

Returns - find: A logical vector indicating whether the sequence was found (TRUE) or not found in the tree (FALSE)

Method `prefix_search()`**:** Search for sequences in the tree that start with a specified prefix. E.g.: a query of "CAR" will find "CART", "CARBON", "CARROT", etc. but not "CATS".

Usage - prefix_search:

```
RadixTree$prefix_search(query)
```

Arguments - prefix_search:

`query` A character vector of sequences to search for in the tree

Returns - prefix_search: A data frame of all matches with columns "query" and "target".

Method `search()`**:** Search for sequences in the tree that are with a specified distance metric to a specified query.

Usage - search:

```
RadixTree$search(
  query,
  max_distance = NULL,
  max_fraction = NULL,
  mode = "levenshtein",
```

```
  cost_matrix = NULL,
  gap_cost = NA_integer_,
  gap_open_cost = NA_integer_,
  nthreads = 1,
  show_progress = FALSE
)
```

Arguments - search:

`query` A character vector of query sequences.

`max_distance` how far to search in units of absolute distance. Can be a single value or a vector. Mutually exclusive with max_fraction.

`max_fraction` how far to search in units of relative distance to each query sequence length. Can be a single value or a vector. Mutually exclusive with max_distance.

`mode` The distance metric to use. One of hamming (hm), global (gb) or anchored (an).

`cost_matrix` A custom cost matrix for use with the "global" or "anchored" distance metrics. See details.

`gap_cost` The cost of a gap for use with the "global" or "anchored" distance metrics. See details.

`gap_open_cost` The cost of a gap opening. See details.

`nthreads` The number of threads to use for parallel computation.

`show_progress` Whether to show a progress bar.

Returns - search: The output is a data.frame of all matches with columns "query" and "target". For anchored searches, the output also includes attributes "query_size" and "target_size" which are vectors containing the portion of the query and target sequences that are aligned.

Method `single_gap_search()`**:** A specialized algorithm for searching for sequences allowing at most a single gap within the alignment itself. The mode is always "anchored" and does not penalize end gaps.

Usage - single_gap_search:

```
RadixTree$single_gap_search(
  query,
  max_distance,
  gap_cost = 1L,
  nthreads = 1,
  show_progress = FALSE
)
```

Arguments - single_gap_search:

`query` A character vector of query sequences.

`max_distance` how far to search in units of absolute distance. Can be a single value or a vector. Mutually exclusive with max_fraction.

`gap_cost` The cost of a gap for use with the "global" or "anchored" distance metrics. See details.

`nthreads` The number of threads to use for parallel computation.

`show_progress` Whether to show a progress bar.

Returns - single_gap_search: The output is a data.frame of matches with columns "query", "target" and "distance".

Method `validate()`**:** Validate the tree

Usage - validate:

```
RadixTree$validate()
```

Returns - validate: A logical indicating whether the tree is valid (TRUE) or not (FALSE). This is mostly an internal function for debugging purposes and should always return TRUE.

See Also

https://en.wikipedia.org/wiki/Radix_tree

Examples

```
tree <- RadixTree$new()
tree$insert(c("ACGT", "AAAA"))
tree$erase("AAAA")
tree$search("ACG", max_distance = 1, mode = "levenshtein")
#   query target distance
# 1   ACG   ACGT        1

tree$search("ACG", max_distance = 1, mode = "hamming")
# query    target   distance
# <0 rows> (or 0-length row.names)
```

`split_search` *split_search*

Description

Search for similar sequences based on splitting sequences into left and right sides and searching for matches in each side using a bi-directional anchored alignment.

Usage

```
split_search(
  query,
  target,
  query_split,
  target_split,
  edge_trim = 0L,
  max_distance = 0L,
  ...
)
```

Arguments

query	A character vector of query sequences.
target	A character vector of target sequences.
query_split	index to split query sequence. Should be within (edge_trim, nchar(query)-edge_trim] or -1 to indicate no split.
target_split	index to split target sequence. Should be within (edge_trim, nchar(query)-edge_trim] or -1 to indicate no split.
edge_trim	number of bases to trim from each side of the sequence (default value: 0).
max_distance	how far to search in units of absolute distance. Can be a single value or a vector. Mutually exclusive with max_fraction.
...	additional arguments passed to RadixTree$search

Details

This function is useful for searching for similar sequences that may have variable windows of sequencing (e.g. different 5' and 3' primers) but contain the same core sequence or position. The two split parameters partition the query and target sequences into left and right sides, where left = stri_sub(sequence, edge_trim+1, split) and right = stri_sub(query, split+1, -edge_trim-1).

Value

data.frame with columns query, target, and distance.

Examples

```
# Consider two sets of sequences
# query1   AGACCTAA CCC
# target1 AAGACCTAA CC
# query2   GGGTGTAA CCACCC
# target2   GGTGTAA CCAC
# Despite having different frames, query1 and query2 and clearly
# match to target1 and target2, respectively.
# One could consider splitting based on a common core sequence,
# e.g. a common TAA stop codon.
split_search(query=c(  "AGACCTAACCC", "GGGTGTAACCACCC"),
             target=c("AAGACCTAACC",   "GGTGTAACCAC"),
             query_split=c(8, 8),
             target_split=c(9, 7),
             edge_trim=0,
             max_distance=0)
```

Index