Package 'seqtrie'

June 3, 2026

Title Radix Tree and Trie-Based String Distances

Version 0.4.0

Date 2026-06-02

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>. It also includes a modified version of the Starcode all-pairs search algorithm (Zorita, Cuscó, and Filion 2015) <doi:10.1093/bioinformatics/btv053>.

License GPL-3

Biarch true

Encoding UTF-8

Depends R (>= 3.5.0)

LazyData true

SystemRequirements GNU make, C++17

LinkingTo Rcpp, RcppParallel

Imports Rcpp (>= 0.12.18.3), RcppParallel (>= 5.1.3), R6, S7

Suggests knitr, rmarkdown, pwalign, igraph, ggplot2

VignetteBuilder knitr

RoxygenNote 7.3.3

Copyright This package includes code from the 'span-lite' library owned by Martin Moene under Boost Software License 1.0; see inst/licenses/span-lite-BSL-1.0-LICENSE. This package includes code from the 'ankerl' library owned by Martin Leitner-Ankerl under MIT License. This package includes a modified version of the Starcode all-pairs search algorithm described by Eduard Zorita, Pol Cuscó, and Guillaume J. Filion (2015). See inst/licenses/starcode-GPL-3-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),
Eduard Zorita [ctb] (Starcode algorithm and publication),
Pol Cuscó [ctb] (Starcode algorithm and publication),
Guillaume J. Filion [ctb] (Starcode algorithm and publication)

Maintainer Travers Ching <traversc@gmail.com>

Repository CRAN

Date/Publication 2026-06-03 08:20:08 UTC

Contents

`align_search` *Alignment search*

Description

Alignment search

Usage

```
align_search(
  x,
  query,
  max_distance = NULL,
  max_fraction = NULL,
  mode = "levenshtein",
  cost_matrix = NULL,
  gap_cost = NA_integer_,
  gap_open_cost = NA_integer_,
  lower_triangle = FALSE,
  match_mode = c("all", "best"),
  nthreads = 1L,
  show_progress = FALSE,
  ...
)
```

Arguments

`x`	A seqtrie S7 object.
`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.
`lower_triangle`	If TRUE, only return matches where the query index is greater than the target insertion index.
`match_mode`	Which matches to return for each query. "all" returns all matches within the distance threshold; "best" returns only matches tied for the lowest distance.
`nthreads`	The number of threads to use for parallel computation.
`show_progress`	Whether to show a progress bar.
`...`	Additional method-specific arguments.

Value

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

`covid_cdr3` *Adaptive COVID TCRB CDR3 data*

Description

Unique TCRB CDR3 sequences from Nolan et al. (2020). CDR3s were extracted via IgBLAST. The data are licensed under the Creative Commons Attribution 4.0 International License.

Usage

```
data(covid_cdr3)
```

Format

A character vector of length 133,033.

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
```

`covid_receptors` *Adaptive COVID TCRB nucleotide rearrangements*

Description

Unique TCRB nucleotide rearrangement sequences (the full "TCR Nucleotide Sequence", spanning the V segment, CDR3 junction, and J segment). Every sequence is 87 nucleotides long. Sequences containing the ambiguous base `N` were removed, and duplicates were collapsed. The data are licensed under the Creative Commons Attribution 4.0 International License.

Usage

```
data(covid_receptors)
```

Format

A character vector of length 139,667, each a 87-nucleotide DNA string.

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_receptors)
# All sequences are the same length
table(nchar(covid_receptors)) # 87: 139667
```

---

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 an N by M matrix where N = length(query) and M = length(target). Note: this can take a *really* long time; be careful with input size.

Three distance metrics are supported, based on the form of alignment performed: 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 either the query sequence or the target sequence (but not both) can be unaligned. This type of alignment is sometimes called an "extension" alignment in the 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 non-negative square integer matrix of substitution costs and should include all characters in query and target as column- and rownames. Diagonal entries are usually zero, but positive diagonal entries are allowed. 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 distance metrics are supported, based on the form of alignment performed: 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 either the query sequence or the target sequence (but not both) can be unaligned. This type of alignment is sometimes called an "extension" alignment in the 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 non-negative square integer matrix of substitution costs and should include all characters in query and target as column- and rownames. Diagonal entries are usually zero, but positive diagonal entries are allowed. 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 = NULL,
  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,
  mismatch_cost = 1L
)
```

Arguments

`query`	A character vector of query sequences.
`target`	A character vector of target sequences. If `NULL`, `query` is searched against itself and only pairs where the query index is strictly greater than the target terminal index are returned.
`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 tree implementation to use. One of RadixTree, RadixForest, StarTree, radix_tree, radix_forest, or star_tree (default: RadixTree)
nthreads	The number of threads to use for parallel computation.
show_progress	Whether to show a progress bar.
mismatch_cost	A single positive integer mismatch cost for fixed StarTree classes.

Details

This function finds all sequences in *target* that are within a distance threshold of any sequence in *query*. If `target = NULL`, the tree is built from `query` and each query is searched against that tree while requiring the query index to be strictly greater than the target terminal index. This returns lower-triangle self-pairs. Duplicate query strings are not given special handling; because the underlying tree stores one terminal index per unique sequence, duplicates naturally collapse to the first inserted occurrence. This function uses a radix_tree/RadixTree, radix_forest/RadixForest, or fixed star_tree/StarTree to store *target* sequences. Use `tree_class = "StarTree"` with `mode = "anchored"` for fixed anchored DNA joins.

Three distance metrics are supported, based on the form of alignment performed: 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 either the query sequence or the target sequence (but not both) can be unaligned. This type of alignment is sometimes called an "extension" alignment in the 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 non-negative square integer matrix of substitution costs and should include all characters in query and target as column- and rownames. Diagonal entries are usually zero, but positive diagonal entries are allowed. 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", "target", and "distance". For anchored searches, the output also includes columns "query_size" and "target_size" 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")
```

`erase` *Erase sequences*

Description

Erase sequences

Usage

```
erase(x, sequences)
```

Arguments

`x`	A seqtrie S7 object.
`sequences`	A character vector of sequences.

Value

A logical vector indicating whether each sequence was erased.

`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)
```

---

has_sequence *Test sequence membership*

Description

Test sequence membership

Usage

```
has_sequence(x, query)
```

Arguments

`x`	A seqtrie S7 object.
`query`	A character vector of query sequences.

Value

A logical vector indicating whether each query is present.

---

insert *Insert sequences*

Description

Insert sequences

Usage

```
insert(x, sequences)
```

Arguments

`x`	A seqtrie S7 object.
`sequences`	A character vector of sequences.

Value

A logical vector indicating whether each sequence was inserted.

---

is_valid *Validate trie or forest structure*

Description

Validate trie or forest structure

Usage

```
is_valid(x)
```

Arguments

x	A seqtrie S7 object.

Value

A logical value indicating whether the object is valid.

---

plot_tree *Plot trie or forest structure*

Description

Plot trie or forest structure

Usage

```
plot_tree(x, depth = -1, root_label = "root", plot = TRUE)
```

Arguments

x	A seqtrie S7 object.
depth	The tree depth to plot. If -1, plot the entire tree.
root_label	The label of the root node in the plot.
plot	Whether to create a plot or return the graph data.

Value

A data frame of parent-child relationships or a ggplot2 object.

prefix_search *Prefix search*

Description

Prefix search

Usage

```
prefix_search(x, query)
```

Arguments

x	A seqtrie S7 object.
query	A character vector of query sequences.

Value

A data frame with columns `query` and `target`.

RadixForest *RadixForest*

Description

R6 compatibility wrapper for `radix_forest`

Details

`RadixForest` preserves the original R6 API while delegating implementation to the S7 radix_forest class. New code can use `radix_forest()` with the exported S7 generics directly; existing code using `$insert()`, `$erase()`, `$find()`, `$prefix_search()`, `$search()`, `$to_vector()`, `$to_string()`, `$size()`, `$graph()`, and `$validate()` remains supported.

The RadixForest implementation stores separate radix trees by sequence length. It supports hamming and global/Levenshtein searches, including custom cost matrices and gap penalties. It does not support anchored searches.

Public fields

`forest_pointer` Map of sequence length to RadixTree.

`char_counter_pointer` Character count data for validating input.

Methods

Public methods:

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

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

Usage:

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

Arguments:

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

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

Usage:

```
RadixForest$show()
```

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

Usage:

```
RadixForest$to_string()
```

Returns: A string representation of the forest.

Method `graph()`**:** Plot the forest using igraph and ggplot2.

Usage:

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

Arguments:

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

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

`plot` Whether to create a plot or return the graph data.

Returns: A data frame of parent-child relationships or a ggplot2 object.

Method `to_vector()`**:** Output all stored sequences as a character vector.

Usage:

```
RadixForest$to_vector()
```

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

Method `size()`**:** Output the size of the forest.

Usage:

```
RadixForest$size()
```

Returns: The number of stored sequences.

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

Usage:

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

Arguments:

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

Returns: A logical vector indicating whether each sequence was inserted.

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

Usage:

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

Arguments:

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

Returns: A logical vector indicating whether each sequence was erased.

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

Usage:

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

Arguments:

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

Returns: A logical vector indicating whether each sequence was found.

Method `has_sequence()`**:** Find sequences in the forest.

Usage:

```
RadixForest$has_sequence(query)
```

Arguments:

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

Returns: A logical vector indicating whether each sequence was found.

Method `prefix_search()`**:** Search for sequences in the forest that start with a prefix.

Usage:

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

Arguments:

`query` A character vector of prefixes.

Returns: A data frame with columns `query` and `target`.

Method `search()`**:** Alignment search.

Usage:

```
RadixForest$search(
  query,
  max_distance = NULL,
  max_fraction = NULL,
  mode = "levenshtein",
  cost_matrix = NULL,
  gap_cost = NA_integer_,
  gap_open_cost = NA_integer_,
  lower_triangle = FALSE,
  match_mode = c("all", "best"),
  nthreads = 1,
  show_progress = FALSE
)
```

Arguments:

`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.

`lower_triangle` If TRUE, only return matches where the query index is greater than the target insertion index.

`match_mode` Which matches to return for each query. "all" returns all matches within the distance threshold; "best" returns only matches tied for the lowest distance.

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

`show_progress` Whether to show a progress bar.

Returns: A data frame with columns `query`, `target`, and `distance`.

Method `align_search()`**:** Alignment search.

Usage:

```
RadixForest$align_search(
  query,
  max_distance = NULL,
  max_fraction = NULL,
```

```
  mode = "levenshtein",
  cost_matrix = NULL,
  gap_cost = NA_integer_,
  gap_open_cost = NA_integer_,
  lower_triangle = FALSE,
  match_mode = c("all", "best"),
  nthreads = 1,
  show_progress = FALSE
)
```

Arguments:

`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.

`lower_triangle` If TRUE, only return matches where the query index is greater than the target insertion index.

`match_mode` Which matches to return for each query. "all" returns all matches within the distance threshold; "best" returns only matches tied for the lowest distance.

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

`show_progress` Whether to show a progress bar.

Returns: A data frame with columns `query`, `target`, and `distance`.

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

Usage:

```
RadixForest$validate()
```

Returns: A logical value indicating whether the forest is valid.

Method `is_valid()`**:** Validate the forest.

Usage:

```
RadixForest$is_valid()
```

Returns: A logical value indicating whether the forest is valid.

See Also

radix_forest, radix_tree

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

R6 compatibility wrapper for `radix_tree`

Details

`RadixTree` preserves the original R6 API while delegating implementation to the S7 radix_tree class. New code can use `radix_tree()` with the exported S7 generics directly; existing code using `$insert()`, `$erase()`, `$find()`, `$prefix_search()`, `$search()`, `$single_gap_search()`, `$to_vector()`, `$to_string()`, `$size()`, `$graph()`, and `$validate()` remains supported.

The RadixTree implementation stores sequences in a trie and searches for similar sequences with hamming, global/Levenshtein, anchored, or single-gap alignment metrics.

Three distance metrics are supported, based on the form of alignment performed: 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 either the query sequence or the target sequence (but not both) can be unaligned. This type of alignment is sometimes called an "extension" alignment in the 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 non-negative square integer matrix of substitution costs and should include all characters in query and target as column- and rownames. Diagonal entries are usually zero, but positive diagonal entries are allowed. 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 validating input.

Methods

Public methods:

- `RadixTree$new()`
- `RadixTree$show()`
- `RadixTree$to_string()`
- `RadixTree$graph()`
- `RadixTree$to_vector()`
- `RadixTree$size()`
- `RadixTree$insert()`
- `RadixTree$erase()`
- `RadixTree$find()`
- `RadixTree$has_sequence()`
- `RadixTree$prefix_search()`
- `RadixTree$search()`
- `RadixTree$align_search()`
- `RadixTree$single_gap_search()`
- `RadixTree$validate()`
- `RadixTree$is_valid()`

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

Usage:

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

Arguments:

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

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

Usage:

```
RadixTree$show()
```

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

Usage:

```
RadixTree$to_string()
```

Returns: A string representation of the tree.

Method `graph()`**:** Plot the tree using igraph and ggplot2.

Usage:

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

Arguments:

`depth` The tree depth to plot. If -1, plot the entire tree.
`root_label` The label of the root node in the plot.
`plot` Whether to create a plot or return the graph data.

Returns: A data frame of parent-child relationships or a ggplot2 object.

Method `to_vector()`**:** Output all stored sequences as a character vector.

Usage:

```
RadixTree$to_vector()
```

Returns: A character vector of all sequences contained in the tree.

Method `size()`**:** Output the size of the tree.

Usage:

```
RadixTree$size()
```

Returns: The number of stored sequences.

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

Usage:

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

Arguments:

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

Returns: A logical vector indicating whether each sequence was inserted.

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

Usage:

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

Arguments:

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

Returns: A logical vector indicating whether each sequence was erased.

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

Usage:

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

Arguments:

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

Returns: A logical vector indicating whether each sequence was found.

Method `has_sequence()`**:** Find sequences in the tree.

Usage:

```
RadixTree$has_sequence(query)
```

Arguments:

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

Returns: A logical vector indicating whether each sequence was found.

Method `prefix_search()`**:** Search for sequences in the tree that start with a prefix.

Usage:

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

Arguments:

`query` A character vector of prefixes.

Returns: A data frame with columns `query` and `target`.

Method `search()`**:** Alignment search.

Usage:

```
RadixTree$search(
  query,
  max_distance = NULL,
  max_fraction = NULL,
  mode = "levenshtein",
  cost_matrix = NULL,
  gap_cost = NA_integer_,
  gap_open_cost = NA_integer_,
  lower_triangle = FALSE,
  match_mode = c("all", "best"),
  nthreads = 1,
  show_progress = FALSE
)
```

Arguments:

`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.

`lower_triangle` If TRUE, only return matches where the query index is greater than the target insertion index.

`match_mode` Which matches to return for each query. "all" returns all matches within the distance threshold; "best" returns only matches tied for the lowest distance.

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

`show_progress` Whether to show a progress bar.

Returns: A data frame with columns `query`, `target`, and `distance`.

Method `align_search()`**:** Alignment search.

Usage:

```
RadixTree$align_search(
  query,
  max_distance = NULL,
  max_fraction = NULL,
  mode = "levenshtein",
  cost_matrix = NULL,
  gap_cost = NA_integer_,
  gap_open_cost = NA_integer_,
  lower_triangle = FALSE,
  match_mode = c("all", "best"),
  nthreads = 1,
  show_progress = FALSE
)
```

Arguments:

`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.

`lower_triangle` If TRUE, only return matches where the query index is greater than the target insertion index.

`match_mode` Which matches to return for each query. "all" returns all matches within the distance threshold; "best" returns only matches tied for the lowest distance.

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

`show_progress` Whether to show a progress bar.

Returns: A data frame with columns `query`, `target`, and `distance`.

Method `single_gap_search()`**:** A specialized anchored search allowing at most one internal gap.

Usage:

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

Arguments:

`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: A data frame with columns `query`, `target`, and `distance`.

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

Usage:

```
RadixTree$validate()
```

Returns: A logical value indicating whether the tree is valid.

Method `is_valid()`**:** Validate the tree.

Usage:

```
RadixTree$is_valid()
```

Returns: A logical value indicating whether the tree is valid.

See Also

radix_tree, 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)
```

`radix_forest` *Radix forest*

Description

`radix_forest()` constructs a mutable S7 wrapper around the seqtrie C++ radix forest implementation. It partitions sequences by length and supports hamming and global/Levenshtein searches.

Usage

```
radix_forest(sequences = NULL)
```

Arguments

sequences	Optional character vector of sequences to insert.

Value

A radix_forest object.

See Also

RadixForest for the R6 compatibility wrapper.

Examples

```
forest <- radix_forest(c("ACGT", "AAAA"))
align_search(forest, "ACG", max_distance = 1, mode = "levenshtein")
```

---

radix_tree *Radix tree*

Description

radix_tree() constructs a mutable S7 wrapper around the seqtrie C++ radix tree implementation. It supports hamming, global/Levenshtein, anchored, and single-gap searches.

Usage

```
radix_tree(sequences = NULL)
```

Arguments

sequences	Optional character vector of sequences to insert.

Value

A radix_tree object.

See Also

RadixTree for the R6 compatibility wrapper.

Examples

```
tree <- radix_tree(c("ACGT", "AAAA"))
align_search(tree, "ACG", max_distance = 1, mode = "levenshtein")
```

result *Return a fixed-tree self-similarity result*

Description

Return a fixed-tree self-similarity result

Usage

```
result(x)
```

Arguments

x	A seqtrie S7 object.

Value

A data frame with columns `query`, `target`, and `distance`. Anchored mode also includes `query_size` and `target_size`.

single_gap_search *Single-gap alignment search*

Description

Single-gap alignment search

Usage

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

Arguments

x	A `radix_tree` object.
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.

Value

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

---

size *Count stored sequences*

Description

Count stored sequences

Usage

```
size(x)
```

Arguments

x	A seqtrie S7 object.

Value

The number of stored sequences.

---

split_search *split_search*

Description

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

Usage

```
split_search(
  query,
  target,
  query_split,
  target_split,
  edge_trim = 0L,
  max_distance = 0L,
  max_fraction = NULL,
  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.
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(target)-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.
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.
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 is useful for searching for similar sequences that may have variable sequencing windows (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 = rev(substr(sequence, edge_trim+1, split)) and right = substr(sequence, split+1, nchar(sequence)-edge_trim).

Value

A 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 can 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)
```

`StarTree` *StarTree*

Description

R6 compatibility wrapper for `star_tree`

Details

`StarTree` is a fixed DNA-only tree using a modified version of the Starcode all-pairs search strategy, adapted to operate over a radix trie rather than being a direct reimplementation. It supports global/Levenshtein, anchored, and Hamming alignment modes. Unlike RadixTree, all sequences and alignment parameters are supplied at construction time and the self-similarity join runs immediately. The tree does not support insertion or deletion after construction.

Use `$result()` to retrieve the construction-time self-similarity join, and `$align_search()` or `$search()` to search additional query sequences against the fixed target set using the same `mode`, `max_distance`, `mismatch_cost`, and `gap_cost`.

The algorithm is based on Starcode (Zorita, Cuscó, and Filion 2015) doi:10.1093/bioinformatics/btv053.

Public fields

`tree_pointer` External pointer to the fixed-tree C++ object.

`mode` Fixed alignment mode.

`max_distance` Fixed distance threshold.

`mismatch_cost` Fixed mismatch cost.

`gap_cost` Fixed gap cost.

`nthreads` Fixed thread count.

`show_progress` Fixed progress flag.

Methods

Public methods:

- `StarTree$new()`
- `StarTree$to_vector()`
- `StarTree$size()`
- `StarTree$result()`
- `StarTree$search()`
- `StarTree$align_search()`

Method `new()`**:** Create a new StarTree object.

Usage:

```
StarTree$new(
  sequences,
  max_distance,
  mode = "levenshtein",
  mismatch_cost = 1L,
  gap_cost = 1L,
  nthreads = 1L,
  show_progress = FALSE
)
```

Arguments:

`sequences` A required character vector of DNA sequences.

`max_distance` A single non-negative integer distance threshold.

`mode` Alignment mode: global/Levenshtein, anchored, or hamming.

`mismatch_cost` A single positive integer mismatch cost.

`gap_cost` A single positive integer gap cost.

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

`show_progress` Whether to show a progress bar.

Method `to_vector()`**:** Output all stored unique sequences as a character vector.

Usage:

```
StarTree$to_vector()
```

Returns: A character vector of all sequences contained in the tree.

Method `size()`**:** Output the size of the tree.

Usage:

```
StarTree$size()
```

Returns: The number of stored unique sequences.

Method `result()`**:** Return the construction-time self-similarity join.

Usage:

```
StarTree$result()
```

Returns: A data frame with columns `query`, `target`, and `distance`. Anchored mode also includes `query_size` and `target_size`.

Method `search()`**:** Search additional query sequences against the fixed tree.

Usage:

```
StarTree$search(
  query,
  nthreads = self$nthreads,
  show_progress = self$show_progress
)
```

Arguments:

`query` A character vector of query sequences.

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

`show_progress` Whether to show a progress bar.

Returns: A data frame with columns `query`, `target`, and `distance`. Anchored mode also includes `query_size` and `target_size`.

Method `align_search()`**:** Search additional query sequences against the fixed tree.

Usage:

```
StarTree$align_search(
  query,
  nthreads = self$nthreads,
  show_progress = self$show_progress
)
```

Arguments:

`query` A character vector of query sequences.

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

`show_progress` Whether to show a progress bar.

Returns: A data frame with columns `query`, `target`, and `distance`. Anchored mode also includes `query_size` and `target_size`.

See Also

`align_search()`

Examples

```
tree <- StarTree$new(c("ACGT", "ACGA", "AAAA"), max_distance = 1)
tree$result()
tree$search(c("ACGT", "AAAT"))

anchored <- StarTree$new(c("ACGT", "ACG", "AAAA"), max_distance = 1,
                         mode = "anchored")
anchored$result()
```

star_tree *Starcode-style fixed tree*

Description

`star_tree()` constructs a fixed DNA-only tree using a modified version of the Starcode all-pairs search strategy, adapted to operate over a radix trie. The input sequences, alignment `mode`, `max_distance`, `mismatch_cost`, and `gap_cost` are fixed at construction, and the self-similarity join runs immediately. Use `result()` to retrieve that self-join, and `align_search()` to search additional query sequences against the fixed target set.

Usage

```
star_tree(
  sequences,
  max_distance,
  mode = "levenshtein",
  mismatch_cost = 1L,
  gap_cost = 1L,
  nthreads = 1L,
  show_progress = FALSE
)
```

Arguments

`sequences`	A required character vector of DNA sequences.
`max_distance`	A single non-negative integer distance threshold.
`mode`	Alignment mode: global/Levenshtein, anchored, or hamming.
`mismatch_cost`	A single positive integer mismatch cost.
`gap_cost`	A single positive integer gap cost.
`nthreads`	The number of threads to use for parallel computation.
`show_progress`	Whether to show a progress bar.

Details

StarTree supports global/Levenshtein-style, anchored, and Hamming DNA alignment. It accepts `A`, `C`, `G`, `T`, and `N` in either case; sequences are stored and returned in uppercase. `N` is treated as a regular ambiguous base with mismatch cost, not as a wildcard. Custom substitution matrices, affine gaps, insertion, and deletion are not supported.

Hamming mode (`mode = "hamming"`) is substitution-only: only equal-length sequences can match, and `max_distance` is the maximum number of mismatching positions (unit substitution cost; `mismatch_cost` and `gap_cost` do not apply). It is typically much faster than global mode for the same data.

For `star_tree` objects, `align_search()` only accepts `query`, `nthreads`, and `show_progress`; all alignment parameters are fixed here at construction. Anchored-mode results also include `query_size` and `target_size`.

The algorithm is based on Starcode (Zorita, Cuscó, and Filion 2015) doi:10.1093/bioinformatics/btv053.

Value

A `star_tree` object.

See Also

`align_search()`

Examples

```
tree <- star_tree(c("ACGT", "ACGA", "AAAA"), max_distance = 1)
result(tree)
align_search(tree, c("ACGT", "AAAT"))

anchored <- star_tree(c("ACGT", "ACG", "AAAA"), max_distance = 1,
                      mode = "anchored")
result(anchored)

hamming <- star_tree(c("ACGT", "ACGA", "TCGT"), max_distance = 1,
                     mode = "hamming")
result(hamming)
```

---

to_string *Convert a trie or forest to a string*

Description

Convert a trie or forest to a string

Usage

```
to_string(x)
```

Arguments

x	A seqtrie S7 object.

Value

A string representation of the trie or forest.

`to_vector` *Convert sequences to a character vector*

Description

Convert sequences to a character vector

Usage

```
to_vector(x)
```

Arguments

`x`	A seqtrie S7 object.

Value

A character vector of stored sequences.

Index