# edist

*Release 1.2.0*

**May 31, 2021**

# Contents:

This library contains several edit distance and alignment algorithms for sequences and trees of arbitrary node type. Additionally, this library contains multiple backtracing mechanisms for every algorithm in order to facilitate more detailed interpretation and subsequent processing.

In more detail, this library currently features the following algorithms.

- Sequence Edit Distance (sed; Levenshtein, 1965)
- Dynamic Time Warping (sed; Vintsyuk, 1968)
- Affine edit distance (aed; Gotoh, 1982)
- Tree Edit Distance (ted; Zhang and Shasha, 1989)
- Constrained Unordered Tree Edit Distance (uted; Zhang and Shasha, 1996)
- Set edit distance (seted; unpublished)

As well as the following meta-algorithms:

- Algebraic Dynamic Programming (adp; according to the dissertation Paaßen, 2019)
- Embedding Edit Distance Learning (bedl; Paaßen et al., 2018)

If you intend to use this library in academic work, please cite the paper:

- **Paaßen, B., Mokbel, B., & Hammer, B. (2015). A Toolbox for Adaptive Sequence** Dissimilarity Measures for Intelligent Tutoring Systems. In O. C. Santos, J. G. Boticario, C. Romero, M. Pechenizkiy, A. Merceron, P. Mitros, J. M. Luna, et al. (Eds.), Proceedings of the 8th International Conference on Educational Data Mining (pp. 632-632). International Educational Datamining Society.

Please consult the project website for more detailed information about the project.

# CHAPTER 1

## Algebraic Dynamic Programming

Implements algebraic dynamic programming for edit distances.

**class** edist.adp.**Grammar**

Models an ADP grammar, consisting of a starting nonterminal, a set of accepting nonterminals, a set of permitted edit operations and a set of rules of the form

*A -> delta B*

where *A* and *B* are nonterminals and *delta* is an edit operation, either a replacement, a deletion, or an insertion. We define the set of possible edit scripts permitted by a given grammar inductively as all scripts that can be produced by starting from the self._start, replacing the current nonterminal with the right-hand-side of a matching rule arbitrary many times, and then deleting the current nonterminal if it is accepting.

**_nonterminals**

A list of nonterminals of this grammar. Defaults to the union of accepting and start.

**Type** list (default = _accepting + {start})

**_start**

The starting nonterminal, which should be in _nonterminals.

**Type** str

**_accepting**

A list of accepting nonterminals, all of which should be in self._nonterminals.

**Type** list

**_reps**

A list of names of possible replacement operations. Defaults to an empty list.

**Type** list (default = [])

**_dels**

A list of names of possible deletion operations. Defaults to an empty list.

**Type** list (default = [])

**_inss**
A list of names of possible insertion operations. Defaults to an empty list.

**Type** list (default = [])

**_rules**
A mapping of nonterminal symbols to RuleEntries. Refer to The documentation above for more details on RuleEntries. Defaults to an empty map.

**Type** dictionary (default = {})

**adjacency_lists**
Returns the adjacency list format of this grammar.

**Returns**

- **start_idx** (*int*) – The index of the starting nonterminal symbol.
- **accpt_idxs** (*list*) – A list of indices for all accepting nonterminals.
- **rep_adj** (*list*) – An adjacency list covering all replacement operations, i.e. a list where the *i*-th entry contains all rules of the form *A -> rep B*, where *A* is the *i*-th nonterminal and the right-hand-sides are represented as tuples of operation indices and nonterminal indices.
- **del_adj** (*list*) – An adjacency list covering all deletion operations, i.e. a list where the *i*-th entry contains all rules of the form *A -> del B*, where *A* is the *i*-th nonterminal and the right-hand-sides are represented as tuples of operation indices and nonterminal indices.
- **ins_adj** (*list*) – An adjacency list covering all insertion operations, i.e. a list where the *i*-th entry contains all rules of the form *A -> ins B*, where *A* is the *i*-th nonterminal and the right-hand-sides are represented as tuples of operation indices and nonterminal indices.

**append_deletion**
Appends a rule to this grammar for a deletion operation, i.e. a rule of the form *A -> del B*.

**Parameters**

- **source** (*str*) – The left-hand-side nonterminal *A* of the new rule. If not in self._nonterminals yet, it is appended automatically.
- **target** (*str*) – The right-hand-side nonterminal *B* of the new rule. If not in self._nonterminals yet, it is appended automatically.
- **operation** (*str*) – The name of the deletion operation del. If not in self._dels yet, it is appended automatically.

**append_insertion**
Appends a rule to this grammar for a deletion operation, i.e. a rule of the form *A -> ins B*.

**Parameters**

- **source** (*str*) – The left-hand-side nonterminal *A* of the new rule. If not in self._nonterminals yet, it is appended automatically.
- **target** (*str*) – The right-hand-side nonterminal *B* of the new rule. If not in self._nonterminals yet, it is appended automatically.
- **operation** (*str*) – The name of the insertion operation ins. If not in self._inss yet, it is appended automatically.

**append_replacement**
Appends a rule to this grammar for a replacement operation, i.e. a rule of the form *A -> rep B*.

**Parameters**

- **source** (*str*) – The left-hand-side nonterminal *A* of the new rule. If not in self._nonterminals yet, it is appended automatically.
- **target** (*str*) – The right-hand-side nonterminal *B* of the new rule. If not in self._nonterminals yet, it is appended automatically.
- **operation** (*str*) – The name of the replacement operation rep. If not in self._reps yet, it is appended automatically.

**inverse_adjacency_lists**

Returns the inverse adjacency list format of this grammar.

**Returns**

- **start_idx** (*int*) – The index of the starting nonterminal symbol.
- **accpt_idxs** (*list*) – A list of indices for all accepting nonterminals.
- **rep_adj** (*list*) – An adjacency list covering all replacement operations, i.e. a list where the *i*-th entry contains all rules of the form *A -> rep B*, where *B* is the *i*-th nonterminal and (*A*, *rep*) is represented as a tuple tuples of operation index and nonterminal index.
- **del_adj** (*list*) – An adjacency list covering all deletion operations, i.e. a list where the *i*-th entry contains all rules of the form *A -> del B*, where *B* is the *i*-th nonterminal and (*A*, *del*) is represented as a tuple tuples of operation index and nonterminal index.
- **ins_adj** (*list*) – An adjacency list covering all insertion operations, i.e. a list where the *i*-th entry contains all rules of the form *A -> ins B*, where *B* is the *i*-th nonterminal and (*A*, *ins*) is represented as a tuple tuples of operation index and nonterminal index.

**nonterminals**

Returns the list of nonterminals of this grammar.

**Returns** **nonts** – the list of nonterminals of this grammar.

**Return type** list

**size**

Returns the number of nonterminals in this grammar.

**Returns** **size** – the number of nonterminals in this grammar.

**Return type** int

**start**

Returns the starting nonterminal of this grammar.

**Returns** **start** – the starting nonterminal of this grammar.

**Return type** str

**validate**

Ensures that this grammar is compatible with the given algebra deltas.

**Parameters** **deltas** (*dictionary*) – An algebra, i.e. a mapping from operation names to distance functions.

**Raises** `ValueError` – If any of the operations of this grammar is not supported by the given algebra.

**class** edist.adp.**RuleEntry**

Models the set of all grammar rules for a single nonterminal symbol in an ADP grammar.

**_reps**

A list of possible replacements, stored as tuples of the form

(rep, B), where rep is the name of a replacement operation and B is the nonterminal we obtain after applying rep.

**Type** list

**_dels**

A list of possible deletions, stored as tuples of the form

(del, B), where del is the name of a deletion operation and B is the nonterminal we obtain after applying del.

**Type** list

**_inss**

A list of possible insertions, stored as tuples of the form

(ins, B), where ins is the name of a insertion operation and B is the nonterminal we obtain after applying ins.

**Type** list

edist.adp.**backtrace**()

Computes a co-optimal alignment between the two input sequences x and y, given the given ADP grammar and algebra. This mechanism is deterministic and will always prefer replacements over other options.

**Parameters**

- **x** (*list*) – A list of objects.
- **y** (*list*) – Another list of objects.
- **grammar** (*class adp.Grammar*) – An ADP grammar. Refer to the documentation above for more information.
- **deltas** (*dictionary*) – An algebra, i.e. a mapping from operation names to distance functions OR a single distance function if the grammar supports only a single replacement, deletion, and insertion operation.

**Returns** **alignment** – a co-optimal alignment between x and y.

**Return type** class alignment.Alignment

edist.adp.**backtrace_matrix**()

Computes three tensors, P_rep, P_del, and P_ins, which summarize all co-optimal alignments between x and y.

In particular, P_rep[k, i, j] specifies the fraction of co-optimal alignments in which node x[i] has been replaced with node y[j] using operation grammar._reps[k]. Accordingly, P_del[k, i] specifies the fraction of co-optimal alignments in which x[i] has been deleted using operation grammar._dels[k], and P_ins[k, j] specifies the fraction of co-optimal alignments in which y[j] has been inserted using operation grammar._inss[k].

**Parameters**

- **x** (*list*) – A list of objects.
- **y** (*list*) – Another list of objects.
- **grammar** (*class adp.Grammar*) – An ADP grammar. Refer to the documentation above for more information.
- **deltas** (*dictionary*) – An algebra, i.e. a mapping from operation names to distance functions OR a single distance function if the grammar supports only a single replacement, deletion, and insertion operation.

**Returns**

- **P_rep** (*array_like*) – a tensor where P_rep[k, i, j] specifies the fraction of co-optimal alignments in which node x[i] has been replaced with node y[j] using operation grammar._reps[k].
- **P_del** (*array_like*) – a matrix where P_del[k, i] specifies the fraction of co-optimal alignments in which x[i] has been deleted using operation grammar._dels[k].
- **P_ins** (*array_like*) – a matrix where P_ins[k, j] specifies the fraction of co-optimal alignments in which y[j] has been inserted using operation grammar._inss[k].
- **k** (*int*) – the number of co-optimal alignments.

edist.adp.**backtrace_stochastic**()

Computes a co-optimal alignment between the two input sequences x and y, given the given ADP grammar and algebra. This mechanism is stochastic and will return a random alignment.

Note that the randomness does _not_ produce a uniform distribution over all co-optimal alignments because random choices at the start of the alignment process dominate. If you wish to characterize the overall distribution accurately, use backtrace_matrix instead.

**Parameters**

- **x** (*list*) – A list of objects.
- **y** (*list*) – Another list of objects.
- **grammar** (*class adp.Grammar*) – An ADP grammar. Refer to the documentation above for more information.
- **deltas** (*dictionary*) – An algebra, i.e. a mapping from operation names to distance functions OR a single distance function if the grammar supports only a single replacement, deletion, and insertion operation.

**Returns** **alignment** – a co-optimal alignment between x and y.

**Return type** class alignment.Alignment

edist.adp.**edit_distance**()

Computes the edit distance between two sequences x and y, based on the given ADP grammar and the given algebra.

**Parameters**

- **x** (*list*) – A list of objects.
- **y** (*list*) – Another list of objects.
- **grammar** (*class adp.Grammar*) – An ADP grammar. Refer to the documentation above for more information.
- **deltas** (*dictionary*) – An algebra, i.e. a mapping from operation names to distance functions OR a single distance function if the grammar supports only a single replacement, deletion, and insertion operation.

**Returns** **d** – The edit distance between x and y.

**Return type** float

edist.adp.**string_to_index_list**()

Converts a list of objects to an index list, given a index mapping.

**Parameters**

- **lst** (*list*) – A list of objects [*x_1*, ..., *x_m*].
- **dct** (*dictionary*) – A mapping from objects to indices.

**Returns** **idx_list** – The list [dict[*x_1*], . . . , dict[*x_m*]]

**Return type** list

edist.adp.**string_to_index_map**()

Inverts a list of objects, i.e. converts a list of objects to a map from objects to indices in the list.

**Parameters** **lst** (*list*) – A list of objects [*x_1*, . . . , *x_m*].

**Returns** **dct** – A map where dct[x_i] = i for all i.

**Return type** dictionary

edist.adp.**string_to_index_tuple_list**()

Converts a list of tuples to a list of tuple-indices.

**Parameters**

- **lst** (*list*) – A list of tuples [(*x_1*, *y_1*), . . . , (*x_m*, *y_m*)].
- **op_dct** (*dictionary*) – A mapping from x-objects to indices.
- **nont_dct** (*dictionary*) – A mapping from y-objects to indices.

**Returns** **idx_list** – The list [(op_dct[*x_1*], nont_dct[*y_1*]), . . . , (op_dct[*x_m*], nont_dct[*y_m*])]

**Return type** list

# CHAPTER 2

## Affine Edit Distance

Implements a sequence edit distance with affine gap costs using ADP.

**class** edist.aed.**AffineAlgebra**(*rep=None*, *gap=1.0*, *skip=0.5*)

This is a class to efficiently store an algebra for the affine edit distance grammar in a pickleable format.

**_rep**

A function for replacement costs, i.e. _rep(x, y) is the cost of replacing x with y.

**Type** function (default = Kronecker distance)

**_gap**

A function for deletion/insertion costs, i.e. _gap(x) is the cost of deleting/inserting x.

**Type** function (default = constant function with 1.0)

**_gap_cost**

a constant cost for deletions/insertions.

**Type** float (default = 1.0)

**_skip**

A function for deletion/insertion extension costs, i.e. _skip(x) is the cost of skip-deleting/-inserting x.

**Type** function (default = constant function with 0.5)

**_skip_cost**

a constant cost for deletion/insertion extensions.

**Type** float (default = 0.5)

edist.aed.**aed**(*x*, *y*, *rep=None*, *gap=1.0*, *skip=0.5*)

Computes the affine edit distance using algebraic dynamic programming.

**Parameters**

- **x** (*list*) – A list-like object.
- **y** (*list*) – Another list-like object.

- **rep** (*function (default = Kronecker delta)*) – A function with two arguments, computing the cost for replacing the first with the second OR an AffineAlgebra object, in which case the remaining aguments will be ignored. Defaults to the Kronecker distance.
- **gap** (*function or float (default = 1.0)*) – A function with two arguments, computing the cost for deleting the first or inserting the second OR a number defining a constant cost. Defaults to 1.
- **skip** (*function or float (default = 0.5)*) – A function with two arguments, computing the cost for deleting the first or inserting the second for gap extensions OR a number defining a constant cost. Defaults to 0.5.

**Returns** **d** – The affine edit distance between x and y.

**Return type** float

edist.aed.**aed_backtrace**(*x*, *y*, *rep=None*, *gap=1.0*, *skip=0.5*)

Computes the backtrace of the affine edit distance using algebraic dynamic programming.

**Parameters**

- **x** (*list*) – A list-like object.
- **y** (*list*) – Another list-like object.
- **rep** (*function (default = Kronecker delta)*) – A function with two arguments, computing the cost for replacing the first with the second OR an AffineAlgebra object, in which case the remaining aguments will be ignored. Defaults to the Kronecker distance.
- **gap** (*function or float (default = 1.0)*) – A function with two arguments, computing the cost for deleting the first or inserting the second OR a number defining a constant cost. Defaults to 1.
- **skip** (*function or float (default = 0.5)*) – A function with two arguments, computing the cost for deleting the first or inserting the second for gap extensions OR a number defining a constant cost. Defaults to 0.5.

**Returns** **alignment** – A co-optimal alignment between x and y according to the affine edit distance.

**Return type** class alignment.Alignment

edist.aed.**aed_backtrace_matrix**(*x*, *y*, *rep=None*, *gap=1.0*, *skip=0.5*)

Computes the backtrace matrix P of the affine edit distance using algebraic dynamic programming.

In particular, P[i, j] contains the probability of node i being replaced with node j in a co-optimal alignment. The last two columns contain deletion and deletion-extension probabilities, the last two rows contains insertion and insertion-extension probabilities.

**Parameters**

- **x** (*list*) – A list-like object.
- **y** (*list*) – Another list-like object.
- **rep** (*function (default = Kronecker distance)*) – A function with two arguments, computing the cost for replacing the first with the second OR an AffineAlgebra object, in which case the remaining aguments will be ignored. Defaults to the Kronecker distance.
- **gap** (*function or float (default = 1.0)*) – A function with two arguments, computing the cost for deleting the first or inserting the second OR a number defining a constant cost. Defaults to 1.

- **skip** (*function or float (default = 0.5)*) – A function with two arguments, computing the cost for deleting the first or inserting the second for gap extensions OR a number defining a constant cost. Defaults to 0.5.

**Returns**

- **P** (*array_like*) – A len(x) + 2 x len(y) + 2 matrix where P[i, j] contains the probability of node i being replaced with node j in a co-optimal alignment. The last two columns contain deletion and deletion-extension probabilities, the last two rows contains insertion and insertion-extension probabilities.
- **k** (*int*) – The number of co-optimal alignments.

edist.aed.**aed_backtrace_stochastic**(*x*, *y*, *rep=None*, *gap=1.0*, *skip=0.5*)

Computes the backtrace of the affine edit distance using algebraic dynamic programming stochastically.

Note that the randomness does _not_ produce a uniform distribution over all co-optimal alignments because random choices at the start of the alignment process dominate. If you wish to characterize the overall distribution accurately, use aed_backtrace_matrix instead.

**Parameters**

- **x** (*list*) – A list-like object.
- **y** (*list*) – Another list-like object.
- **rep** (*function (default = Kronecker delta)*) – A function with two arguments, computing the cost for replacing the first with the second OR an AffineAlgebra object, in which case the remaining aguments will be ignored. Defaults to the Kronecker distance.
- **gap** (*function or float (default = 1.0)*) – A function with two arguments, computing the cost for deleting the first or inserting the second OR a number defining a constant cost. Defaults to 1.
- **skip** (*function or float (default = 0.5)*) – A function with two arguments, computing the cost for deleting the first or inserting the second for gap extensions OR a number defining a constant cost. Defaults to 0.5.

**Returns** **alignment** – A co-optimal alignment between x and y according to the affine edit distance.

**Return type** class alignment.Alignment

CHAPTER 3

# Alignment

Implements an alignment between two sequences or trees.

**class** edist.alignment.**Alignment**

Models a list of tuples. Note that, by convention, an alignment between sequences only permits tuples in lexicographically ascending order, i.e. an alignment of nothing to 0, 0 to 1, and 1 to 2, should be stored in that order and not as [(1, 2), (0, 1), (-1, 0)], for example. The same holds for tree alignments, with the additional requirement that aligned indices must respect the structure of the tree, i.e. if i is aligned to j and i2 to j2, then i can only be a parent of i2 if j is a parent of j2 (and vice versa).

**append_tuple**(*left*, *right*, *op=None*)

Appends a new tuple to the current Alignment.

**Parameters**

- **left** (*int*) – the left index.
- **right** (*int*) – the right index.
- **op** (*str (default = None)*) – a name for the underlying edit operation.

**cost**(*x*, *y*, *deltas*)

Computes the cost of this trace. This is equivalent to the sum of the cost of all tuples in this trace.

**Parameters**

- **x** (*list*) – A symbol list for the left indices.
- **y** (*list*) – A symbol list for the right indices.
- **deltas** (*function or dictionary*) – The cost function delta mapping pairs of elements to replacement/deletion/insertion costs OR A map which contains for any operation name such a function.

**Returns** **cost** – The cost assigned by deltas to this Alignment.

**Return type** float

**render**(*x*, *y*, *deltas=None*)

Represents this trace as a string, showing the left and right indices in addition to the respective labels in x

and y, and in addition to the tuple cost. This is equivalent as to calling 'render' on all tuples in this trace and joining the resulting strings with newlines.

**Parameters**

- **x** (*list*) – A symbol list for the left indices.
- **y** (*list*) – A symbol list for the right indices.
- **deltas** (*function or dictionary (default = None)*) – The cost function delta mapping pairs of elements to replacement/deletion/insertion costs OR A map which contains for any operation name such a function. If provided, the cost for any operation is rendered as well.

**Returns** **repr** – A string representing this Alignment.

**Return type** str

**class** edist.alignment.**Tuple**(*name*, *left*, *right*)

Models a single alignment entry with an edit operation name, a left index, and a right index.

**_name**

The name of the corresponding edit operation.

**Type** str

**_left**

The index of the aligned object on the left or -1 if no such object exists.

**Type** int

**_right**

The index of the aligned object on the right or -1 if no such object exists.

**Type** int

**cost**(*x*, *y*, *deltas*)

Computes the cost of the current edit tuple.

**Parameters**

- **x** (*list*) – A symbol list for the left indices.
- **y** (*list*) – A symbol list for the right indices.
- **deltas** (*function or dictionary*) – The cost function delta mapping pairs of elements to replacement/deletion/insertion costs OR A map which contains for any operation name such a function.

**Returns** **cost** – The cost assigned by deltas to this tuple.

**Return type** float

**render**(*x*, *y*, *deltas=None*)

Represents an tuple as a string, showing the left and right indices in addition to the respective labels in x and y, and in addition to the tuple cost.

**Parameters**

- **x** (*list*) – A symbol list for the left indices.
- **y** (*list*) – A symbol list for the right indices.
- **deltas** (*function or dictionary (default = None)*) – The cost function delta mapping pairs of elements to replacement/deletion/insertion costs OR A map which

contains for any operation name such a function. If provided, the cost for any operation is rendered as well.

**Returns** **repr** – A string representing this tuple.

**Return type** str

# CHAPTER 4

## Embedding Edit Distance Learning

Implements embedding edit distance learning as described in the paper

Paaßen, B., Gallicchio, C., Micheli, A., and Hammer, B. (2018). Tree Edit Distance Learning via Adaptive Symbol Embeddings. Proceedings of the 35th International Conference on Machine Learning (ICML 2018). URL: http://proceedings.mlr.press/v80/paassen18a.html

**class** edist.bedl.**BEDL**(*K*, *T=5*, *phi=None*, *phi_grad=None*, *distance=None*, *distance_backtrace=None*)

Implements the embedding edit distance learning (BEDL) scheme for metric learning on edit distances.

In more detail, this learns a median generalized learning vector quantization (MGLVQ) classifier on the data and an embedding of the input symbols which yields an edit distance that makes classification with this classifier easier.

**K**

The number of prototypes for the MGLVQ classifier.

**Type** int

**T**

The number of learning epochs we use at most. Defaults to 5.

**Type** int

**phi**

A squashing function to post-process each error term. Defaults to the identity.

**Type** function (default = identity)

**phi_grad**

The gradient function corresponding to phi.

**Type** function (default = one)

**distance**

The edit distance function that shall be learned. Defaults to the sequence edit distance sed.sed.

**Type** function (default = sed.sed)

**distance_backtrace**
The matrix backtracing function for the distance. Defaults to sed.sed_backtrace_matrix. Note that this currently does NOT support ADP because ADP returns a different backtracing format.

**Type** function (default = sed.sed_backtrace_matrix)

**_classifier**
The learned MGLVQ classifier model.

**Type** class proto_dist_ml.MGLVQ

**_idx**
A mapping from alphabet to indices.

**Type** dictionary

**_embedding**
A len(alphabet) x len(alphabet) - 1 embedding matrix for all symbols in the alphabet.

**Type** array_like

**_delta_obj**
An internal object to make storing of the delta function more efficient.

**Type** class bedl.EmbeddingDelta

**_delta**
The learned delta function.

**Type** function

**fit**(*X*, *y*)
Trains a BEDL model on the given input data.

In more detail, we iterate the following steps in each training epoch:

1. We re-compute the distance matrix.
2. We adapt the MGLVQ classifier to the current distance matrix.
3. We compute the matrix backtraces for the data-to-prototype distances.
4. We optimize the embedding for the current data-to-prototype distances.

For more details, please refer to the ICML 2018 paper.

**Parameters**

- **X** (*list*) – a list of data points, each being either a list or a tree, depending on the edit distance that shall be learned.
- **y** (*array_like or list*) – an array-like or list-like structure with labels for each data point.

**Returns** self

**Return type** class bedl.BEDL

**class** edist.bedl.**EmbeddingDelta**(*embedding*)
This class serves as a storage for an embedding to make the embedding delta function pickleable.

**Parameters** **embedding** (*array_like*) – An embedding matrix.

**_Delta**
The current symbol-to-symbol distance matrix

**Type** array_like

**delta**(*x*, *y*)
Computes the distance between two embedding vectors identified by their index.

**Parameters**

- **x** (*int*) – the left-hand-side index or None.
- **y** (*int*) – the right-hand-side index or None.

**Returns** **d** – The Euclidean distance between the embedding for x and for y.

**Return type** float

**delta_with_indexing**(*x*, *y*)
Computes the distance between two symbols based on their embedding vectors.

**Parameters**

- **x** (*str*) – a symbol.
- **y** (*str*) – another symbol.

**Returns** **d** – The Euclidean distance between the embedding for x and for y.

**Return type** float

edist.bedl.**create_index**(*lst*)
Creates a map of list elements to indices.

**Parameters** **lst** (*list*) – A list.

**Returns** **idx** – A map of list elements to indices.

**Return type** dictionary

edist.bedl.**index_data**(*Xs*, *idx*)
Indexes all data in the input dataset according to the given index.

**Parameters**

- **Xs** (*list*) – A list of data, each being either a list or a tree in node list/adjacency list format.
- **idx** (*dictionary*) – A map from symbols to indices.

**Returns** **Ys** – A copy of Xs, where each symbol is replaced by a symbol index.

**Return type** list

edist.bedl.**initialize_embedding**(*size*)
Sets up a size-dimensional simplex with side length 1 and size+1 vertices (i.e. an equilateral hyper-triangle).

**Parameters** **size** (*int*) – the dimensionality of the simplex.

**Returns** **embedding** – A size x size matrix, where each row contains one vertex of the simplex. The origin is the last vertex.

**Return type** array_like

edist.bedl.**reduce_backtrace**(*P*, *x*, *y*, *size*)
Transforms the input backtrace matrix P of size len(x) + 1 x len(y) + 1 into a reduced matrix Phat of size size + 1 x size + 1 which accumulates the probabilities for same symbols being replaced, i.e. Phat[k, l] is the sum over all entries P[i, j] where x[i] = k and y[j] = l.

**Parameters**

- **P** (*array_like*) – A len(x) + 1 x len(y) + 1 backtracing matrix between sequence x and y, where P[i, j] is the probability of x[i] being replaced with y[j] in a co-optimal alignment,

P[i, len(y)] is the deletion probability for x[i] and P[len(x), j] is the insertion probability for y[j]. The last row and column are optional (as in case of DTW).

- **X** (*list*) – A list of objects, either sequences or trees.
- **Y** (*list*) – A list of objects, either sequences or trees.
- **size** (*int*) – The alphabet size.

**Returns** **Phat** – A size+1 x size+1 matrix which accumulates the probabilities for same symbols being replaced, i.e. Phat[k, l] is the sum over all entries P[i, j] where x[i] = k and y[j] = l.

**Return type** array_like

# CHAPTER 5

## Dynamic Time Warping

Implements the dynamic time warping distance of Vintsyuk (1968) and its backtracing in cython.

edist.dtw.**dtw**()

Computes the dynamic time warping distance between the input sequence x and the input sequence y, given the element-wise distance function delta.

**Parameters**

- **x** (*list*) – a sequence of objects.
- **y** (*list*) – another sequence of objects.
- **delta** (*function*) – a function that takes an element of x as first and an element of y as second input and returns the distance between them.

**Returns** **d** – the dynamic time warping distance between x and y according to delta.

**Return type** float

edist.dtw.**dtw_backtrace**()

Computes a co-optimal alignment between the two input sequences x and y, given the element-wise distance function delta. This mechanism is deterministic and will always prefer replacements over other options.

**Parameters**

- **x** (*list*) – a sequence of objects.
- **y** (*list*) – another sequence of objects.
- **delta** (*function*) – a function that takes an element of x as first and an element of y as second input and returns the distance between them.

**Returns** **alignment** – A co-optimal alignment between x and y according to dynamic time warping.

**Return type** class alignment.Alignment

edist.dtw.**dtw_backtrace_matrix**()

Computes a matrix, summarizing all co-optimal alignments between x and y in a matrix P, where entry P[i, j] specifies the fraction of co-optimal alignments in which node x[i] has been aligned with node y[j].

**Parameters**

- **x** (*list*) – a sequence of objects.
- **y** (*list*) – another sequence of objects.
- **delta** (*function*) – a function that takes an element of x as first and an element of y as second input and returns the distance between them.

**Returns**

- **P** (*array_like*) – a matrix, where entry P[i, j] specifies the fraction of co-optimal alignments in which node x[i] has been aligned with node y[j].
- **K** (*array_like*) – a matrix that contains the counts for all co-optimal alignments in which node x[i] has been aligned with node y[j].
- **k** (*int*) – the number of co-optimal alignments overall, such that P = K / k.

edist.dtw.**dtw_backtrace_stochastic**()

Computes a co-optimal alignment between the two input sequences x and y, given the element-wise distance function delta. This mechanism is stochastic and will return a random co-optimal alignment.

Note that the randomness does _not_ produce a uniform distribution over all co-optimal alignments because random choices at the start of the alignment process dominate. If you wish to characterize the overall distribution accurately, use sed_backtrace_matrix instead.

**Parameters**

- **x** (*list*) – a sequence of objects.
- **y** (*list*) – another sequence of objects.
- **delta** (*function*) – a function that takes an element of x as first and an element of y as second input and returns the distance between them.

**Returns** **alignment** – A co-optimal alignment between x and y according to dynamic time warping.

**Return type** class alignment.Alignment

edist.dtw.**dtw_euclidean**()

Computes the multivariate dynamic time warping distance between two input arrays x and y, using the Euclidean distance as element-wise distance measure.

**Parameters**

- **x** (*array_like*) – a m x K matrix of doubles.
- **y** (*array_like*) – a n x K matrix of doubles.

**Returns** **d** – the dynamic time warping distance between x and y.

**Return type** float

edist.dtw.**dtw_manhattan**()

Computes the multivariate dynamic time warping distance between two input arrays x and y, using the Manhattan distance as element-wise distance measure.

**Parameters**

- **x** (*array_like*) – a m x K matrix of doubles.
- **y** (*array_like*) – a n x K matrix of doubles.

**Returns** **d** – the dynamic time warping distance between x and y.

**Return type** float

edist.dtw.**dtw_numeric**()

Computes the dynamic time warping distance between two input arrays x and y, using the absolute value as element-wise distance measure.

**Parameters**

- **x** (*array_like*) – an array of doubles.
- **y** (*array_like*) – another array of doubles.

**Returns** **d** – the dynamic time warping distance between x and y.

**Return type** float

edist.dtw.**dtw_string**()

Computes the dynamic time warping distance between two input strings x and y, using the Kronecker distance as element-wise distance measure.

**Parameters**

- **x** (*str*) – a string.
- **y** (*str*) – another string.

**Returns** **d** – the dynamic time warping distance between x and y.

**Return type** float

CHAPTER 6

# Edits

Implements list edits, i.e. functions which take a list as input and return a changed list.

**class** edist.edits.**Deletion**(*index*)
Deletes node self._index.

**_index**
The index of the node to be deleted.

**Type** int

**apply**(*lst*)
Deletes the self._indexth entry.

**Parameters** **lst** (*list*) – a list

**Returns** **res** – a copy of the list with the applied edit.

**Return type** list

**apply_in_place**(*lst*)
Deletes the self._indexth entry.

**Parameters** **lst** (*list*) – a list

**class** edist.edits.**Edit**
An abstract parent class for all edits.

**apply**(*lst*)
Applies this edit to the given list and returns a copy of the list with the applied changes. The original list remains unchanged.

**Parameters** **lst** (*list*) – a list

**Returns** **res** – a copy of the list with the applied edit.

**Return type** list

**apply_in_place**(*lst*)
Applies this edit to the given list. Note that this changes the input argument.

**Parameters** **lst** (*list*) – a list

**class** edist.edits.**Insertion**(*index*, *label*)
Inserts a new self._label entry at position self._index .

**_index**
The index at which this edit will be applied.

**Type** int

**_label**
The new entry.

**Type** str

**apply**(*lst*)
Inserts a new self._label entry at position self._index.

**Parameters** **lst** (*list*) – a list

**Returns** **res** – a copy of the list with the applied edit.

**Return type** list

**apply_in_place**(*lst*)
Inserts a new self._label entry at position self._index.

**Parameters** **lst** (*list*) – a list

**class** edist.edits.**Replacement**(*index*, *label*)
Replaces node self._index with label self._label.

**_index**
The index of the node to be deleted.

**Type** int

**_label**
The new label for node self._index.

**Type** str

**apply**(*lst*)
Replaces entry self._index with self._label.

**Parameters** **lst** (*list*) – a list

**Returns** **res** – a copy of the list with the applied edit.

**Return type** list

**apply_in_place**(*lst*)
Replaces entry self._index with self._label.

**Parameters** **lst** (*list*) – a list

**class** edist.edits.**Script**(*lst=[]*)
A list of Edits.

**apply**(*lst*)
Applies all edits in this script.

**Parameters** **lst** (*list*) – a list

**Returns** **res** – a copy of the list with the applied edits of this script.

**Return type** list

**apply_in_place**(*lst*)
Applies all edits in this script in place.

**Parameters** **lst** (*list*) – a list

edist.edits.**alignment_to_script**(*alignment*, *x*, *y*)
Converts the given alignment into an edit script which maps the given list x to the given list y such that all entries of the alignment are translated one to one into edits.

Note that the order of operations does change because the script will first apply replacements (in input order), then deletions (in descending order), and finally insertions (in ascending order), which simplifies conversion.

**Parameters**

- **alignment** (*class alignment.Alignment*) – an Alignment object which maps between the given lists x and y.
- **x** (*list*) – a list.
- **y** (*list*) – another list.

**Returns** **script** – A Script object script such that script.apply(x) = y and where every tuple in the alignment has one corresponding edit.

**Return type** class edits.Script

CHAPTER 7

# Parallel Pairwise Edit Distance Computation

Provides general utility functions to compute pairwise edit distances in parallel.

edist.multiprocess.**pairwise_backtraces**(*Xs*, *Ys*, *dist_backtrace*, *delta=None*, *num_jobs=8*)

Computes the pairwise backtraces between the objects in Xs and the objects in Ys. Each object in Xs and Ys needs to be a valid input for the given distance function, i.e. a sequence or a tree.

Optionally, it is possible to specify a component-wise distance function delta, which will then be forwarded to the input distance function

**Parameters**

- **Xs** (*list*) – a list of sequences or trees.
- **Ys** (*list*) – another list of sequences or trees.
- **dist_backtrace** (*function*) – a function that takes an element of Xs as first and an element of Ys as second input and returns an arbitrary object.
- **delta** (*function (default = None)*) – a function that takes two elements of the input sequences or trees as inputs and returns their pairwise distance, where delta(x, None) should be the cost of deleting x and delta(None, y) should be the cost of inserting y. If this is not None, dist needs to accept an optional argument 'delta' as well. Defaults to None.
- **num_jobs** (*int (default = 8)*) – The number of jobs to be used for parallel processing. Defaults to 8.

**Returns** **B** – a len(Xs) x len(Ys) list of lists of pairwise backtraces.

**Return type** list

edist.multiprocess.**pairwise_distances**(*Xs*, *Ys*, *dist*, *delta=None*, *num_jobs=8*)

Computes the pairwise edit distances between the objects in Xs and the objects in Ys. Each object in Xs and Ys needs to be a valid input for the given distance function, i.e. a sequence or a tree.

Optionally, it is possible to specify a component-wise distance function delta, which will then be forwarded to the input distance function

**Parameters**

- **Xs** (*list*) – a list of sequences or trees.
- **Ys** (*list*) – another list of sequences or trees.
- **dist** (*function*) – a function that takes an element of Xs as first and an element of Ys as second input and returns a scalar distance value between them.
- **delta** (*function (default = None)*) – a function that takes two elements of the input sequences or trees as inputs and returns their pairwise distance, where delta(x, None) should be the cost of deleting x and delta(None, y) should be the cost of inserting y. If this is not None, dist needs to accept an optional argument 'delta' as well. Defaults to None.
- **num_jobs** (*int (default = 8)*) – The number of jobs to be used for parallel processing. Defaults to 8.

**Returns** **D** – a len(Xs) x len(Ys) matrix of pairwise edit distance values.

**Return type** array_like

edist.multiprocess.**pairwise_distances_symmetric**(*Xs*, *dist*, *delta=None*, *num_jobs=8*)

Computes the pairwise edit distances between the objects in Xs, assuming that the distance measure is symmetric. Each object in Xs needs to be a valid input for the given distance function, i.e. a sequence or a tree. Due to symmetry, this method is about double as fast compared to pairwise_distances.

Optionally, it is possible to specify a component-wise distance function delta, which will then be forwarded to the input distance function

**Parameters**

- **Xs** (*list*) – a list of sequences or trees.
- **dist** (*function*) – a function that takes two elements of Xs as inputs and returns a scalar distance value between them.
- **delta** (*function (default = None)*) – a function that takes two elements of the input sequences or trees as inputs and returns their pairwise distance, where delta(x, None) should be the cost of deleting x and delta(None, y) should be the cost of inserting y. If this is not None, dist needs to accept an optional argument 'delta' as well. Defaults to None.
- **num_jobs** (*int (default = 8)*) – The number of jobs to be used for parallel processing. Defaults to 8.

**Returns** **D** – a symmetric len(Xs) x len(Xs) matrix of pairwise edit distance values.

**Return type** array_like

CHAPTER 8

# Sequence Edit Distance

Implements the sequence edit distance of Levenshtein (1965) and its backtracing in cython.

edist.sed.**sed**()

Computes the sequence edit distance between the input sequence x and the input sequence y, given the element-wise distance function delta.

**Parameters**

- **x** (*list*) – a sequence of objects.
- **y** (*list*) – another sequence of objects.
- **delta** (*function (default = None)*) – a function that takes an element of x as first and an element of y as second input and returns the distance between them. If None, this method calls standard_sed instead.

**Returns** **d** – the sequence edit distance between x and y according to delta.

**Return type** float

edist.sed.**sed_backtrace**()

Computes a co-optimal alignment between the two input sequences x and y, given the element-wise distance function delta. This mechanism is deterministic and will always prefer replacements over other options.

**Parameters**

- **x** (*list*) – a sequence of objects.
- **y** (*list*) – another sequence of objects.
- **delta** (*function (default = None)*) – a function that takes an element of x as first and an element of y as second input and returns the distance between them. If None, this method calls standard_sed_backtrace instead.

**Returns** **alignment** – A co-optimal alignment between x and y according to the sequence edit distance.

**Return type** class alignment.Alignment

edist.sed.**sed_backtrace_matrix**()

Computes a matrix, summarizing all co-optimal alignments between x and y in a matrix P, where entry P[i, j] specifies the fraction of co-optimal alignments in which node x[i] has been aligned with node y[j].

**Parameters**

- **x** (*list*) – a sequence of objects.
- **y** (*list*) – another sequence of objects.
- **delta** (*function (default = None)*) – a function that takes an element of x as first and an element of y as second input and returns the distance between them. If None, this method calls standard_sed_backtrace_matrix instead.

**Returns**

- **P** (*array_like*) – a matrix, where entry P[i, j] specifies the fraction of co-optimal alignments in which node x[i] has been aligned with node y[j]. P[i, n] contains the fraction of deletions of node x[i] and P[m, j] the fraction of insertions of node y[j].
- **K** (*array_like*) – a matrix that contains the counts for all co-optimal alignments in which node x[i] has been aligned with node y[j].
- **k** (*int*) – the number of co-optimal alignments overall, such that P = K / k.

edist.sed.**sed_backtrace_stochastic**()

Computes a co-optimal alignment between the two input sequences x and y, given the element-wise distance function delta. This mechanism is stochastic and will return a random co-optimal alignment.

Note that the randomness does _not_ produce a uniform distribution over all co-optimal alignments because random choices at the start of the alignment process dominate. If you wish to characterize the overall distribution accurately, use sed_backtrace_matrix instead.

**Parameters**

- **x** (*list*) – a sequence of objects.
- **y** (*list*) – another sequence of objects.
- **delta** (*function (default = None)*) – a function that takes an element of x as first and an element of y as second input and returns the distance between them. If None, this method calls standard_sed_backtrace_stochastic instead.

**Returns** **alignment** – A co-optimal alignment between x and y according to the sequence edit distance.

**Return type** class alignment.Alignment

edist.sed.**sed_string**()

Computes the standard sequence edit distance/Levenshtein distance between two input strings x and y, using the Kronecker distance as element-wise distance measure.

**Parameters**

- **x** (*str*) – a string.
- **y** (*str*) – another string.

**Returns** **d** – the standard sequence edit distance between x and y.

**Return type** int

edist.sed.**standard_sed**()

Computes the standard sequence edit distance/Levenshtein distance between the input sequence x and the input sequence y.

**Parameters**

- **x** (*list*) – a sequence of objects.
- **y** (*list*) – another sequence of objects.

**Returns** **d** – the standard sequence edit distance between x and y.

**Return type** int

edist.sed.**standard_sed_backtrace**()

Computes a co-optimal alignment between the two input sequences x and y. This mechanism is deterministic and will always prefer replacements over other options.

**Parameters**

- **x** (*list*) – a sequence of objects.
- **y** (*list*) – another sequence of objects.

**Returns** **alignment** – A co-optimal alignment between x and y according to the sequence edit distance.

**Return type** class alignment.Alignment

edist.sed.**standard_sed_backtrace_matrix**()

Computes a matrix, summarizing all co-optimal alignments between x and y in a matrix P, where entry P[i, j] specifies the fraction of co-optimal alignments in which node x[i] has been aligned with node y[j].

**Parameters**

- **x** (*list*) – a sequence of objects.
- **y** (*list*) – another sequence of objects.

**Returns**

- **P** (*array_like*) – a matrix, where entry P[i, j] specifies the fraction of co-optimal alignments in which node x[i] has been aligned with node y[j]. P[i, n] contains the fraction of deletions of node x[i] and P[m, j] the fraction of insertions of node y[j].
- **K** (*array_like*) – a matrix that contains the counts for all co-optimal alignments in which node x[i] has been aligned with node y[j].
- **k** (*int*) – the number of co-optimal alignments overall, such that P = K / k.

edist.sed.**standard_sed_backtrace_stochastic**()

Computes a co-optimal alignment between the two input sequences x and y, given the element-wise distance function delta. This mechanism is stochastic and will return a random co-optimal alignment.

Note that the randomness does _not_ produce a uniform distribution over all co-optimal alignments because random choices at the start of the alignment process dominate. If you wish to characterize the overall distribution accurately, use sed_backtrace_matrix instead.

**Parameters**

- **x** (*list*) – a sequence of objects.
- **y** (*list*) – another sequence of objects.

**Returns** **alignment** – A co-optimal alignment between x and y according to the sequence edit distance.

**Return type** class alignment.Alignment

CHAPTER 9

# Set Edit Distance

Provides a set edit distance to compare two sets (each represented as lists for computational ease).

edist.seted.**seted**()

Computes the set edit distance between the input set x and the input set y, given the element-wise distance function delta.

In more detail, this function finds an alignment M = {(i, j)} subset of {1, . . . , len(x)} x {1, . . . , len(y)}, such that the following loss is minimized:

$$\sum_{(i,j)\in M} \delta(x_i, y_j) + \sum_{i\notin M} \delta(x_i, -) + \sum_{j\notin M} \delta(-, y_j)$$

where i is said to be not in M if there exists no tuple (i, j) in M for any j, and j is said to be not in M if there exists no tuple (i, j) in M for any i.

This problem can be solved via the Hungarian or Munkres algorithm in O([len(x) + len(y)]^3). So be advised that this method can become very slow for large input sets.

**Parameters**

- **x** (*list-like*) – a set of objects.
- **y** (*list-like*) – another set of objects.
- **delta** (*function (default = None)*) – a function that takes an element of x as first and an element of y as second input and returns the distance between them. If not given, standard_seted is called instead.

**Returns** **d** – the set edit distance between x and y according to delta.

**Return type** float

edist.seted.**seted_backtrace**()

Computes a co-optimal alignment between the two input sequences x and y, given the element-wise distance function delta. This mechanism is deterministic and depends on the implementation of scipy.optimize.linear_sum_assignment for the choice of co-optimal alignment.

**Parameters**

- **x** (*list-like*) – a set of objects.
- **y** (*list-like*) – another set of objects.
- **delta** (*function (default = None)*) – a function that takes an element of x as first and an element of y as second input and returns the distance between them. If None, this method calls standard_seted_backtrace instead.

**Returns** **alignment** – A co-optimal alignment between x and y according to the set edit distance.

**Return type** class alignment.Alignment

edist.seted.**standard_seted**()

Computes the standard set edit distance between the input set x and the input set y according to the Kronecker distance, i.e. the replacement cost is zero if two elements are equal and one otherwise.

In more detail, this function finds an alignment M = {(i, j)} subset of {1, . . . , len(x)} x {1, . . . , len(y)}, such that the following loss is minimized:

$$\sum_{(i,j)\in M} \delta(x_i, y_j) + |i \notin M| + |j \notin M|$$

where i is said to be not in M if there exists no tuple (i, j) in M for any j, and j is said to be not in M if there exists no tuple (i, j) in M for any i.

This problem can be solved via the Hungarian or Munkres algorithm in O([len(x) + len(y)]^3). So be advised that this method can become very slow for large input sets.

**Parameters**

- **x** (*list-like*) – a set of objects.
- **y** (*list-like*) – another set of objects.

**Returns** **d** – the set edit distance between x and y according to delta.

**Return type** float

edist.seted.**standard_seted_backtrace**()

Computes a co-optimal alignment between the two input sequences x and y, given the element-wise distance function delta. This mechanism is deterministic and depends on the implementation of scipy.optimize.linear_sum_assignment for the choice of co-optimal alignment.

**Parameters**

- **x** (*list-like*) – a set of objects.
- **y** (*list-like*) – another set of objects.

**Returns** **alignment** – A co-optimal alignment between x and y according to the standard set edit distance.

**Return type** class alignment.Alignment

# CHAPTER 10

## Tree Edit Distance

Implements the tree edit distance of Zhang and Shasha (1989) and its backtracing in cython.

edist.ted.**extract_from_tuple_input**()

Assumes that both x and y are tuples and unpacks those tuples.

**x: tuple** a tuple

**y: tuple** another tuple

**Returns**

- **x_nodes** (*list*) – x[0]
- **x_adj** (*list*) – x[1]
- **y_nodes** (*list*) – y[0]
- **y_adj** (*list*) – y[1]

**Raises** ValueError – if the input is not tuple-shaped.

edist.ted.**keyroots**()

Computes the keyroots of a tree based on its outermost right leaf array. The keyroot for a node i is defined as the lowest k, such that orl[i] = orl[k].

**Parameters** **orl** (*array_like*) – An outermost right leaf array as computed by the outermost_right_leaves function above.

**Returns** **keyroots** – An array of keyroots in descending order.

**Return type** int array

edist.ted.**outermost_right_leaves**()

Computes the outermost right leaves of a tree based on its adjacency list. The outermost right leaf of a tree is defined as recursively accessing the right-most child of a node until we hit a leaf.

Note that we assume a proper depth-first-search order of adj, i.e. for every node i, the following indices are all part of the subtree rooted at i until we hit the index of i's right sibling or the end of the tree.

**Parameters** **adj** (*list*) – An adjacency list representation of the tree, i.e. an array such that for every i, adj[i] is the list of child indices for node i.

**Returns** **orl** – An array containing the outermost right leaf index for every node in the tree.

**Return type** int array

edist.ted.**standard_ted**()

Computes the standard tree edit distance between the trees x and y, each described by a list of nodes and an adjacency list adj, where adj[i] is a list of indices pointing to children of node i.

The 'standard' refers to the fact that we use the kronecker distance as delta, i.e. this call computes the same as

ted(x_nodes, x_adj, y_nodes, y_adj, kronecker_distance) where

kronecker_distance(x, y) = 1 if x != y and 0 if x = y.

However, this implementation here is notably faster because we can apply integer arithmetic.

Note that we assume a proper depth-first-search order of adj, i.e. for every node i, the following indices are all part of the subtree rooted at i until we hit the index of i's right sibling or the end of the tree.

**Parameters**

- **x_nodes** (*list or tuple*) – a list of nodes for tree x OR a tuple of the form (x_nodes, x_adj).
- **x_adj** (*list or tuple*) – an adjacency list for tree x OR a tuple of the form (y_nodes, y_adj).
- **y_nodes** (*list (default = x_adj[0])*) – a list of nodes for tree y.
- **y_adj** (*list (default = x_adj[1])*) – an adjacency list for tree y.

**Returns** **d** – the standard tree edit distance between x and y according.

**Return type** int

edist.ted.**standard_ted_backtrace**()

Computes the standard tree edit distance between the trees x and y, each described by a list of nodes and an adjacency list adj, where adj[i] is a list of indices pointing to children of node i. This function returns an alignment representation of the distance.

The 'standard' refers to the fact that we use the kronecker distance as delta, i.e. this call computes the same as

ted(x_nodes, x_adj, y_nodes, y_adj, kronecker_distance) where

kronecker_distance(x, y) = 1 if x != y and 0 if x = y.

However, this implementation here is notably faster because we can apply integer arithmetic.

Note that we assume a proper depth-first-search order of adj, i.e. for every node i, the following indices are all part of the subtree rooted at i until we hit the index of i's right sibling or the end of the tree.

**Parameters**

- **x_nodes** (*list or tuple*) – a list of nodes for tree x OR a tuple of the form (x_nodes, x_adj).
- **x_adj** (*list or tuple*) – an adjacency list for tree x OR a tuple of the form (y_nodes, y_adj).
- **y_nodes** (*list (default = x_adj[0])*) – a list of nodes for tree y.
- **y_adj** (*list (default = x_adj[1])*) – an adjacency list for tree y.

**Returns** **alignment** – A co-optimal alignment between x and y according to the tree edit distance.

**Return type** class alignment.Alignment

edist.ted.**standard_ted_backtrace_matrix**()

Computes a matrix P where entry P[i, j] represents how often node i in tree x was aligned with node j in tree y in co-optimal alignments according to the standard tree edit distance.

Note that we assume a proper depth-first-search order of adj, i.e. for every node i, the following indices are all part of the subtree rooted at i until we hit the index of i's right sibling or the end of the tree.

**Parameters**

- **x_nodes** (*list or tuple*) – a list of nodes for tree x OR a tuple of the form (x_nodes, x_adj).
- **x_adj** (*list or tuple*) – an adjacency list for tree x OR a tuple of the form (y_nodes, y_adj).
- **y_nodes** (*list (default = x_adj[0])*) – a list of nodes for tree y.
- **y_adj** (*list (default = x_adj[1])*) – an adjacency list for tree y.

**Returns**

- **P** (*array_like*) – a matrix, where entry P[i, j] specifies the fraction of co-optimal alignments in which node x[i] has been aligned with node y[j]. P[i, n] contains the fraction of deletions of node x[i] and P[m, j] the fraction of insertions of node y[j].
- **K** (*array_like*) – a matrix that contains the counts for all co-optimal alignments in which node x[i] has been aligned with node y[j].
- **k** (*int*) – the number of co-optimal alignments overall, such that P = K / k.

edist.ted.**ted**()

Computes the tree edit distance between the trees x and y, each described by a list of nodes and an adjacency list adj, where adj[i] is a list of indices pointing to children of node i.

Note that we assume a proper depth-first-search order of adj, i.e. for every node i, the following indices are all part of the subtree rooted at i until we hit the index of i's right sibling or the end of the tree.

**Parameters**

- **x_nodes** (*list or tuple*) – a list of nodes for tree x OR a tuple of the form (x_nodes, x_adj).
- **x_adj** (*list or tuple*) – an adjacency list for tree x OR a tuple of the form (y_nodes, y_adj).
- **y_nodes** (*list (default = x_adj[0])*) – a list of nodes for tree y.
- **y_adj** (*list (default = x_adj[1])*) – an adjacency list for tree y.
- **delta** (*function (default = None)*) – a function that takes two nodes as inputs and returns their pairwise distance, where delta(x, None) should be the cost of deleting x and delta(None, y) should be the cost of inserting y. If undefined, this method calls standard_ted instead.

**Returns** **d** – the tree edit distance between x and y according to delta.

**Return type** float

edist.ted.**ted_backtrace**()

Computes the tree edit distance between the trees x and y, each described by a list of nodes and an adjacency list adj, where adj[i] is a list of indices pointing to children of node i. This function returns an alignment representation of the distance.

Note that we assume a proper depth-first-search order of adj, i.e. for every node i, the following indices are all part of the subtree rooted at i until we hit the index of i's right sibling or the end of the tree.

**Parameters**

- **x_nodes** (*list or tuple*) – a list of nodes for tree x OR a tuple of the form (x_nodes, x_adj).
- **x_adj** (*list or tuple*) – an adjacency list for tree x OR a tuple of the form (y_nodes, y_adj).
- **y_nodes** (*list (default = x_adj[0])*) – a list of nodes for tree y.
- **y_adj** (*list (default = x_adj[1])*) – an adjacency list for tree y.
- **delta** (*function (default = None)*) – a function that takes two nodes as inputs and returns their pairwise distance, where delta(x, None) should be the cost of deleting x and delta(None, y) should be the cost of inserting y. If undefined, this method calls standard_ted instead.

**Returns** **alignment** – A co-optimal alignment between x and y according to the tree edit distance.

**Return type** class alignment.Alignment

edist.ted.**ted_backtrace_matrix**()

Computes a matrix P where entry P[i, j] represents how often node i in tree x was aligned with node j in tree y in co-optimal alignments according to the tree edit distance.

Note that we assume a proper depth-first-search order of adj, i.e. for every node i, the following indices are all part of the subtree rooted at i until we hit the index of i's right sibling or the end of the tree.

**Parameters**

- **x_nodes** (*list or tuple*) – a list of nodes for tree x OR a tuple of the form (x_nodes, x_adj).
- **x_adj** (*list or tuple*) – an adjacency list for tree x OR a tuple of the form (y_nodes, y_adj).
- **y_nodes** (*list (default = x_adj[0])*) – a list of nodes for tree y.
- **y_adj** (*list (default = x_adj[1])*) – an adjacency list for tree y.
- **delta** (*function (default = None)*) – a function that takes two nodes as inputs and returns their pairwise distance, where delta(x, None) should be the cost of deleting x and delta(None, y) should be the cost of inserting y. If undefined, this method calls standard_ted instead.

**Returns**

- **P** (*array_like*) – a matrix, where entry P[i, j] specifies the fraction of co-optimal alignments in which node x[i] has been aligned with node y[j]. P[i, n] contains the fraction of deletions of node x[i] and P[m, j] the fraction of insertions of node y[j].
- **K** (*array_like*) – a matrix that contains the counts for all co-optimal alignments in which node x[i] has been aligned with node y[j].
- **k** (*int*) – the number of co-optimal alignments overall, such that P = K / k.

CHAPTER 11

# Unordered Tree Edit Distance

Implements the constrained unordered tree edit distance of Zhang (1996) and its backtracing in cython.

edist.uted.**adjmat_**()
Converts an adjacency list into an int array

edist.uted.**munkres**()
This calls the Munkres algorithm to find a minimal matching for the given cost matrix C. Note that this function is more for debugging purposes. If you want to call this algorithm from Python, you are better served by calling scipy.optimize.linear_sum_assignment.

**Parameters** **C** (*ndarray*) – An m x m cost matrix.

**Returns** **pi** – An m-element array where pi[i] is the index to which i is assigned.

**Return type** ndarray

edist.uted.**uted**()

**Computes the constrained, unordered tree edit distance between the** trees x and y, each described by a list of nodes and an adjacency list

adj, where adj[i] is a list of indices pointing to children of node i.

Note that we assume a proper depth-first-search order of adj, i.e. for every node i, the following indices are all part of the subtree rooted at i until we hit the index of i's right sibling or the end of the tree.

**Parameters**

- **x_nodes** (*list or tuple*) – a list of nodes for tree x OR a tuple of the form (x_nodes, x_adj).
- **x_adj** (*list or tuple*) – an adjacency list for tree x OR a tuple of the form (y_nodes, y_adj).
- **y_nodes** (*list (default = x_adj[0])*) – a list of nodes for tree y.
- **y_adj** (*list (default = x_adj[1])*) – an adjacency list for tree y.
- **delta** (*function (default = None)*) – a function that takes two nodes as inputs and returns their pairwise distance, where delta(x, None) should be the cost of deleting x

and delta(None, y) should be the cost of inserting y. If undefined, this method uses unit costs.

**Returns** **d** – the tree edit distance between x and y according to delta.

**Return type** float

edist.uted.**uted_backtrace**()

Computes the unordered tree edit distance between the trees x and y, each described by a list of nodes and an adjacency list adj, where adj[i] is a list of indices pointing to children of node i. This function returns an alignment representation of the distance.

Note that we assume a proper depth-first-search order of adj, i.e. for every node i, the following indices are all part of the subtree rooted at i until we hit the index of i's right sibling or the end of the tree.

**Parameters**

- **x_nodes** (*list or tuple*) – a list of nodes for tree x OR a tuple of the form (x_nodes, x_adj).
- **x_adj** (*list or tuple*) – an adjacency list for tree x OR a tuple of the form (y_nodes, y_adj).
- **y_nodes** (*list (default = x_adj[0])*) – a list of nodes for tree y.
- **y_adj** (*list (default = x_adj[1])*) – an adjacency list for tree y.
- **delta** (*function (default = None)*) – a function that takes two nodes as inputs and returns their pairwise distance, where delta(x, None) should be the cost of deleting x and delta(None, y) should be the cost of inserting y. If undefined, this method calls standard_ted instead.

**Returns** **alignment** – A co-optimal alignment between x and y according to the unordered tree edit distance.

**Return type** class alignment.Alignment

CHAPTER 12

# Tree Edits

Implements tree edits, i.e. functions which take a tree as input and return a changed tree.

**class** edist.tree_edits.**Deletion**(*index*)

**Deletes node self._index and raises all its children to the parent** node. Note that deleting the root node of the tree results in a forest instead of a tree.

**_index**
The index of the tree node to which this edit is applied.

**Type** int

**apply**(*nodes_orig*, *adj_orig*)
Deletes node self._index and raises all its children to the parent node. Note that deleting the root node of the tree results in a forest instead of a tree.

**Parameters**

- **nodes** (*list*) – a node list.
- **adj** (*list*) – an adjacency list.

**Returns**

- **nodes** (*list*) – a copy of nodes with the applied edit.
- **adj** (*list*) – a copy of adj with the applied edit.

**apply_in_place**(*nodes*, *adj*)
Deletes node self._index and raises all its children to the parent node. Note that deleting the root node of the tree results in a forest instead of a tree.

**Parameters**

- **nodes** (*list*) – a node list.
- **adj** (*list*) – an adjacency list.

**class** edist.tree_edits.**Edit**
An abstract parent class for all edits.

**apply**(*nodes*, *adj*)
Applies this edit to the given tree and returns a copy of the tree with the applied changes. The original tree remains unchanged.

**Parameters**

- **nodes** (*list*) – a node list.
- **adj** (*list*) – an adjacency list.

**Returns**

- **nodes** (*list*) – a copy of nodes with the applied edit.
- **adj** (*list*) – a copy of adj with the applied edit.

**apply_in_place**(*nodes*, *adj*)
Applies this edit to the given tree. Note that this changes the input arguments.

**Parameters**

- **nodes** (*list*) – a node list.
- **adj** (*list*) – an adjacency list.

**class** edist.tree_edits.**Insertion**(*parent_index*, *child_index*, *label*, *num_children=0*)

**Inserts a new node with label self._label as self._child_index'th** child of node self._parent_index and uses the next self._num_children right siblings of itself as its children.

**_parent_index**
The index of the tree node to which we add a new child.

**Type** int

**_label**
The label for the new child node.

**Type** str

**_child_index**
The new node will be the _child_indexth child of node _parent_index.

**Type** int

**_num_children**
The number of right siblings that will be used as new grandchildren.

**Type** int

**apply**(*nodes_orig*, *adj_orig*)
Inserts a new node with label self._label as self._child_index'th child of node self._parent_index and uses the next self._num_children right siblings of itself as its children.

Note that inserting a new child at the root leads to a forest instead of a tree structure.

**Parameters**

- **nodes** (*list*) – a node list.
- **adj** (*list*) – an adjacency list.

**Returns**

- **nodes** (*list*) – a copy of nodes with the applied edit.
- **adj** (*list*) – a copy of adj with the applied edit.

**apply_in_place**(*nodes*, *adj*)
Inserts a new node with label self._label as self._child_index'th child of node self._parent_index and uses the next self._num_children right siblings of itself as its children.

Note that inserting a new child at the root leads to a forest instead of a tree structure.

**Parameters**

- **nodes** (*list*) – a node list.
- **adj** (*list*) – an adjacency list.

**class** edist.tree_edits.**Replacement**(*index*, *label*)
Replaces the label of node self._index with self._label.

**_index**
The index of the tree node to which this edit is applied.

**Type** int

**_label**
The new label for the self._indexth node.

**Type** str

**apply**(*nodes_orig*, *adj*)
Replaces the label of node self._index with self._label.

**Parameters**

- **nodes** (*list*) – a node list.
- **adj** (*list*) – an adjacency list.

**Returns**

- **nodes** (*list*) – a copy of nodes with the applied edit.
- **adj** (*list*) – a copy of adj with the applied edit.

**apply_in_place**(*nodes*, *adj*)
Replaces the label of node self._index with self._label.

**Parameters**

- **nodes** (*list*) – a node list.
- **adj** (*list*) – an adjacency list.

**class** edist.tree_edits.**Script**(*lst=[]*)
A list of Edits.

**apply**(*nodes_orig*, *adj_orig*)
Applies all edits in this script.

**Parameters**

- **nodes** (*list*) – a node list.
- **adj** (*list*) – an adjacency list.

**Returns**

- **nodes** (*list*) – a copy of nodes with the applied edits.
- **adj** (*list*) – a copy of adj with the applied edits.

**apply_in_place**(*nodes*, *adj*)
Applies all edits in this script.

**Parameters**

- **nodes** (*list*) – a node list.
- **adj** (*list*) – an adjacency list.

edist.tree_edits.**alignment_to_script**(*alignment*, *x_nodes*, *x_adj*, *y_nodes*, *y_adj*)
Converts the given alignment into an edit script which maps the given tree x to the given tree y such that all tuples in the alignment correspond to exactly one edit in the script.

Note that the order of operations does change because the script will first apply replacements (in input order), then deletions (in descending order), and finally insertions (in ascending order), which simplifies conversion.

The precise algorithm is described in the paper: https://arxiv.org/abs/1805.06869

**Parameters**

- **alignment** (*class alignment.Alignment*) – an Alignment object which maps between the given trees x and y.
- **x_nodes** (*list*) – the node list of tree x.
- **x_adj** (*list*) – the adjacency list of tree x.
- **y_nodes** (*list*) – the node list of tree y.
- **y_adj** (*list*) – the adjacency list of tree y.

**Returns** **script** – A Script object script such that script.apply(x_nodes, x_adj) = (y_nodes, y_adj) where every Tuple in the alignment has one corresponding edit.

**Return type** class tree_edits.Script

edist.tree_edits.**get_roots**(*adj*)
Returns all roots of a forest, described by an adjacency list.

**Parameters** **adj** (*list*) – an adjacency list

**Returns** **roots** – a list of roots, ascendingly sorted.

**Return type** list

edist.tree_edits.**num_descendants**(*adj*, *filter_set*)
Counts the number of descendants of each node which are _not_ members of the given filter set.

**Parameters**

- **adj** (*list*) – an adjacency list
- **filter_set** (*set_like*) – a set excluding some node indices

**Returns** **out** – A list which contains, for each node, the number of descendants not from the filter set.

**Return type** list

# CHAPTER 13

## Tree Utilities

Provides general utility functions to process trees.

edist.tree_utils.**check_dfs_structure**(*adj*, *i=0*)

Verifies that a given adjacency list is in depth-first-search order, in the sense that the descendants of a node i are all indices i+1, i+2, ... orl[i], where orl[i] is the outermost right leaf of i.

We verify this by performing a depth-first search over the tree and checking whether the indices are equivalent

**Parameters** **adj** (*list*) – a directed graph in adjacency list format.

**Returns** **size** – the size of the input tree, i.e. the last DFS index.

**Return type** int

**Raises** ValueError – if the given adjacency list is not in DFS format.

edist.tree_utils.**check_tree_structure**(*adj*)

Verifies that a given adjacency list describes a tree.

In particular, we build a parent representation of the tree and throw an exception if that fails. This is valid because trees are the subclass of directed graphs with a unique parent.

**Parameters** **adj** (*list*) – a directed graph in adjacency list format.

**Raises** ValueError – if the given adjacency list does not form a tree.

edist.tree_utils.**dataset_from_json**(*path*)

Reads trees in node list/adjacency list format from all JSON files in the given directory.

**Parameters** **path** (*str*) – a path to a directory which contains JSON files.

**Returns**

- **X** (*list*) – A list of tuples in node list/adjacency list format.
- **filenames** (*list*) – A list of filenames from which we read the trees.

**Raises** Exception – if the file access does not work, if some JSON data is malformed, if a parsed data is not a tree, or if a parsed tree is not in depth first search order.

edist.tree_utils.**from_json**(*filename*)
Loads a tree in node list/adjacency list format from a JSON file.

**Parameters** **filename** (*str*) – A JSON filename containing tree data as written by the to_json method.

**Returns**

- **nodes** (*list*) – The node list of the tree.
- **adj** (*list*) – The adjacency list of the tree.

**Raises** Exception – if the given file is not accessible, if the JSON data is malformed, if the parsed data is not a tree, or if the parsed tree is not in depth first search order.

edist.tree_utils.**parents**(*adj*)
Returns the parent representation of the tree with the given adjacency list.

**Parameters** **adj** (*list*) – The adjacency list of the tree.

**Returns** **par** – a numpy integer array with len(adj) elements, where the ith element contains the index of the parent of the ith node. Nodes without children contain the entry -1.

**Return type** int array

edist.tree_utils.**root**(*adj*)
Returns the root of a tree and raises an error if the input adjacency matrix does not correspond to a tree.

**Parameters** **adj** (*list*) – a directed graph in adjacency list format.

**Returns** **root** – the index of the root of the tree.

**Return type** int

**Raises** ValueError – if the given adjacency list does not form a tree.

edist.tree_utils.**subtree**(*nodes*, *adj*, *i*)
Returns the subtree rooted at node i in node list/adjacency list format.

**Parameters**

- **nodes** (*list*) – The node list of the tree.
- **adj** (*list*) – The adjacency list of the tree.
- **i** (*int*) – The index of the desired subtree.

**Returns**

- **nodes_sub** (*list*) – The node list of the subtree rooted at i.
- **adj_sub** (*list*) – The adajency list of the subtree rooted at i.

edist.tree_utils.**to_dfs_structure**(*nodes*, *adj*)
Re-orders a tree to conform to a depth-first search structure.

Note that this method performs a copy and leaves the original tree untouched.

**Parameters**

- **nodes** (*list*) – A node list.
- **adj** (*list*) – An adjacency list.

**Raises** ValueError – if the input is not a tree.

edist.tree_utils.**to_json**(*filename*, *nodes*, *adj*)
Writes a tree in node list/adjacency list format to a JSON file.

**Parameters**

- **filename** (*str*) – The filename for the resulting JSON file.
- **nodes** (*list*) – The node list of the tree.
- **adj** (*list*) – The adjacency list of the tree.

**Raises** Exception – if the file is not accessible or the JSON writeout fails.

edist.tree_utils.**tree_to_string**(*nodes*, *adj*, *indent=False*, *with_indices=False*)

Prints a tree in node list/adjacency list format as string.

**Parameters**

- **nodes** (*list*) – The node list of the tree.
- **adj** (*list*) – The adjacency list of the tree.
- **indent** (*bool (default = False)*) – A boolean flag; if True, each node is printed on a new line.
- **with_indices** (*bool (default = False)*) – A boolean flag; if True, each node is printed with its index.

**Raises** ValueError – if the adjacency list does not correspond to a tree.

CHAPTER 14

# Indices and tables

- genindex
- modindex
- search

# Python Module Index

## e

# Index

## Symbols



## A

## B



## C



## D



## E



## F



## G



## I



## K



## M



## N



## O



## P

## R



## S



## T



## U



## V