Package 'simona'

November 12, 2026

Type Package

Title Semantic Similarity on Bio-Ontologies

Version 1.10.0

Date 2026-01-30

Depends R (>= 4.1.0)

Imports methods, Rcpp, matrixStats, GetoptLong, grid, GlobalOptions, igraph, Polychrome, S4Vectors, xml2 (>= 1.3.3), circlize, ComplexHeatmap, grDevices, stats, utils, shiny, fastmatch

Suggests knitr, testthat, BiocManager, GO.db, org.Hs.eg.db, proxyC, AnnotationDbi, Matrix, DiagrammeR, ragg, png, InteractiveComplexHeatmap, UniProtKeywords, simplifyEnrichment, AnnotationHub, jsonlite

LinkingTo Rcpp

VignetteBuilder knitr

Description This package implements infrastructures for ontology analysis by offering efficient data structures, fast ontology traversal methods, and elegant visualizations. It provides a robust toolbox supporting over 70 methods for semantic similarity analysis.

biocViews Software, Annotation, GO, BiomedicalInformatics

URL https://github.com/jokergoo/simona

BugReports https://github.com/jokergoo/simona/issues

SystemRequirements Perl, Java

License MIT + file LICENSE

RoxygenNote 7.3.2

Encoding UTF-8

Roxygen list(markdown = TRUE)

git_url https://git.bioconductor.org/packages/simona

git_branch RELEASE_3_23

git_last_commit 3c729fd

git_last_commit_date 2026-04-28

Repository Bioconductor 3.23

Date/Publication 2026-11-11

Author Zuguang Gu [aut, cre] (ORCID: <https://orcid.org/0000-0002-7395-8709>)

Maintainer Zuguang Gu <guzuguang@suat-sz.edu.cn>

Contents

add_annotation . . . 3
all_term_IC_methods . . . 3
create_ontology_DAG . . . 4
create_ontology_DAG_from_GO_db . . . 6
create_ontology_DAG_from_igraph . . . 7
dag_all_terms . . . 8
dag_as_igraph . . . 9
dag_circular_viz . . . 9
dag_depth . . . 12
dag_distinct_ancestors . . . 13
dag_enrich_on_items . . . 14
dag_enrich_on_offsprings . . . 15
dag_enrich_on_offsprings_by_permutation . . . 17
dag_filter . . . 18
dag_has_terms . . . 19
dag_longest_dist_to_offspring . . . 20
dag_parents . . . 21
dag_random_tree . . . 22
dag_reorder . . . 23
dag_root . . . 24
dag_shiny . . . 25
dag_treelize . . . 26
group_sim . . . 27
import_obo . . . 33
mcols,ontology_DAG-method . . . 36
method_param . . . 37
MICA_term . . . 37
n_annotations . . . 40
n_offspring . . . 41
ontology_DAG-class . . . 42
ontology_kw . . . 43
partition_by_level . . . 45
print.print_source . . . 46
random_terms . . . 47
shortest_distances_via_NCA . . . 48
show,ontology_DAG-method . . . 49
simona_opt . . . 49
term_annotations . . . 51
term_IC . . . 52
term_sim . . . 56

[,ontology_DAG,ANY,ANY,missing-method 66

Index **68**

`add_annotation` *Add annotations to the DAG object*

Description

Add annotations to the DAG object

Usage

```
add_annotation(dag, annotation)
```

Arguments

`dag`	An `ontology_DAG` object.
`annotation`	A list of character vectors which contain items annotated to the terms. Names of the list should be the term names. In the DAG, items annotated to a term will also be annotated to its parents. Such merging is applied automatically in the package.

Value

An `ontology_DAG` object.

`all_term_IC_methods` *Supported methods*

Description

Supported methods

Usage

```
all_term_IC_methods(require_anno = NULL)

all_term_sim_methods(require_anno = NULL)

all_group_sim_methods(require_anno = NULL)
```

Arguments

`require_anno`	If it is set to `TRUE`, methods that require external annotations are only returned. If it is set to `FALSE`, methods that do not require annotations are returned. A value of `NULL` returns both.

Details

- `all_term_IC_methods()`: A vector of all supported IC methods.
- `all_term_sim_methods()`: A vector of all supported term similarity methods.
- `all_group_sim_methods()`: A vector of all supported group similarity methods.

Value

A character vector of all supported methods.

Examples

```
all_term_IC_methods()
all_term_sim_methods()
all_group_sim_methods()
```

`create_ontology_DAG` *Create the ontology_DAG object*

Description

Create the ontology_DAG object

Usage

```
create_ontology_DAG(
  parents,
  children,
  relations = NULL,
  relations_DAG = NULL,
  source = "Ontology",
  annotation = NULL,
  remove_cyclic_paths = FALSE,
  remove_rings = FALSE,
  alternative_terms = list(),
  verbose = simona_opt$verbose
)
```

Arguments

`parents`	A character vector of parent terms. You can also construct the `ontology_DAG` object by a list of parent-child links. See **Examples**.
`children`	A character vector of child terms.
`relations`	A character vector of parent-child relations, e.g. "is_a", "part_of", or self-defined semantic relations. If it is set, it should have the same length as `parents` and `children`.

`relations_DAG` If the relation types have hierarchical relations, it can also be constructed by `create_ontology_DAG()` first. See **Examples**. When the DAG for relation types is provided, the ancestor/offspring relationship of relation types will be taken into consideration automatically.

`source` Source of the ontology. It is only used as a label of the object.

`annotation` A list of character vectors which contain items annotated to the terms. Names of the list should be the term names. In the DAG, items annotated to a term will also be annotated to its parents. Such merging is applied automatically in the package.

`remove_cyclic_paths`
Whether to remove cyclic paths If a cyclic path is represented as `[a, b, ..., z, a]`, the last link (i.e. `z->a`) is simply removed. If the value is set to `FALSE` and if there are cyclic paths, there will be an error that lists all cyclic paths.

`remove_rings` There might be rings that are isolated to the main DAG where there are no roots on the rings, thus they cannot be attached to the main DAG. If the value of `remove_rings` is set to `TRUE`, such rings are removed.

`alternative_terms`
A named list or vector that contains mappings from alternative term IDs to terms used in the DAG. In an ontology, there might be old terms IDs marked as "replaced_by", "consider" or "alt_id" in ".obo" file. You can provide mappings from old term iDs to current term IDs with this argument. If it is a one-to-one mapping, the mapping can be a named vector where alternative term IDs are names and DAG term IDs are values. It it is a one-to-many mapping, the variable should be a named list where each member vector will first be matched to the DAG terms. If the mapping is still one-to-many, the first one is selected.

`verbose` Whether to print messages.

Value

An `ontology_DAG` object.

Examples

```
parents  = c("a", "a", "b", "b", "c", "d")
children = c("b", "c", "c", "d", "e", "f")
dag = create_ontology_DAG(parents, children)

# with annotations
annotation = list(
    "a" = c("t1", "t2", "t3"),
    "b" = c("t3", "t4"),
    "c" = "t5",
    "d" = "t7",
    "e" = c("t4", "t5", "t6", "t7"),
    "f" = "t8"
)
dag = create_ontology_DAG(parents, children, annotation = annotation)

# with relations
```

```
dag = create_ontology_DAG(parents, children,
    relations = c("is_a", "part_of", "is_a", "part_of", "is_a", "part_of"))

# with relations_DAG
relations_DAG = create_ontology_DAG(c("r2", "r2"), c("r3", "r4"))
dag = create_ontology_DAG(parents, children,
    relations = c("r1", "r2", "r1", "r3", "r1", "r4"),
    relations_DAG = relations_DAG)

# with a list of parent-child relations
dag = create_ontology_DAG(c("a-b", "a-c", "b-c", "b-d", "c-e", "e-f"))
```

`create_ontology_DAG_from_GO_db` *Create the ontology_DAG object from the GO.db package*

Description

Create the ontology_DAG object from the GO.db package

Mappings between alternative GO terms to official GO terms

Usage

```
create_ontology_DAG_from_GO_db(
  namespace = "BP",
  relations = "part of",
  org_db = NULL,
  evidence_code = NULL,
  retrieve_alternative = FALSE,
  verbose = simona_opt$verbose
)

alternative_GO_terms(
  tag = c("replaced_by", "alt_id", "consider"),
  version = NULL,
  verbose = TRUE
)
```

Arguments

`namespace`	One of "BP", "CC" and "MF".
`relations`	Types of the GO term relations. In the **GO.db** package, the GO term relations can be "is_a", "part_of", "regulates", "negatively regulates", "positively regulates". Note since "regulates" is a parent relation of "negatively regulates", "positively regulates", if "regulates" is selected, "negatively regulates" and "positively regulates" are also selected. Note "is_a" is always included.

org_db	The name of the organism package or the corresponding database object, e.g. "org.Hs.eg.db" or directly the org.Hs.eg.db::org.Hs.eg.db object for human, then the gene annotation to GO terms will be added to the object. For other non-model organisms, consider to use the **AnnotationHub** package to find one.
evidence_code	A vector of evidence codes for gene annotation to GO terms. See https://geneontology.org/docs/guide-go-evidence-codes/.
retrieve_alternative	Whether to retrieve alternative/obsolete GO terms from geneontology.org?
verbose	Whether to print messages.
tag	In the go-basic.obo file, there are three tags which define alternative GO terms: replaced_by, alt_id and consider. See https://owlcollab.github.io/oboformat/doc/GO.format.obo-1_4.html#S.2.2.1
version	Version of the go-basic.obo file. By default it is the version for building **GO.db** package. The value is a string in the format of "2024-01-17".

Value

An ontology_DAG object.

A list of named vectors where names are alternative GO IDs and value vectors are current GO IDs in use.

Examples

```
dag = create_ontology_DAG_from_GO_db()
dag
```

---

create_ontology_DAG_from_igraph

Create the ontology_DAG object from the igraph object

Description

Create the ontology_DAG object from the igraph object

Usage

```
create_ontology_DAG_from_igraph(
  g,
  relations = NULL,
  verbose = simona_opt$verbose
)
```

Arguments

g	An igraph::igraph object.
relations	A vector of relation types. The length of the vector should be the same as the number of edges in g.
verbose	Whether to print messages.

Value

An ontology_DAG object.

---

dag_all_terms *Names of all terms*

Description

Names of all terms

Usage

```
dag_all_terms(dag)

dag_n_terms(dag)

dag_n_relations(dag)

dag_n_leaves(dag)
```

Arguments

dag	An ontology_DAG object.

Value

dag_all_terms() returns a vector of term names. dag_n_terms() returns a single iteger.

Examples

```
parents  = c("a", "a", "b", "b", "c", "d")
children = c("b", "c", "c", "d", "e", "f")
dag = create_ontology_DAG(parents, children)
dag_all_terms(dag)
dag_n_terms(dag)
```

dag_as_igraph *Convert to an igraph object*

Description

Convert to an igraph object

Usage

```
dag_as_igraph(dag)
```

Arguments

dag	An ontology_DAG object.

Details

If relations is already set in create_ontology_DAG(), relations are also set as an edge attribute in the igraph::igraph object.

Value

An igraph::igraph object.

Examples

```
parents  = c("a", "a", "b", "b", "c", "d")
children = c("b", "c", "c", "d", "e", "f")
dag = create_ontology_DAG(parents, children)
dag_as_igraph(dag)
```

dag_circular_viz *Visualize the DAG*

Description

Visualize the DAG

Usage

```
dag_circular_viz(
  dag,
  highlight = NULL,
  start = 0,
  end = 360,
  partition_by_level = 1,
  partition_by_size = NULL,
  node_col = NULL,
  node_transparency = 0.4,
  node_size = NULL,
  edge_col = NULL,
  edge_transparency = default_edge_transparency(dag),
  legend_labels_from = NULL,
  legend_labels_max_width = 50,
  other_legends = list(),
  use_raster = dag_n_terms(dag) > 10000,
  newpage = TRUE,
  verbose = simona_opt$verbose
)

dag_as_DOT(
  dag,
  node_param = default_node_param,
  edge_param = default_edge_param,
  rankdir = c("TB", "LR", "BT", "RL")
)

dag_graphviz(
  dag,
  node_param = default_node_param,
  edge_param = default_edge_param,
  rankdir = "TB",
  ...
)
```

Arguments

dag	An ontology_Dag object.
highlight	A vector of terms to be highlighted on the DAG.
start	Start of the circle, measured in degree.
end	End of the circle, measured in degree.
partition_by_level	If node_col is not set, users can cut the DAG into clusters with different node colors. The partitioning is applied by partition_by_level().
partition_by_size	Similar as partition_by_level, but the partitioning is applied by partition_by_size().

node_col
: Colors of nodes. If the value is a vector, the order should correspond to terms in dag_all_terms().

node_transparency
: Transparency of nodes. The same format as node_col.

node_size
: Size of nodes. The same format as node_col.

edge_col
: A named vector where names correspond to relation types.

edge_transparency
: A named vector where names correspond to relation types.

legend_labels_from
: If partitioning is applied on the DAG, a legend is generated showing different top terms. By default, the legend labels are the term IDs. If there are additionally column stored in the meta data frame of the DAG object, the column name can be set here to replace the term IDs as legend labels.

legend_labels_max_width
: Maximal width of legend labels measured by the number of characters per line. Labels are wrapped into multiple lines if the widths exceed it.

other_legends
: A list of legends generated by ComplexHeatmap::Legend().

use_raster
: Whether to first write the circular image into a temporary png file, then add to the plot as a raster object?

newpage
: Whether call grid::grid.newpage() to create a new plot?

verbose
: Whether to print messages.

node_param
: A list of parameters. Each parameter has the same format. The value can be a single scalar, a full length vector with the same order as in dag_all_terms(), or a named vector that contains a subset of terms that need to be customized. The full set of parameters can be found at https://graphviz.org/docs/nodes/.

edge_param
: A list of parameters. Each parameter has the same format. The value can be a single scalar, or a named vector that contains a subset of terms that need to be customized. The full set of parameters can be found at https://graphviz.org/docs/edges/. If the parameter is set to a named vector, it can be named by relation types c("is_a" = ...), or directly relations c("a -> b" = ...). Please see the vignette for details.

rankdir
: The direction of the layout. Only four values are allowed: "TB", "LR", "BT" and "RL".

...
: Pass to DiagrammeR::grViz().

Details

dag_circular_viz() uses a circular layout for visualizing large DAGs. dag_graphviz() uses a hierarchical layout for visualizing small DAGs.

dag_as_DOT() generates the DOT code of the DAG.

dag_graphviz() visualizes the DAG with the **DiagrammeR** package.

Value

dag_as_DOT() returns a vector of DOT code.

See Also

http://magjac.com/graphviz-visual-editor/ is nice place to try the DOT code.

Examples

```
dag = create_ontology_DAG_from_GO_db()
dag_circular_viz(dag)

1
if(interactive()) {
dag = create_ontology_DAG_from_GO_db()
dag_graphviz(dag[, "GO:0010228"])
dag_graphviz(dag[, "GO:0010228"],
    edge_param = list(color = c("is_a" = "purple", "part_of" = "darkgreen"),
                      style = c("is_a" = "solid", "part_of" = "dashed")),
    width = 800, height = 800)

# the DOT code for graphviz
dag_as_DOT(dag[, "GO:0010228"])
}
```

---

dag_depth *Depth and height in the DAG*

Description

Depth and height in the DAG

Usage

```
dag_depth(dag, terms = NULL, use_cache = TRUE)

dag_height(dag, terms = NULL, use_cache = TRUE)

dag_shortest_dist_from_root(dag, terms = NULL, use_cache = TRUE)

dag_shortest_dist_to_leaves(dag, terms = NULL, use_cache = TRUE)
```

Arguments

dag	An ontology_DAG object.
terms	A vector of term names. If it is set, the returned vector will be subsetted to the terms that have been set here.
use_cache	Internally used.

Details

The depth of a term in the DAG is defined as the maximal distance from the root. The height of a term in the DAG is the maximal finite distance to all leaf terms.

`dag_shortest_dist_from_root()` and `dag_shortest_dist_to_leaves()` calculate the minimal distance from the root or to the leaves. The word "from" and "to" emphasize the distancer is directinoal.

Value

An integer vector with length the same as the number of total terms in the DAG.

Examples

```
parents  = c("a", "a", "b", "b", "c", "d")
children = c("b", "c", "c", "d", "e", "f")
dag = create_ontology_DAG(parents, children)
dag_depth(dag)
dag_height(dag)
dag_shortest_dist_from_root(dag)
dag_shortest_dist_to_leaves(dag)
```

---

dag_distinct_ancestors

Distinct ancestors of a list of terms

Description

For a given list of terms, it returns a subset of terms which have no ancestor relations to each other.

Usage

```
dag_distinct_ancestors(
  dag,
  terms,
  in_labels = TRUE,
  verbose = simona_opt$verbose
)
```

Arguments

`dag`	An `ontology_DAG` object.
`terms`	A vector of term names.
`in_labels`	Whether the terms are represented in their names or as integer indices?
`verbose`	Whether to print messages.
	Consider a subgraph that contains `terms` and their offspring terms, induced from the complete DAG. the returned subset of terms are those with zero in-degree, or have no finite directional distance from others in the subgraph.

Value

An integer vector or a character vector depending on the value of `in_labels`.

Examples

```
parents  = c("a", "a", "b", "b", "c", "d")
children = c("b", "c", "c", "d", "e", "f")
dag = create_ontology_DAG(parents, children)
dag_distinct_ancestors(dag, c("c", "d", "e", "f"))
```

---

dag_enrich_on_items *Enrichment analysis on the number of annotated items*

Description

The analysis task is to evaluate which terms the given items are enriched to.

Usage

```
dag_enrich_on_items(dag, items, min_hits = 5, min_items = 10)

dag_enrich_on_genes(dag, genes, min_hits = 5, min_genes = 10)
```

Arguments

`dag`	An `ontology_DAG` object.
`items`	A vector of item names.
`min_hits`	Minimal number of items in the term set.
`min_items`	Minimal size of the term set.
`genes`	A vector of gene IDs. The gene ID type can be found by directly printing the `ontology_DAG` object.
`min_genes`	Minimal number of genes.

Details

The function tests whether the list of items are enriched in terms on the DAG. The test is based on the hypergeometric distribution. In the following 2x2 contigency table, `S` is the set of `items`, for a term `t` in the DAG, `T` is the set of items annotated to `t` (by automatically merging from its offspring terms), the aim is to test whether `S` is over-represented in `T`.

The universal set `all` correspond to the full set of items annotated to the DAG.

```
+----------+------+----------+-----+
|          | in S | not in S | all |
+----------+------+----------+-----+
| in T     |  x11 |    x12   | x10 |
```

```
| not in T |  x21 |    x22   | x20 |
+----------+------+----------+-----+
| all      |  x01 |    x02   |  x  |
+----------+------+----------+-----+
```

dag_enrich_on_genes() is the same as dag_enrich_on_items() which only changes the argument item to gene.

Value

A data frame with the following columns:

- term: Term names.
- n_hits: Number of items in items intersecting to t's annotated items.
- n_anno: Number of annotated items of t. Specifically for dag_enrich_on_genes(), this column is renamed to n_gs.
- n_items: Number of items in items intersecting to all annotated items in the DAG. Specifically for dag_enrich_on_genes(), this column is renamed to n_genes.
- n_all: Number of all annotated items in the DAG.
- log2_fold_enrichment: Defined as log2(observation/expected).
- z_score: Defined as (observed-expected)/sd.
- p_value: P-values from hypergeometric test.
- p_adjust: Adjusted p-values from the BH method.

The number of rows in the data frame is the same as the number of terms in the DAG.

Examples

```
## Not run:
dag = create_ontology_DAG_from_GO_db(org_db = "org.Hs.eg.db")
items = random_items(dag, 1000)
df = dag_enrich_on_items(dag, items)

## End(Not run)
1
```

---

dag_enrich_on_offsprings

Enrichment analysis on offspring terms

Description

The analysis task is to evaluate how significant a term includes terms.

Usage

```
dag_enrich_on_offsprings(dag, terms, min_hits = 3, min_offspring = 10)
```

Arguments

dag	An ontology_DAG object.
terms	A vector of term names.
min_hits	Minimal number of terms in an offspring set.
min_offspring	Minimal size of the offspring set.

Details

Given a list of terms in terms, the function tests whether they are enriched in a term's offspring terms. The test is based on the hypergeometric distribution. In the following 2x2 contigency table, S is the set of terms, for a term t in the DAG, T is the set of its offspring plus the t itself, the aim is to test whether S is over-represented in T.

If there is a significant p-value, we can say the term t preferably includes terms in term.

```
+----------+------+----------+-----+
|          | in S | not in S | all |
+----------+------+----------+-----+
| in T     |  x11 |    x12   | x10 |
| not in T |  x21 |    x22   | x20 |
+----------+------+----------+-----+
| all      |  x01 |    x02   |  x  |
+----------+------+----------+-----+
```

Value

A data frame with the following columns:

- term: Term names.
- n_hits: Number of terms in terms intersecting to t's offspring terms.
- n_offspring: Number of offspring terms of t (including t itself).
- n_terms: Number of terms in term intersecting to all terms in the DAG.
- n_all: Number of all terms in the DAG.
- log2_fold_enrichment: Defined as log2(observation/expected).
- z_score: Defined as (observed-expected)/sd.
- p_value: P-values from hypergeometric test.
- p_adjust: Adjusted p-values from the BH method.

The number of rows in the data frame is the same as the number of terms in the DAG.

Examples

```
## Not run:
dag = create_ontology_DAG_from_GO_db()
terms = random_terms(dag, 100)
df = dag_enrich_on_offsprings(dag, terms)

## End(Not run)
1
```

dag_enrich_on_offsprings_by_permutation *Enrichment analysis on offspring terms by permutation test*

Description

Enrichment analysis on offspring terms by permutation test

Usage

```
dag_enrich_on_offsprings_by_permutation(
  dag,
  value,
  perm = 1000,
  min_offspring = 10,
  verbose = simona_opt$verbose
)
```

Arguments

dag	An ontology_DAG object.
value	A numeric value. The value should correspond to terms in dag@terms.
perm	Number of permutations.
min_offspring	Minimal size of the offspring set.
verbose	Whether to print messages.

Details

In the function dag_enrich_on_offsprings(), the statistic for testing is the number of terms in each category. Here this funtion makes the testing procedure more general

The function tests whether a term t's offspring terms have an over-represented pattern on values in value. Denote T as the set of t's offspring terms plus t itself, and v as the numeric vector of value, we first calculate a score s based on values in T:

```
s = mean_{terms in T}(v)
```

To construct a random version of s, we randomly sample n_T terms from the DAG where n_T is the size of set T:

```
sr_i = mean_{n_T randomly sampled terms}(v)
```

where index i represents the ith sampling. If we sample k times, the p-value is calculated as:

```
p = sum_{i in 1..k}(I(sr_i > s))/k
```

Value

A data frame with the following columns:

- `term`: Term names.
- `stats`: The statistics of terms.
- `n_offspring`: Number of offspring terms of `t` (including `t` itself).
- `log2_fold_enrichment`: defined as `log2(s/mean)` where `mean` is calculated from random permutation.
- `z_score`: Defined as `(s - mean)/sd` where `mean` and `sd` are calculated from random permutation.
- `p_value`: P-values from permutation test.
- `p_adjust`: Adjusted p-values from the BH method.

The number of rows in the data frame is the same as the number of terms in the DAG.

Examples

```
## Not run:
dag = create_ontology_DAG_from_GO_db()
value = runif(dag_n_terms(dag)) # a set of random values
df = dag_enrich_on_offsprings_by_permutation(dag, value)

## End(Not run)
1
```

---

dag_filter *Filter the DAG*

Description

Filter the DAG

Usage

```
dag_filter(
  dag,
  terms = NULL,
  relations = NULL,
  root = NULL,
  leaves = NULL,
  mcols_filter = NULL,
  namespace = NULL
)
```

Arguments

dag	An ontology_DAG object.
terms	A vector of term names. The sub-DAG will only contain these terms.
relations	A vector of relations. The sub-DAG will only contain these relations. Valid values of "relations" should correspond to the values set in the relations argument in the create_ontology_DAG(). If relations_DAG is already provided, offspring relation types will all be selected. Note "is_a" is always included.
root	A vector of term names which will be used as roots of the sub-DAG. Only these with their offspring terms will be kept. If there are multiple root terms, a super root will be automatically added.
leaves	A vector of leaf terms. Only these with their ancestor terms will be kept.
mcols_filter	Filtering on columns in the meta data frame.
namespace	The prefix before ":" of the term IDs.

Details

If the DAG is reduced into several disconnected parts after the filtering, a super root is automatically added.

Value

An ontology_DAG object.

Examples

```
parents  = c("a", "a", "b", "b", "c", "d")
children = c("b", "c", "c", "d", "e", "f")
dag = create_ontology_DAG(parents, children)
dag_filter(dag, terms = c("b", "d", "f"))
dag_filter(dag, root = "b")
dag_filter(dag, leaves = c("c", "b"))
dag_filter(dag, root = "b", leaves = "e")


dag = create_ontology_DAG_from_GO_db()
dag_filter(dag, relations = "is_a")
```

---

dag_has_terms *Whether the terms exist in the DAG*

Description

Whether the terms exist in the DAG

Usage

```
dag_has_terms(dag, terms)
```

Arguments

dag	An ontology_DAG object.
terms	A vector of term IDs.

Value

A logical vector.

---

dag_longest_dist_to_offspring *Distance from all ancestors/to all offspring in the DAG*

Description

Distance from all ancestors/to all offspring in the DAG

Usage

```
dag_longest_dist_to_offspring(dag, from, terms = NULL, background = NULL)

dag_shortest_dist_to_offspring(dag, from, terms = NULL, background = NULL)

dag_longest_dist_from_ancestors(dag, to, terms = NULL, background = NULL)

dag_shortest_dist_from_ancestors(dag, to, terms = NULL, background = NULL)
```

Arguments

dag	An ontology_DAG object.
from	A single term name or a vector of term names.
terms	A vector of term names. If it is set, the returned vector will be subsetted to the terms that have been set here.
background	A vector of terms. Then the lookup will only be applied in this set of terms.
to	Same format as the from argument.

Details

If from or to is a vector, for a specific, the longest/shortest distance among all from/to terms is taken.

As a special case, when from is the root term, dag_longest_dist_to_offspring() is the same as dag_depth(), and when to are all leaf terms, dag_longest_dist_to_offspring() is the same as dag_height().

Value

An integer vector having length the same as the number of terms in the DAG. If terms are not reachable to the `from` or `to` terms, the corresponding value is -1.

Examples

```
parents  = c("a", "a", "b", "b", "c", "d")
children = c("b", "c", "c", "d", "e", "f")
dag = create_ontology_DAG(parents, children)
dag_longest_dist_from_ancestors(dag, "e")
dag_shortest_dist_from_ancestors(dag, "e")
dag_longest_dist_to_offspring(dag, "b")
```

---

dag_parents *Parent/child/ancestor/offspring terms*

Description

Parent/child/ancestor/offspring terms

Usage

```
dag_parents(dag, term, in_labels = TRUE)

dag_children(dag, term, in_labels = TRUE)

dag_siblings(dag, term, in_labels = TRUE)

dag_ancestors(dag, term, in_labels = TRUE, include_self = FALSE)

dag_offspring(dag, term, in_labels = TRUE, include_self = FALSE)
```

Arguments

`dag`	An `ontology_DAG` object.
`term`	The value can be a vector of multiple term names. If it is a vector, it returns union of the upstream/downstream terms of the selected set of terms. For `dag_siblings()`, the value can only be a single term.
`in_labels`	Whether the terms are represented in their names or as integer indices?
`include_self`	For `dag_offspring()` and `dag_ancestors()`, this controls whether to also include the query term itself.

Value

An integer vector or a character vector depending on the value of `in_labels`.

Examples

```
parents  = c("a", "a", "b", "b", "c", "d")
children = c("b", "c", "c", "d", "e", "f")
dag = create_ontology_DAG(parents, children)
dag_parents(dag, "b")
dag_parents(dag, "c", in_labels = FALSE)
dag_children(dag, "b")
dag_siblings(dag, "c")
dag_ancestors(dag, "e")
dag_ancestors(dag, "b")
```

---

dag_random_tree *Generate a random DAG*

Description

Generate a random DAG

Usage

```
dag_random_tree(
  n_children = 2,
  p_stop = 0,
  max = 2^10 - 1,
  verbose = simona_opt$verbose
)

dag_add_random_children(
  dag,
  p_add = 0.1,
  new_children = c(1, 4),
  add_random_children_fun = NULL,
  verbose = simona_opt$verbose
)

dag_random(
  n_children = 2,
  p_stop = 0,
  max = 2^10 - 1,
  p_add = 0.1,
  new_children = c(1, 4),
  verbose = simona_opt$verbose
)
```

Arguments

n_children	Number of children of a term. The value can also be a vector of length two representing the range of the number of child terms.

p_stop	The probability of a term to stop growing.
max	Maximal number of terms.
verbose	Whether to print messages.
dag	An ontology_DAG object.
p_add	The probability to add children on each term.
new_children	The number or range of numbers of new children if a term is selected to add more children.
add_random_children_fun	A function to randomly add children from the DAG.

Details

dag_random_tree() generates a random DAG tree from the root term. In a certain step of the growing, let's denote the set of all leaf terms as L, then in the next round of growing, floor(length(L)*p_stop) leaf terms stop growing, and for the remaining leaf terms that continue to grow, each term will add child terms with number in uniformly sampled within [n_children[1], n_children[2]]. The growing stops when the total number of terms in the DAG exceeds max.

dag_add_random_children() adds more links in a DAG. Each term is associated with a probability p_add to add new links where the term, if it is selected, is as a parent term, linking to other terms in the DAG. The number of new child terms is controlled by new_children which can be a single number of a range. By default, new child terms of a term t are randomly selected from other terms that are lower than the term t (check the function simona:::add_random_children). The way how to randomly select new child terms for t can be controlled by a self-defined function for the add_random_children_fun argument.

dag_random(): it simply wraps dag_random_tree() and dag_add_random_children().

Value

An ontology_DAG object.

Examples

```
tree = dag_random_tree()
dag = dag_random()
```

---

dag_reorder *Reorder the DAG*

Description

Reorder the DAG

Usage

```
dag_reorder(dag, value, verbose = simona_opt$verbose)

dag_permutate_children(dag, verbose = simona_opt$verbose)
```

Arguments

dag	An ontology_Dag object.
value	A vector of numeric values. See the **Details** section.
verbose	Whether to print messages.

Details

In dag_reorder(), there are two ways to set the value argument. It can be a vector corresponding to all terms (in the same order as in dag_all_terms()) or a vector corresponding to all leaf terms (in the same order as in dag_leaves()). If value corresponds to all terms, the score associates to each term is the average value of all its offspring terms. And if value corresponds to all leaf terms, the score for each term is the average of all its connectable leaves.

The reordering is simply applied on each term to reorder its child terms.

dag_permutate_children() randomly permute child terms under a term.

Value

An ontology_DAG object.

Examples

```
parents  = c("a", "a", "b", "b", "c", "d")
children = c("b", "c", "c", "d", "e", "f")
# by default, c and e locate on the left side, d and f locate on the right side
dag = create_ontology_DAG(parents, children)
dag_children(dag, "b")

# move c and e to the right side of the diagram
dag2 = dag_reorder(dag, value = c(1, 1, 10, 1, 10, 1))
dag_children(dag2, "b")

# we can also only set values for leaf terms
# there are two leaf terms c and e
# we let v(c) > v(e) to move c to the right side of the diagram
dag3 = dag_reorder(dag, value = c(10, 1))
dag_children(dag3, "b")
```

---

dag_root *Root or leaves of the DAG*

Description

Root or leaves of the DAG

Usage

```
dag_root(dag, in_labels = TRUE)

dag_leaves(dag, in_labels = TRUE)

dag_is_leaf(dag, terms)
```

Arguments

dag	An ontology_DAG object.
in_labels	Whether the terms are represented in their names or as integer indices?
terms	A vector of term names.

Value

A character or an integer vector.

Examples

```
parents  = c("a", "a", "b", "b", "c", "d")
children = c("b", "c", "c", "d", "e", "f")
dag = create_ontology_DAG(parents, children)
dag_root(dag)
dag_leaves(dag)
```

---

dag_shiny *A shiny app for the DAG*

Description

A shiny app for the DAG

Usage

```
dag_shiny(dag)
```

Arguments

dag	An ontology_DAG object.

Examples

```
if(FALSE) {
    dag = create_ontology_DAG_from_GO_db()
    dag_shiny(dag)
}
```

dag_treelize *Reduce the DAG to a tree*

Description

Reduce the DAG to a tree

Usage

```
dag_treelize(dag, verbose = simona_opt$verbose)

dag_as_dendrogram(dag)

## S3 method for class 'ontology_tree'
print(x, ...)
```

Arguments

dag	An ontology_DAG object.
verbose	Whether to print messages.
x	An ontology_DAG object.
...	Ignored.

Details

A tree is a reduced DAG where a child only has one parent. The reducing is applied by a breadth-first searching

Starting from the root and on a certain depth (the depth is the maximal distance to root), for every term t on this depth, its child term c and parent-child relation are kept only when depth(c) == depth(t) + 1. If c is selected, it is marked as visited and will not be checked again.

In this way, depths of all terms in the orignal DAG are still identical to the depths in the tree (see the Examples section).

dag_as_dendrogram() coverts the tree to a dendrogram object.

Value

A tree is also an ontology_DAG object.

Examples

```
parents  = c("a", "a", "b", "b", "c", "d")
children = c("b", "c", "c", "d", "e", "f")
dag = create_ontology_DAG(parents, children)
tree = dag_treelize(dag)
d1 = dag_depth(dag)
d2 = dag_depth(tree)
```

```
identical(d1, d2)

dend = dag_as_dendrogram(tree)
dend
```

`group_sim` *Semantic similarity between two groups of terms*

Description

Semantic similarity between two groups of terms

Usage

```
group_sim(
  dag,
  group1,
  group2,
  method,
  control = list(),
  verbose = simona_opt$verbose
)
```

Arguments

`dag`	An `ontology_DAG` object.
`group1`	A vector of term names or a list of term vectors.
`group2`	A vector of term names or a list of term vectors..
`method`	A group similarity method. All available methods are in `all_group_sim_methods()`.
`control`	A list of parameters passing to individual methods. The term similarity method is controlled by `term_sim_method` and the IC method is controlled by `IC_method`. Other term similarity related parameters can also be specified in `control`. See the subsections.
`verbose`	Whether to print messages.

Details

If `annotation` is set in `create_ontology_DAG()` and you want to directly calculate semantic similarity between two annotated items, you can first get the associated terms of the two items by `annotated_terms()`:

```
group1 = annotated_terms(dag, item1)[[1]]
group2 = annotated_terms(dag, item2)[[1]]
group_sim(dag, group1, group2, ...)
```

Value

A numeric scalar, a numeric vector or a matrix depending on the dat type of group1 and group2.

Methods

GroupSim_pairwise_avg:

Denote S(a, b) as the semantic similarity between terms a and b where a is from group1 and b is from group2, The similarity between group1 and group2 is the average similarity of every pair of individual terms in the two groups:

```
group_sim = mean_{a in group1, b in group2}(S(a, b))
```

The term semantic similarity method and the IC method can be set via control argument:

```
group_sim(dag, group1, group2, method = "GroupSim_pairwise_avg"
   control = list(term_sim_method = "Sim_Lin_1998", IC_method = "IC_annotation")`.
```

Other parameters for the term_sim_method can also be set in the control list.
Pape link: doi:10.1093/bioinformatics/btg153.

GroupSim_pairwise_max:

This is the maximal S(a, b) among all pairs of terms in group1 and group2:

```
group_sim = max_{a in group1, b in group2}(S(a, b))
```

The term semantic similarity method and the IC method can be set via control argument:

```
group_sim(dag, group1, group2, method = "GroupSim_pairwise_max"
   control = list(term_sim_method = "Sim_Lin_1998", IC_method = "IC_annotation")`.
```

Other parameters for the term_sim_method can also be set in the control list.
Paper link: doi:10.1109/TCBB.2005.50.

GroupSim_pairwise_BMA:

BMA stands for "best-match average". First define similarity of a term to a group of terms as

```
S(x, group) = max_{y in group}(x, y)
```

which is the most similar terms in group to x.
Then the BMA similarity is calculated as:

```
group_sim = 0.5*(mean_{a in group1}(S(a, group2)) + mean_{b in group2}(S(b, group1)))
```

So it is the average of the similarity of every term in group1 to the whole group2 and every term in group2 to the whole group1.
The term semantic similarity method and the IC method can be set via control argument:

```
group_sim(dag, group1, group2, method = "GroupSim_pairwise_BMA"
   control = list(term_sim_method = "Sim_Lin_1998", IC_method = "IC_annotation")`.
```

Other parameters for the term_sim_method can also be set in the control list.
Paper link: doi:10.1155/2012/975783.

GroupSim_pairwise_BMM:
BMM stands for "best-match max". It is defined as:

```
group_sim = max(mean_{a in group1}(S(a, group2)), mean_{b in group2}(S(b, group1)))
```

The term semantic similarity method and the IC method can be set via `control` argument:

```
group_sim(dag, group1, group2, method = "GroupSim_pairwise_BMM"
   control = list(term_sim_method = "Sim_Lin_1998", IC_method = "IC_annotation")`.
```

Other parameters for the `term_sim_method` can also be set in the `control` list.
Paper link: doi:10.1186/147121057302.

GroupSim_pairwise_ABM:
ABM stands for "average best-match". It is defined as:

```
group_sim = (sum_{a in group1}(S(a, group2)) + sum_{b in group2}(S(b, group1)))/(n1 + n2)
```

where `n1` and `n2` are the number of terms in `group1` and `group2`.
The term semantic similarity method and the IC method can be set via `control` argument:

```
group_sim(dag, group1, group2, method = "GroupSim_pairwise_ABM"
   control = list(term_sim_method = "Sim_Lin_1998", IC_method = "IC_annotation")`.
```

Other parameters for the `term_sim_method` can also be set in the `control` list.
Paper link: doi:10.1186/1471210514284.

GroupSim_pairwise_HDF:
First define the distance of a term to a group of terms:

```
D(x, group) = 1 - S(x, group)
```

Then the Hausdorff distance between two groups are:

```
HDF = max(max_{a in group1}(D(a, group2)), max_{b in group2}(D(b, group1)))
```

This final similarity is:

```
group_sim = 1 - HDF
```

The term semantic similarity method and the IC method can be set via `control` argument:

```
group_sim(dag, group1, group2, method = "GroupSim_pairwise_HDF"
   control = list(term_sim_method = "Sim_Lin_1998", IC_method = "IC_annotation")`.
```

Other parameters for the `term_sim_method` can also be set in the `control` list.

GroupSim_pairwise_MHDF:
Instead of using the maximal distance from a group to the other group, MHDF uses mean distance:

```
MHDF = max(mean_{a in group1}(D(a, group2)), mean_{b in group2}(D(b, group1)))
```

This final similarity is:

```
group_sim = 1 - MHDF
```

The term semantic similarity method and the IC method can be set via `control` argument:

```
group_sim(dag, group1, group2, method = "GroupSim_pairwise_MHDF"
   control = list(term_sim_method = "Sim_Lin_1998", IC_method = "IC_annotation")`.
```

Other parameters for the `term_sim_method` can also be set in the `control` list.
Paper link: doi:10.1109/ICPR.1994.576361.

GroupSim_pairwise_VHDF:
It is defined as:

```
VHDF = 0.5*(sqrt(mean_{a in group1}(D(a, group2)^2)) + sqrt(mean_{b in group2}(D(b, group1)^2)))
group_sim = 1 - VHDF
```

The term semantic similarity method and the IC method can be set via `control` argument:

```
group_sim(dag, group1, group2, method = "GroupSim_pairwise_VHDF"
   control = list(term_sim_method = "Sim_Lin_1998", IC_method = "IC_annotation")`.
```

Other parameters for the `term_sim_method` can also be set in the `control` list.
Paper link: doi:10.1073/pnas.0702965104.

GroupSim_pairwise_Froehlich_2007:
The similarity is:

```
group_sim = exp(-HDF(group1, group2))
```

The term semantic similarity method and the IC method can be set via `control` argument:

```
group_sim(dag, group1, group2, method = "GroupSim_pairwise_Froehlich_2007"
   control = list(term_sim_method = "Sim_Lin_1998", IC_method = "IC_annotation")`.
```

Other parameters for the `term_sim_method` can also be set in the `control` list.
Paper link: doi:10.1186/147121058166.

GroupSim_pairwise_Joeng_2014:
Similar to *VHDF*, but it directly uses the similarity:

```
group_sim = 0.5*(sqrt(mean_{a in group1}(S(a, group2)^2)) + sqrt(mean_{b in group2}(S(b, group1)^2)))
```

The term semantic similarity method and the IC method can be set via `control` argument:

```
group_sim(dag, group1, group2, method = "GroupSim_pairwise_Joeng_2014"
   control = list(term_sim_method = "Sim_Lin_1998", IC_method = "IC_annotation")`.
```

Other parameters for the `term_sim_method` can also be set in the `control` list.
Paper link: doi:10.1109/TCBB.2014.2343963.

GroupSim_SimALN:
It is based on the average distances between every pair of terms in the two groups:

```
exp(-mean_{a in group1, b in group2}(d(a, b)))
```

d(a, b) is the distance between a and b, which can be the shortest distance between the two terms or the longest distnace via LCA.
There is a parameter distance which takes value of "longest_distances_via_LCA" (the default) or "shortest_distances_via_NCA":

```
group_sim(dag, group1, group2, method = "GroupSim_SimALN",
    control = list(distance = "shortest_distances_via_NCA"))
```

Paper link: doi:10.1109/ISCC.2008.4625763.

GroupSim_SimGIC:
Denote A and B as the two sets of ancestors terms of terms in group1 and group2 respectively, the SimGIC is:

```
group_sim = sum_{x in intersect(A, B)}(IC(x))/sum_{x in union(A, B)}(IC(x))
```

IC method can be set via control = list(IC_method = ...).

GroupSim_SimDIC:
Similar as *GroupSim_SimGIC*, it calculates the Dice coeffcient:

```
group_sim = 2*sum_{x in intersect(A, B)}(IC(x))/(sum_{x in A}(IC(x)) + sum_{x in B}(IC(x)))
```

IC method can be set via control = list(IC_method = ...).

GroupSim_SimUIC:
Similar as *GroupSim_SimGIC*, it is calculated as:

```
group_sim = sum_{x in intersect(A, B)}(IC(x))/max(sum_{x in A}(IC(x)), sum_{x in B}(IC(x)))
```

IC method can be set via control = list(IC_method = ...).

GroupSim_SimUI:
It is only based on the number of terms. A is the set of all ancestors of group1 terms and B is the set of all ancestors of group2 terms.

```
group_sim = length(intersect(A, B))/length(union(A, B))
```

GroupSim_SimDB:
It is:

```
group_sim = 2*length(intersect(A, B))/(length(A) + length(B))
```

GroupSim_SimUB:
It is:

```
group_sim = length(intersect(A, B))/max(length(A), length(B))
```

GroupSim_SimNTO:
It is:

```
group_sim = length(intersect(A, B))/min(length(A), length(B))
```

GroupSim_SimCOU:

It is based on the dot product of two vectors `p` and `q` which correspond to terms in `group1` and `group2`. `p` and `q` have the same length as the total number of terms. Value of position `i` in `p` or `q` corresponds to term `t`. The value takes `IC(t)` if `t` is an ancestor of any term in `p` or `q`, and the value takes zero if `t` is not. The similarity betweem `group1` terms and `group2` terms is calculated as:

```
<p,q>/||p||/||q||
```

where `<p,q>` is the dot product between the two, and `||p||` or `||q||` is the norm of the vector. The equation can be written as:

```
group_sim = sum_{x in intersect(A, B)}(IC(x)^2) /
              sqrt(sum_{x in A}(IC(x)^2)) /
              sqrt(sum_{x in B}(IC(x)^2))
```

IC method can be set via `control = list(IC_method = ...)`.

GroupSim_SimCOT:

Similar as *GroupSim_SimCOU*, the similarity is:

```
<p,q>/(||p||^2 + ||q||^2 - <p,q>)
```

And it can be rewritten as:

```
group_sim = sum_{x in intersect(A, B)}(IC(x)^2) /
   (sum_{x in A}(IC(x)^2) + sum_{x in B}(IC(x)^2) - sum_{x in intersect(A, B)}(IC(x)^2))
```

IC method can be set via `control = list(IC_method = ...)`.

GroupSim_SimLP:

It is the longest depth for the terms in `intersect(A, B)`.

```
group_sim = max(depth(intersect(A, B)))
```

GroupSim_Ye_2005:

It is a normalized version of *GroupSim_SimLP*:

```
group_sim = max(depth(intersect(A, B)))/max_depth
```

Since the minimal depth is zero for root.

GroupSim_SimCHO:

It is based on the annotated items. Denote `sigma(t)` as the total annotated items of `t`. The similarity is calculated as

```
group_sim = log(C/sigma_max)/log(sigma_min/sigma_max)
```

where `C` is `min(sigma_{x in intersect(A, B)}(x))`, i.e., the minimal sigma in the intersection of group1 and group2. Note Now `A` and `B` are just two sets of terms in `group1` and `group2`. `sigma_max` is the total number of items annotated to the DAG, `sig_min` is the minimal number of items annotated to a term, which is mostly 1.

GroupSim_SimALD:

A and B are just two sets of terms in group1 and group2. The similarity is calculated as:

```
group_sim = max_{t in intersect(A, B)}(1 - sigma(t)/N)
```

GroupSim_Jaccard:

Say A is the set of items annotated to terms in group1 and B is the set of items annotated to group2. This is the Jaccard coeffcient between two sets.

The universe/background can be set via control = list(universe = ...).

GroupSim_Dice:

It is the Dice coeffcient between A and B.

The universe/background can be set via control = list(universe = ...).

GroupSim_Overlap:

It is the Overlap coeffcient between A and B.

The universe/background can be set via control = list(universe = ...).

GroupSim_Kappa:

The universe/background can be set via control = list(universe = ...).

Examples

```
parents  = c("a", "a", "b", "b", "c", "d")
children = c("b", "c", "c", "d", "e", "f")
annotation = list(
    "a" = c("t1", "t2", "t3"),
    "b" = c("t3", "t4"),
    "c" = "t5",
    "d" = "t7",
    "e" = c("t4", "t5", "t6", "t7"),
    "f" = "t8"
)
dag = create_ontology_DAG(parents, children, annotation = annotation)
group_sim(dag, c("c", "e"), c("d", "f"),
    method = "GroupSim_pairwise_avg",
    control = list(term_sim_method = "Sim_Lin_1998")
)
```

---

import_obo *Import ontology file to an ontology_DAG object*

Description

Import ontology file to an ontology_DAG object

Usage

```
import_obo(
  file,
  relation_type = character(0),
  inherit_relations = TRUE,
  verbose = simona_opt$verbose,
  ...
)

import_owl(
  file,
  relation_type = character(0),
  inherit_relations = TRUE,
  verbose = simona_opt$verbose,
  ...
)

import_ontology(
  file,
  robot_jar = simona_opt$robot_jar,
  JAVA_ARGS = "",
  verbose = simona_opt$verbose,
  ...
)

import_ttl(file, relation_type = "part_of", verbose = simona_opt$verbose, ...)
```

Arguments

`file`	Path of the ontology file or an URL.
`relation_type`	Semantic relation types to include. Note `is_a` relation is always included.
`inherit_relations`	Relations may also be structured as a DAG. It controls whether to merge with a relations's offspring relations.
`verbose`	Whether to print messages.
`...`	Pass to `create_ontology_DAG()`.
`robot_jar`	The path of the `robot.jar` file. It can be downloaded from https://github.com/ontodev/robot/releases. Internally, the file is converted to the obo format and parsed by `import_obo()`. The value of `robot_jar` can be set as a global option `simona_opt$robot_jar = ...`.
`JAVA_ARGS`	Options for `java`. For example you can set `-Xmx20G` if you want to increase the memory to 20G for java.

Details

Public bio-ontologies can be obtained from Ontology Foundry or BioPortal.

The import_obo() function parses the ontology file in .obo format. To parse other formats, external tool robot.jar is required.

import_owl() only recognizes <owl:Class> and <owl:ObjectProperty>. If the .owl file does not contain these tags, please use import_ontology() directly.

robot.jar can automatically recognize the following formats:

- json: OBO Graphs JSON
- obo: OBO Format
- ofn: OWL Functional
- omn: Manchester
- owl: RDF/XML
- owx: OWL/XML
- ttl: Turtle

The description of the ROBOT tool is at http://robot.obolibrary.org/convert.

import_ttl() is a simple parser for the .ttl format files. It only recognizes terms that have the owl:Class object. The "is_a" relation is recognized by the predicate rdfs:subClassOf or an ontology-specific predicate that contains .*/isa. Other relation types are defined with the predicate owl:ObjectProperty. The format is parsed by a Perl script system.file("scripts", "parse_ttl.pl", package = "simona").

Value

An ontology_DAG object.

Examples

```
# The plant ontology: http://obofoundry.org/ontology/po.html
import_obo("https://raw.githubusercontent.com/Planteome/plant-ontology/master/po.obo")


import_owl("http://purl.obolibrary.org/obo/po.owl")

## Not run:
# The plant ontology: http://obofoundry.org/ontology/po.html
dag = import_ontology("http://purl.obolibrary.org/obo/po.owl", robot_jar = ...)

## End(Not run)

# file is from https://bioportal.bioontology.org/ontologies/MSTDE
import_ttl("https://jokergoo.github.io/simona/MSTDE.ttl")
```

mcols,ontology_DAG-method *Get or set meta columns on DAG*

Description

Get or set meta columns on DAG

Usage

```
## S4 method for signature 'ontology_DAG'
mcols(x, use.names = TRUE, ...)

## S4 replacement method for signature 'ontology_DAG'
mcols(x, ...) <- value
```

Arguments

x	An ontology_DAG object.
use.names	Please ignore.
...	Other argument. For mcols(), it can be a vector of column names in the meta data frame.
value	A data frame or a matrix where rows should correspond to terms in x@terms.

Value

A data frame.

Examples

```
parents  = c("a", "a", "b", "b", "c", "d")
children = c("b", "c", "c", "d", "e", "f")
dag = create_ontology_DAG(parents, children)
mcols(dag) = data.frame(id = letters[1:6], v = 1:6)
mcols(dag)
mcols(dag, "id")
dag
```

method_param *All Papameters of a given method*

Description

All Papameters of a given method

Usage

```
method_param(IC_method = NULL, term_sim_method = NULL, group_sim_method = NULL)
```

Arguments

IC_method	A single IC method name.
term_sim_method	A single term similarity method name.
group_sim_method	A single group similarity method name.

Value

A vector of parameter names.

Examples

```
method_param(IC_method = "IC_annotation")
method_param(term_sim_method = "Sim_Wang_2007")
```

MICA_term *Various types of common ancestors*

Description

Various types of common ancestors

Usage

```
MICA_term(
  dag,
  terms,
  IC_method,
  in_labels = TRUE,
  distance = "longest",
  verbose = simona_opt$verbose
)
```

```
MICA_IC(dag, terms, IC_method, verbose = simona_opt$verbose)

LCA_term(
  dag,
  terms,
  in_labels = TRUE,
  distance = "longest",
  verbose = simona_opt$verbose
)

LCA_depth(dag, terms, verbose = simona_opt$verbose)

NCA_term(dag, terms, in_labels = TRUE, verbose = simona_opt$verbose)

max_ancestor_v(dag, terms, value, verbose = simona_opt$verbose)

max_ancestor_id(
  dag,
  terms,
  value,
  in_labels = FALSE,
  distance = "longest",
  verbose = simona_opt$verbose
)

max_ancestor_path_sum(
  dag,
  terms,
  value,
  add_v,
  distance = "longest",
  verbose = simona_opt$verbose
)

CA_terms(dag, term1, term2, in_labels = TRUE)
```

Arguments

dag	An ontology_DAG object.
terms	A vector of term names.
IC_method	An IC method. Valid values are in all_term_IC_methods().
in_labels	Whether the terms are represented in their names or as integer indices?
distance	If there are multiple LCA or MICA of two terms, whether to take the one with the longest distance of shortest distance to the two terms. Possible values are "longest" and "shortest".
verbose	Whether to print messages.
value	A numeric vector. The elements should corrrespond to terms in dag_all_terms() (should have the same length as the number of terms in the DAG).

add_v	Values to be added along the path to the MICA or LCA. The same format as value.
term1	A single term ID.
term2	A single term ID.

Details

There are the following three types of common ancestors:

- MICA (most informative common ancestor): The common ancestor with the highest IC value.
- LCA (lowest common ancestor): The common ancestor with the largest depth (The depth of a term is the maximal distance from the root term). If there are multiple ancestors having the same max depth, the ancestor with the smallest distance to the two terms is used.
- NCA (nearest common ancestor): The common ancestor with the smallest distance to the two terms. If there are multiple ancestors with the same smallest distance, the ancestor with the largest depth is used.

max_ancestor_v() and max_ancestor_id() are more general functions which return common ancestors with the highest value in value.

Given a path connecting two terms and their MICA/LCA, max_ancestor_path_sum() calculates the sum of terms along the path. The values to be added in specified in add_v argument.

Value

- MICA_term() returns an integer or a character matrix of the MICA terms depending on the value of in_labels.
- MICA_IC() returns a numeric matrix of the IC of the MICA terms.
- LCA_term() returns an integer or a character matrix of the LCA term depending on the value of in_labels.
- LCA_depth() returns an integer matrix of the depth of the LCA terms.
- NCA_term() returns an integer or a character matrix of the NCA term depending on the value of in_labels. The shortest distance from NCA terms can be calculated by shortest_distances_via_NCA().
- max_ancestor_v() returns a numeric matrix.
- max_ancestor_id() returns an integer or a character matrix.
- CA_terms() returns a vector of term IDs.

Examples

```
parents  = c("a", "a", "b", "b", "c", "d")
children = c("b", "c", "c", "d", "e", "f")
dag = create_ontology_DAG(parents, children)
MICA_term(dag, letters[1:6], "IC_universal")
MICA_IC(dag, letters[1:6], "IC_universal")
LCA_term(dag, letters[1:6])
LCA_depth(dag, letters[1:6])
NCA_term(dag, letters[1:6])
CA_terms(dag, "c", "d")
```

`n_annotations` *Number of annotated items*

Description

Number of annotated items

Usage

```
n_annotations(
  dag,
  terms = NULL,
  uniquify = simona_opt$anno_uniquify,
  use_cache = simona_opt$use_cache
)

has_annotation(dag)
```

Arguments

`dag`	An `ontology_DAG` object.
`terms`	A vector of term names. If it is set, the returned vector will be subsetted to the terms that have been set here.
`uniquify`	Whether to uniquify items that are annotated to the term? See **Details**. It is suggested to always be `TRUE`.
`use_cache`	Internally used.

Details

Due to the nature of the DAG, a parent term includes all annotated items of its child terms, and an ancestor term includes all annotated items from its offspring recursively. In current tools, there are two different implementations to deal with such recursive merging.

For a term `t`, denote `S_1`, `S_2`, ... as the sets of annotated items for its child 1, 2, ..., also denote `S_t` as the set of items that are **directly** annotated to `t`. The first method takes the union of annotated items on `t` and all its child terms:

```
n = length(union(S_t, S_1, S_2, ...))
```

And the second method takes the sum of numbers of items on `t` and on all its child terms:

```
n = sum(length(s_t) + length(S_1) + length(S_2) + ...)
```

In `n_annotations()`, when `uniquify = TRUE`, the first method is used; and when `uniquify = FALSE`, the second method is used.

For some annotation sources, it is possible that an item is annotated to multiple terms, thus, the second method which simply adds numbers of all its child terms may not be proper because an item

may be counted duplicatedly, thus over-estimating n. The two methods are identical only if an item is annotated to a unique term in the DAG.

We suggest to always set uniquify = TRUE (the default), and the scenario of uniquify = FALSE is only for the testing or benchmarking purpose.

Value

n_annotations() returns an integer vector.

has_annotation() returns a logical scalar.

Examples

```
parents  = c("a", "a", "b", "b", "c", "d")
children = c("b", "c", "c", "d", "e", "f")
annotation = list(
    "a" = c("t1", "t2", "t3"),
    "b" = c("t3", "t4"),
    "c" = "t5",
    "d" = "t7",
    "e" = c("t4", "t5", "t6", "t7"),
    "f" = "t8"
)
dag = create_ontology_DAG(parents, children, annotation = annotation)
n_annotations(dag)
```

---

n_offspring *Number of parent/child/ancestor/offspring/leaf terms*

Description

Number of parent/child/ancestor/offspring/leaf terms

Usage

```
n_offspring(dag, terms = NULL, use_cache = TRUE, include_self = FALSE)

n_ancestors(dag, terms = NULL, use_cache = TRUE, include_self = FALSE)

n_connected_leaves(dag, terms = NULL, use_cache = TRUE)

n_parents(dag, terms = NULL)

n_children(dag, terms = NULL)

avg_parents(dag)

avg_children(dag)
```

Arguments

dag	An ontology_DAG object.
terms	A vector of term names. If the value is NULL, it returns for all terms in the DAG.
use_cache	Internally used.
include_self	For n_offspring() and n_ancestors(), this controls whether to also include the query term itself.

Details

For n_connected_leaves(), leaf nodes have value of 1.

In avg_parents(), root term is removed.

In avg_children(), leaf term is removed.

Value

An integer vector.

Examples

```
parents  = c("a", "a", "b", "b", "c", "d")
children = c("b", "c", "c", "d", "e", "f")
dag = create_ontology_DAG(parents, children)
n_parents(dag)
n_children(dag)
n_offspring(dag)
n_ancestors(dag)
n_connected_leaves(dag)
```

---

ontology_DAG-class *The ontology_DAG class*

Description

This class defines the DAG structure of an ontology.

Value

An ontology_DAG object.

Slots

terms A character vector of length n of all term names. Other slots that store term-level information use the integer indices of terms.

n_terms An integer scalar of the total number of terms in the DAG.

n_relations An integer scalar of the total number of relations in the DAG.

`lt_parents` A list of length `n`. Each element in the list is an integer index vector of the parent terms of the ith term.

`lt_children` A list of length `n`. Each element in the list is an integer index vector of the child terms of the ith term.

`lt_children_relations` A list of length `n`. Each element is a vector of the semantic relations between the ith term and its child terms, e.g. a child "is_a" parent. The relations are represented as integers. The character name of the relations is in `attr(dag@lt_children_relations, "levels")`.

`relations_DAG` A simple `ontology_DAG` object but constructed for relation types.

`source` The source of the ontology. A character scalar only used as a mark of the returned object.

`root` An integer scalar of the root term.

`leaves` An integer vector of the indicies of leaf terms.

`alternative_terms` A named character vector of mappings between alternative terms to DAG terms.

`tpl_sorted` An integer vector of reordered term indices which has been topologically sorted in the DAG. Terms are sorted first by the depth (maximal distance from root), then the number of child terms, then the number of parent terms, and last the term names.

`tpl_pos` The position of the original term in the topologically sorted path (similar as the rank), e.g. the value of the first element in the vector is the position of term 1 in the topologically sorted path.

`annotation` A list of two elements: `list` and `names`. The `dag@annotation$list` element contains a list of length `n` and each element is a vector of integer indices of annotated items. The full list of annotated items is in `dag@annotation$names`.

`term_env` An environment which contains various term-level statistics. It is mainly for cache purpose.

`aspect_ratio` A numeric vector of length two. The aspect ratio is calculated as `w/h`. For each term, there is a distance to root, `h` is the maximal distance of all terms, `w` is the maximal number of items with the same distance. The two values in the `aspect_ratio` slot use maximal distance to root (the height) and the shortest distance to root as the distance measure.

`elementMetadata` An additional data frame with the same number of rows as the number of terms in DAG. Order of rows should be the same as order of terms in `dag@terms`.

Examples

```
1
# This function should not be used directly.
```

`ontology_kw` *Import ontologies already having gene annotations*

Description

Import ontologies already having gene annotations

Usage

```
ontology_kw(
  organism = "human",
  gene_annotation = TRUE,
  verbose = simona_opt$verbose,
  ...
)

ontology_chebi(
  organism = c("human", "mouse", "rat", "pig", "dog"),
  gene_annotation = TRUE,
  verbose = simona_opt$verbose,
  ...
)

ontology_hp(
  organism = c("human", "mouse"),
  gene_annotation = TRUE,
  verbose = simona_opt$verbose,
  ...
)

ontology_pw(
  organism = c("human", "mouse", "rat", "pig", "dog", "chimpanzee"),
  gene_annotation = TRUE,
  verbose = simona_opt$verbose,
  ...
)

ontology_rdo(
  organism = c("human", "mouse", "rat", "pig", "dog", "chimpanzee"),
  gene_annotation = TRUE,
  verbose = simona_opt$verbose,
  ...
)

ontology_vt(
  organism = c("human", "mouse", "rat", "pig", "dog", "chimpanzee"),
  gene_annotation = TRUE,
  verbose = simona_opt$verbose,
  ...
)

ontology_go(...)

ontology_reactome(
  organism = "HSA",
  gene_annotation = TRUE,
```

```
  verbose = simona_opt$verbose,
  ...
)
```

Arguments

organism	Organism.
gene_annotation	Whether to add gene annotations to the DAG.
verbose	Whether to print messages?
...	Pass to create_ontology_DAG().

Details

There are the following ontologies:

- ontology_kw(): UniProt Keywords. The list of supported organisms can be found in UniProtKeywords::load_keywo
- ontology_chebi(): Chemical Entities of Biological Interest.
- ontology_hp(): The Human Phenotype Ontology.
- ontology_pw(): Pathway Ontology.
- ontology_rdo(): RGD Disease Ontology.
- ontology_vt(): Vertebrate Trait Ontology.

The source of the original files can be found with simona:::RGD_TB.

ontology_go() is an alias of create_ontology_DAG_from_GO_db(). All arguments go there.

Valid values for organism argument in ontology_reactome() are

```
c("BTA", "CEL", "CFA", "DRE", "DDI", "DME", "GGA", "HSA", "MMU",
  "MTU", "PFA", "RNO", "SCE", "SPO", "SSC", "XTR")
```

---

partition_by_level *Partition the DAG*

Description

Partition the DAG

Usage

```
partition_by_level(dag, level = 1, from = NULL, term_pos = NULL)

partition_by_size(dag, size = round(dag_n_terms(dag)/5))
```

Arguments

dag	An ontology_DAG object.
level	Depth in the DAG to cut. The DAG is cut below terms (or cut the links to their child terms) with depth == level.
from	A list of terms to cut. If it is set, level is ignored.
term_pos	Internally used.
size	Number of terms in a cluster. The splitting stops on a term if all its child-trees are smaller than size.

Details

Let's call the terms below the from term as "top terms" because they will be on top of the sub-DAGs after the partitioning. It is possible that a term in the middle of the DAG can be traced back to more than one top terms. To partition all terms exclusively, a term partitioned to the sub-DAG from the top term with the largest distance to the term. If a term has the same largest distances to several top terms, a random top term is selected.

In partition_by_size(), the DAG is first reduced to a tree where a child term only has one parent. The partition is done recursively by cutting into its child-trees. The splitting stops when all the child-trees have size less than size.

NA is assigned to the from terms, their ancestor terms, and terms having infinite directed distance to from terms.

Value

A character vector of top terms in each partition.

Examples

```
dag = create_ontology_DAG_from_GO_db()
pa = partition_by_level(dag)
table(pa)
pa = partition_by_size(dag, size = 1000)
table(pa)
```

```
1
```

---

print.print_source *Print the source*

Description

Print the source

Usage

```
## S3 method for class 'print_source'
print(x, ...)
```

Arguments

x	An object in the print_source class.
...	Other arguments.

Details

Internally used.

---

random_terms *Randomly sample terms/items*

Description

Randomly sample terms/items

Usage

```
random_terms(dag, n)

random_items(dag, n)
```

Arguments

dag	An ontology_DAG object.
n	Number of terms or items.

Value

A character vector of terms or items.

Examples

```
parents  = c("a", "a", "b", "b", "c", "d")
children = c("b", "c", "c", "d", "e", "f")
annotation = list(
    "a" = c("t1", "t2", "t3"),
    "b" = c("t3", "t4"),
    "c" = "t5",
    "d" = "t7",
    "e" = c("t4", "t5", "t6", "t7"),
    "f" = "t8"
)
dag = create_ontology_DAG(parents, children, annotation = annotation)
random_terms(dag, 3)
random_items(dag, 3)
```

shortest_distances_via_NCA *Distance on the DAG*

Description

Distance on the DAG

Usage

```
shortest_distances_via_NCA(dag, terms, verbose = simona_opt$verbose)

longest_distances_via_LCA(dag, terms, verbose = simona_opt$verbose)

shortest_distances_directed(dag, terms, verbose = simona_opt$verbose)

longest_distances_directed(dag, terms, verbose = simona_opt$verbose)
```

Arguments

dag	An ontology_DAG object.
terms	A vector of term names.
verbose	Whether to print messages.

Details

Denote two terms as a and b, a common ancestor as c, and the distance function d() calculates the longest distance or the shortest distance depending on the function.

- shortest_distances_via_NCA(): It calculates the smallest d(c, a) + d(c, b) where d() calculates the shortest distance between two terms. In this case, c is the NCA (nearest common ancestor) of a and b.
- longest_distances_via_LCA(): It calculates the largest d(c, a) + d(c, b) where d() calculates the longest distance between two terms *via the LCA (lowest common ancestor) term*. In this case, c is the LCA of a and b.
- shortest_distances_directed(): It calculates d(a, b) where d() calculates the shortest distance between two terms. The distance is only calculated when a is an ancestor of b, otherwise the distance value is -1.
- longest_distances_directed(): It calculates d(a, b) where d() calculates the longest distance between two terms. The distance is only calculated when a is an ancestor of b, otherwise the distance value is -1.

Value

A numeric distance matrix.

Examples

```
parents  = c("a", "a", "b", "b", "c", "d")
children = c("b", "c", "c", "d", "e", "f")
dag = create_ontology_DAG(parents, children)
shortest_distances_via_NCA(dag, letters[1:6])
longest_distances_via_LCA(dag, letters[1:6])
shortest_distances_directed(dag, letters[1:6])
longest_distances_directed(dag, letters[1:6])
```

---

show,ontology_DAG-method *Print the ontology_DAG object*

Description

Print the ontology_DAG object

Usage

```
## S4 method for signature 'ontology_DAG'
show(object)
```

Arguments

object	An ontology_DAG object.

Value

No value is returned.

---

simona_opt *Global options*

Description

Global options

Usage

```
simona_opt(..., RESET = FALSE, READ.ONLY = NULL, LOCAL = FALSE, ADD = FALSE)
```

Arguments

...	Name-value pairs for options.
RESET	Reset to default option values.
READ.ONLY	Only return read only options.
LOCAL	Only return local options.
ADD	Add new options.

Details

There are the following global options:

- use_cache: By default, information content of all terms is cached and reused. If use_cache is set to FALSE, IC will be re-calculated.
- verbose: Whether to print messages?
- anno_uniquify: In the annotation-based IC method, the union of items annotated to the term as well as all its offspring terms is used, which means the set of annotated items for the term is uniquified. If anno_uniquify is set to FALSE, the uniquification is not applied, we simply add the number of items annotated to the term and the numbers of items annotated to each of its offspring terms.
- robot_jar: Path of the robot.jar file. The file can be found from https://github.com/ontodev/robot/releases.

To set an option, you can use $:

```
simona_opt$verbose = FALSE
```

or use it as a function:

```
simona_opt(verbose = FALSE)
```

Value

A single option value.

Examples

```
simona_opt
```

`term_annotations` *Term-item associations*

Description

Term-item associations

Usage

```
term_annotations(dag, terms, return = "list")

annotated_terms(dag, anno, return = "list")
```

Arguments

`dag`	An `ontology_DAG` object.
`terms`	A vector of term names.
`return`	Whether the returned object is a list or a matrix?
`anno`	A vector of annotated item names.

Details

If an item is annotated to a term, all this term's ancestor terms are also annotated.

Value

A list or a binary matrix showing annotation relations between terms and items.

Examples

```
parents  = c("a", "a", "b", "b", "c", "d")
children = c("b", "c", "c", "d", "e", "f")
annotation = list(
    "a" = c("t1", "t2", "t3"),
    "b" = c("t3", "t4"),
    "c" = "t5",
    "d" = "t7",
    "e" = c("t4", "t5", "t6", "t7"),
    "f" = "t8"
)
dag = create_ontology_DAG(parents, children, annotation = annotation)
term_annotations(dag, letters[1:6])
term_annotations(dag, letters[1:6], return = "matrix")
annotated_terms(dag, c("t1", "t2", "t3"))
annotated_terms(dag, c("t1", "t2", "t3"), return = "matrix")
```

term_IC *Information content*

Description

Information content

Usage

```
term_IC(
  dag,
  method,
  terms = NULL,
  control = list(),
  verbose = simona_opt$verbose
)
```

Arguments

dag	An ontology_DAG object.
method	An IC method. All available methods are in all_term_IC_methods().
terms	A vector of term names. If it is set, the returned vector will be subsetted to the terms that have been set here.
control	A list of parameters passing to individual methods. See the subsections.
verbose	Whether to print messages.

Value

A numeric vector.

Methods

IC_offspring:

Denote k as the number of offspring terms plus the term itself and N is such value for root (or the total number of terms in the DAG), the information content is calculated as:

```
IC = -log(k/N)
```

IC_height:

For a term t in the DAG, denote d as the maximal distance from root (i.e. the depth) and h as the maximal distance to leaves (i.e. the height), the relative position p on the longest path from root to leaves via term t is calculated as:

```
p = (h + 1)/(h + d + 1)
```

In the formula where 1 is added gets rid of p = 0. Then the information content is:

```
IC = -log(p)
   = -log((h+1)/(h+d+1))
```

IC_annotation:

Denote k as the number of items annotated to a term t, and N is the number of items annotated to the root (which is the total number of items annotated to the DAG), IC for term t is calculated as:

```
IC = -log(k/N)
```

In current implementations in other tools, there is an inconsistency of defining k and N. Please see n_annotations() for explanation.

NA is assigned to terms with no item annotated.

IC_universal:

It measures the probability of a term getting full transmission from the root. Each term is associated with a p-value and the root has the p-value of 1.

For example, an intermediate term t has two parent terms parent1 and parent2, also assume parent1 has k1 children and parent2 has k2 children, assume a parent transmits information equally to all its children, then respectively parent1 only transmits 1/k1 and parent2 only transmits 1/k2 of its content to term t, or the probability of a parent to reach t is 1/k1 or 1/k2. Let's say p1 and p2 are the accumulated contents from the root to parnet1 and parent2 respectively (or the probability of the two parent terms getting full transmission from root), then the probability of reaching t via a full transmission graph from parent1 is the multiplication of p1 and 1/k1, which is p1/k1, and same for p2/k2. Then, for term t, if getting transmitted from parent1 and parent2 are independent, the probability of t (denoted as p_t) to get transmitted from both parents is:

```
p_t = (p1/k1) * (p2/k2)
```

Since the two parents are the full set of t's parents, p_t is the probability of t getting full transmission from root. And the final information content is:

```
IC = -log(p_t)
```

Paper link: doi:10.1155/2012/975783.

IC_Zhang_2006:

It measures the number of ways from a term to reach leaf terms. E.g. in the following DAG:

```
    a      upstream
   /|\
  b | c
    |/
    d      downstream
```

term a has three ways to reach leaf, which are a->b, a->d and a->c->d.

Let's denote k as the number of ways for term t to reach leaves and N as the maximal value of k which is associated to the root term, the information content is calculated as

```
IC = -log(k/N)
   = log(N) - log(k)
```

Paper link: doi:10.1186/147121057135.

IC_Seco_2004:
It is based on the number of offspring terms of term `t`. The information content is calculated as:

```
IC = 1 - log(k+1)/log(N)
```

where `k` is the number of offspring terms of `t`, or you can think `k+1` is the number of `t`'s offspring terms plus itself. `N` is the total number of terms on the DAG.
Paper link: doi:10.5555/3000001.3000272.

IC_Zhou_2008:
It is a correction of *IC_Seco_2004* which considers the depth of a term in the DAG. The information content is calculated as:

```
IC = 0.5*IC_Seco + 0.5*log(depth)/log(max_depth)
```

where `depth` is the depth of term `t` in the DAG, defined as the maximal distance from root. `max_depth` is the largest depth in the DAG. So IC is composed with two parts: the numbers of offspring terms and positions in the DAG.
Paper link: doi:10.1109/FGCNS.2008.16.

IC_Seddiqui_2010:
It is also a correction to *IC_Seco_2004*, but considers number of relations connecting a term (i.e. number of parent terms and child terms). The information content is defined as:

```
(1-sigma)*IC_Seco + sigma*log((n_parents + n_children + 1)/log((total_edges + 1))
```

where `n_parents` and `n_children` are the numbers of parents and children of term `t`. The tuning factor `sigma` is defined as

```
sigma = log(total_edges+1)/(log(total_edges) + log(total_terms))
```

where `total_edges` is the number of all relations (all parent-child relations) and `total_terms` is the number of all terms in the DAG.
Paper link: doi:10.5555/1862330.1862343.

IC_Sanchez_2011:
It measures the average contribution of term `t` on leaf terms. First denote `zeta` as the number of leaf terms that can be reached from term `t` (or `t`'s offspring that are leaves.). Since all `t`'s ancestors can also reach `t`'s leaves, the contribution of `t` on leaf terms is scaled by `n_ancestors` which is the number of `t`'s ancestor terms. The final information content is normalized by the total number of leaves in the DAG, which is the possible maximal value of `zeta`. The complete definition of information content is:

```
IC = -log( (zeta/n_ancestor) / n_all_leaves)
```

Paper link: doi:10.1016/j.knosys.2010.10.001.

IC_Meng_2012:
It has a complex form which takes account of the term depth and the downstream of the term. The first factor is calculated as:

```
f1 = log(depth)/long(max_depth)
```

The second factor is calculated as:

```
f1 = 1 - log(1 + sum_{x => t's offspring}(1/depth_x))/log(total_terms)
```

In the equation, the summation goes over t's offspring terms.
The final information content is the multiplication of f1 and f2:

```
IC = f1 * f2
```

Paper link: http://article.nadiapub.com/IJGDC/vol5_no3/6.pdf.
There is one parameter correct. If it is set to TRUE, the first factor f1 is calculated as:

```
f1 = log(depth + 1)/long(max_depth + 1)
```

correct can be set as:

```
term_IC(dag, method = "IC_Meng_2012", control = list(correct = TRUE))
```

IC_Wang_2007:
Each relation is weighted by a value less than 1 based on the semantic relation, i.e. 0.8 for "is_a" and 0.6 for "part_of". For a term t and one of its ancestor term a, it first calculates an "S-value" which corresponds to a path from a to t where the accumulated multiplication of weights along the path reaches maximal:

```
S(a->t) = max_{path}(prod_{node on the paty}(w))
```

Here max goes over all possible paths from a to t, and prod() multiplies edge weights in a certain path.
The formula can be transformed as (we simply rewrite S(a->t) to S):

```
1/S = min(prod(1/w))
log(1/S) = log(min(prod(1/w)))
         = min(sum(log(1/w)))
```

Since w < 1, log(1/w) is positive. According to the equation, the path (a->...->t) is actually the shortest path from a to t by taking log(1/w) as the weight, and log(1/S) is the weighted shortest distance.
If S(a->t) can be thought as the maximal semantic contribution from a to t, the information content is calculated as the sum from all t's ancestors (including t itself):

```
IC = sum_{a in t's ancestors + t}(S(a->t))
```

Paper link: doi:10.1093/bioinformatics/btm087.
The contribution of different semantic relations can be set with the contribution_factor parameter. The value should be a named numeric vector where names should cover the relations defined in relations set in create_ontology_DAG(). For example, if there are two relations "relation_a" and "relation_b" set in the DAG, the value for contribution_factor can be set as:

```
term_IC(dag, method = "IC_Wang",
   control = list(contribution_factor = c("relation_a" = 0.8, "relation_b" = 0.6)))
```

Note the **IC_Wang_2007** method is normally used within the **Sim_Wang_2007** semantic similarity method.

Examples

```
parents  = c("a", "a", "b", "b", "c", "d")
children = c("b", "c", "c", "d", "e", "f")
annotation = list(
    "a" = c("t1", "t2", "t3"),
    "b" = c("t3", "t4"),
    "c" = "t5",
    "d" = "t7",
    "e" = c("t4", "t5", "t6", "t7"),
    "f" = "t8"
)
dag = create_ontology_DAG(parents, children, annotation = annotation)
term_IC(dag, "IC_annotation")
```

---

term_sim *Semantic similarity*

Description

Semantic similarity

Usage

```
term_sim(dag, terms, method, control = list(), verbose = simona_opt$verbose)
```

Arguments

dag	An ontology_DAG object.
terms	A vector of term names.
method	A term similarity method. All available methods are in all_term_sim_methods().
control	A list of parameters passing to individual methods. See the subsections.
verbose	Whether to print messages.

Value

A numeric symmetric matrix.

Methods

Sim_Lin_1998:

The similarity between two terms a and b is calculated as the IC of their MICA term c normalized by the average of the IC of the two terms:

```
sim = IC(c)/((IC(a) + IC(b))/2)
    = 2*IC(c)/(IC(a) + IC(b))
```

Although any IC method can be used here, in more applications, it is normally used together with the *IC_annotation* method.
Paper link: doi:10.5555/645527.657297.

Sim_Resnik_1999:
The IC method is fixed to `IC_annotation`.
The original Resnik similarity is the IC of the MICA term. There are three ways to normalize the Resnik similarity into the scale of `[0, 1]`:

1. *Nunif*

```
sim = IC(c)/log(N)
```

where `N` is the total number of items annotated to the whole DAG, i.e. number of items annotated to the root. Then the IC of a term with only one item annotated is `-log(1/N)` = log(N)‘ which is the maximal IC value in the DAG.

1. *Nmax*

`IC_max` is the maximal IC of all terms. If there is a term with only one item annotated, `Nmax` is identical to the ‘Nunif* method.

```
sim = IC(c)/IC_max
```

1. *Nunivers*

The IC is normalized by the maximal IC of term `a` and `b`.

```
sim = IC(c)/max(IC(a), IC(b))
```

Paper link: doi:10.1613/jair.514, doi:10.1186/147121059S5S4, doi:10.1186/1471210511562, doi:10.1155/2013/292063.
The normalization method can be set with the `norm_method` parameter:

```
term_sim(dag, terms, control = list(norm_method = "Nmax"))
```

Possible values for the `norm_method` parameter are "Nunif", "Nmax", "Nunivers" and "none".

Sim_FaITH_2010:
It is calculated as:

```
sim = IC(c)/(IC(a) + IC(b) - IC(c))
```

The relation between *FaITH_2010* similarity and *Lin_1998* similarity is:

```
sim_FaITH = sim_Lin/(2 - sim_Lin)
```

Paper link: doi:10.1007/9783642177460_39.

Sim_Relevance_2006:
The IC method is fixed to `IC_annotation`.
If thinking *Lin_1998* is a measure of how close term `a` and `b` to their MICA term `c`, the relevance method corrects it by multiplying a factor which considers the specificity of how `c` brings the information. The factor is calculated as `1-p(c)` where `p(c)` is the annotation-based probability `p(c)` = `k/N` where `k` is the number of items annotated to `c` and `N` is the total number of items annotated to the DAG. Then the Relevance semantic similarity is calculated as:

```
sim = (1 - p(c)) * IC_Lin
    = (1 - p(c)) * 2*IC(c)/(IC(a) + IC(b))
```

Paper link: doi:10.1186/147121057302.

Sim_SimIC_2010:

The IC method is fixed to `IC_annotation`.

The SimIC method is an improved correction method of the Relevance method because the latter works bad when `p(c)` is very small. The SimIC correction factor for MICA term `c` is:

```
1 - 1/(1 + IC(c))
```

Then the similarity is:

```
sim = (1 - 1/(1 + IC(c))) * IC_Lin
    = (1 - 1/(1 + IC(c))) * 2*IC(c)/(IC(a) + IC(b))
```

Paper link: doi:10.48550/arXiv.1001.0958.

Sim_XGraSM_2013:

The IC method is fixed to `IC_annotation`.

Being different from the "Relevance" and "SimIC_2010" methods that only use the IC of the MICA term, the *XGraSM_2013* uses IC of all common ancestor terms of `a` and `b`. First it calculates the mean IC of all common ancestor terms with positive IC values:

```
IC_mean = mean_t(IC(t)) where t is an ancestor of both a and b, and IC(t) > 0
```

then similar to the *Lin_1998* method, normalize to the average IC of `a` and `b`:

```
sim = IC_mean*2/(IC(a) + IC(b))
```

Paper link: doi:10.1186/1471210514284.

Sim_EISI_2015:

The IC method is fixed to `IC_annotation`.

It also selects a subset of common ancestors of terms `a` and `b`. It only selects common ancestors which can reach `a` or `b` via one of its child terms that does not belong to the common ancestors. In other words, from the common ancestor, there exist a path where the information is uniquely transmitted to `a` or `b`, not passing the other.

Then the mean IC of the subset common ancestors is calculated and normalized by the *Lin_1998* method.

Paper link: doi:10.1016/j.gene.2014.12.062.

Sim_AIC_2014:

It uses the aggregate information content from ancestors. First define the semantic weight (`Sw`) of a term `t` in the DAG:

```
Sw = 1/(1 + exp(-1/IC(t)))
```

Then calculate the aggregation only in the common ancestors and the aggregationn in the ancestors of the two terms `a` and `b` separatedly:

```
SV_{common ancestors} = sum_{t in common ancestors}(Sw(t))
SV_a = sum{a' in a's ancestors}(Sw(a'))
SV_b = sum{b' in b's ancestors}(Sw(b'))
```

The similarity is calculated as the ratio between the aggregation on the common ancestors and the average on a's ancestors and b's ancestors separatedly.

```
sim = 2*SV_{common_ancestors}/(SV_a + SV_b)
```

Paper link: doi:10.1109/tcbb.2013.176.

Sim_Zhang_2006:

It uses the *IC_Zhang_2006* IC method and the *Lin_1998* method to calculate similarities:

```
sim = 2*IC_zhang(c)/(IC_zhang(a) + IC_zhang(b))
```

Sim_universal:

It uses the *IC_universal* IC method and the *Nunivers* method to calculate similarities:

```
sim = IC_universal(c)/max(IC_universal(a), IC_universal(b))
```

Sim_Wang_2007:

First, S-value of an ancestor term `c` on a term `a` (`S(c->a)`) is calculated (the definition of the S-value can be found in the help page of `term_IC()`). Similar to the *Sim_AIC_2014*, aggregation only to common ancestors, to a's ancestors and to b's ancestors are calculated.

```
SV_{common ancestors} = sum_{c in common ancestors}(S(c->a) + S(c->b))
SV_a = sum{a' in a's ancestors}(S(a'->a))
SV_b = sum{b' in b's ancestors}(S(b'->b))
```

Then the similarity is calculated as:

```
sim = SV_{common_ancestors}*2/(SV_a + SV_b)
```

Paper link: doi:10.1093/bioinformatics/btm087.

The contribution of different semantic relations can be set with the `contribution_factor` parameter. The value should be a named numeric vector where names should cover the relations defined in `relations` set in `create_ontology_DAG()`. For example, if there are two relations "relation_a" and "relation_b" set in the DAG, the value for `contribution_factor` can be set as:

```
term_sim(dag, terms, method = "Sim_Wang_2007",
   control = list(contribution_factor = c("relation_a" = 0.8, "relation_b" = 0.6)))
```

Sim_GOGO_2018:

It is very similar as *Sim_Wang_2007*, but with a corrected contribution factor when calculating the S-value. From a parent term to a child term, *Sim_Wang_2007* directly uses a weight for the relation between the parent and the child, e.g. 0.8 for "is_a" relation type and 0.6 for "part_of" relation type. In *Sim_GOGO_2018*, the weight is also scaled by the total number of children of that parent:

```
w = 1/(c + nc) + w_0
```

where w_0 is the original contribution factor, nc is the number of child terms of the parent, c is calculated to ensure that maximal value of w is no larger than 1, i.e. `c = max(w_0)/(1 - max(w_0))`, assuming minimal value of nc is 1. By default *Sim_GOGO_2018* sets contribution factor of 0.4 for "is_a" and 0.3 for "part_of", then `w = 1/(2/3 + nc) + w_0`.

Paper link: doi:10.1038/s4159801833219y.

The contribution of different semantic relations can be set with the `contribution_factor` parameter. The value should be a named numeric vector where names should cover the relations defined in `relations` set in `create_ontology_DAG()`. For example, if there are two relations "relation_a" and "relation_b" set in the DAG, the value for `contribution_factor` can be set as:

```
term_sim(dag, terms, method = "Sim_GOGO_2018",
   control = list(contribution_factor = c("relation_a" = 0.4, "relation_b" = 0.3)))
```

Sim_Rada_1989:

It is based on the distance between term a and b. It is defined as:

```
sim = 1/(1 + d(a, b))
```

The distance can be the shortest distance between a and b or the longest distance via the LCA term.

Paper link: doi:10.1109/21.24528.

There is a parameter `distance` which takes value of "longest_distances_via_LCA" (the default) or "shortest_distances_via_NCA":

```
term_sim(dag, terms, method = "Sim_Rada_1989",
    control = list(distance = "shortest_distances_via_NCA"))
```

Sim_Resnik_edge_2005:

It is also based on the distance between term a and b:

```
sim = 1 - d(a, b)/2/max_depth
```

where `max_depth` is the maximal depth (maximal distance from root) in the DAG. Similarly, `d(a, b)` can be the shortest distance or the longest distance via LCA.

Paper link: doi:10.1145/1097047.1097051.

There is a parameter `distance` which takes value of "longest_distances_via_LCA" (the default) or "shortest_distances_via_NCA":

```
term_sim(dag, terms, method = "Sim_Resnik_edge_2005",
    control = list(distance = "shortest_distances_via_NCA"))
```

Sim_Leocock_1998:

It is similar as the *Sim_Resnik_edge_2005* method, but it applies log-transformation on the distance and the depth:

```
sim = 1 - log(d(a, b) + 1)/log(2*max_depth + 1)
```

Paper link: doi:10.1186/1471210513261.

There is a parameter `distance` which takes value of "longest_distances_via_LCA" (the default) or "shortest_distances_via_NCA":

```
term_sim(dag, terms, method = "Sim_Leocock_1998",
    control = list(distance = "shortest_distances_via_NCA"))
```

Sim_WP_1994:

It is based on the depth of the LCA term c and the longest distance between term a and b:

```
sim = 2*depth(c)/(len_c(a, b) + 2*depth(c))
```

where len_c(a, b) is the longest distance between a and b via LCA c. The denominator in the equation can also be written as:

```
len_c(a, b) + 2*depth(c) = depth(c) + len(c, a) + depth(c) + len(c, b)
                         = depth_c(a) + depth_c(b)
```

where depth_c(a) is the longest distance from root to a passing through c.

Paper link: doi:10.3115/981732.981751.

Sim_Slimani_2006:

It is a correction of the *Sim_WP_1994* method. The correction factor for term a and b regarding to their LCA t is:

```
CF(a, b) = (1-lambda)*(min(depth(a), depth(b)) - depth(c)) +
           lambda/(1 + abs(depth(a) - depth(b)))
```

lambda takes value of 1 if a and b are in ancestor-offspring relation, or else it takes 0.

Paper link: https://zenodo.org/record/1075130.

Sim_Shenoy_2012:

It is a correction of the *Sim_WP_1994* method. The correction factor for term a and b is:

```
CF(a, b) = exp(-lambda*d(a, b)/max_depth)
```

lambda takes value of 1 if a and b are in ancestor-offspring relation, or else it takes 0. ‘d(a, b)

Paper link: doi:10.48550/arXiv.1211.4709.

There is a parameter distance which takes value of "longest_distances_via_LCA" (the default) or "shortest_distances_via_NCA":

```
term_sim(dag, terms, method = "Sim_Leocock_1998",
    control = list(distance = "shortest_distances_via_NCA"))
```

Sim_Pekar_2002:

It is very similar to the *Sim_WP_1994* method:

```
sim = depth(c)/(len_c(a, b) + depth(c))
    = d(root, c)/(d(c, a) + d(c, b) + d(root, c))
```

where d(a, b) is the longest distance between a and b.

Paper link: https://aclanthology.org/C02-1090/.

Sim_Stojanovic_2001:

It is purely based on the depth of term a, b and their LCA c.

```
sim = depth(c)/(depth(a) + depth(b) - depth(c))
```

The similarity value might be negative because there is no restrction that the path from root to a or b must pass c.
Paper link: doi:10.1145/500737.500762.

Sim_Wang_edge_2012:
It is calculated as:

```
sim = depth(c)^2/depth_c(a)/depth_c(b)
```

where depth_c(a) is the longest distance between root to a passing through c.
Paper link: doi:10.1186/1477595610s1s18.

Sim_Zhong_2002:
For a term x, it first calculates a "mile-stone" value as

```
m(x) = 0.5/2^depth(x)
```

The the distance bewteen term a and b via LCA term c is:

```
D(c, a) + D(c, b) = m(c) - m(a) + m(c) - m(b)
                  = 2*m(c) - m(a) - m(b)
                  = 1/2^depth(c) - 0.5/2^depth(a) - 0.5/2^depth(b)
```

We change the original depth(a) to let it go through LCA term c when calculating the depth:

```
1/2^depth(c) - 0.5/2^depth(a) - 0.5/2^depth(b)
   = 1/2^depth(c)- 0.5/2^(depth(c) + len(c, a)) - 0.5/2^(depth(c) + len(c, b))
   = 1/2^depth(c) * (1 - 1/2^(len(c, a) + 1) - 1/2^(len(c, b) + 1))
   = 2^-depth(c) * (1 - 2^-(len(c, a) + 1) - 2^-(len(c, b) + 1))
```

And the final similarity is 1 - distance:

```
sim = 1 - 2^-depth(c) * (1 - 2^-(len(c, a) + 1) - 2^-(len(c, b) + 1))
```

Paper link: doi:10.1007/3540454837_8.
There is a parameter depth_via_LCA that can be set to TRUE or FALSE. IF it is set to TRUE, depth(a) is re-defined as should pass the LCA term c. If it is FALSE, it goes to the original similarity definition in the paper and note the similarity might be negative.

```
term_sim(dag, terms, method = "Sim_Zhong_2002",
    control = list(depth_via_LCA = FALSE))
```

Sim_AlMubaid_2006:
It also takes accout of the distance between term a and b, and the depth of the LCA term c in the DAG. The distance is calculated as:

```
D(a, b) = log(1 + d(a, b)*(max_depth - depth(c)))
```

Here d(a, b) can be the shortest distance between a and b or the longst distance via LCA c.
Then the distance is transformed into the similarity value scaled by the possible maximal and minimal values of D(a, b) from the DAG:

```
D_max = log(1 + 2*max_depth * max_depth)
```

And the minimal value of `D(a, b)` is zero when `a` is identical to `b`. Then the similarity value is scaled as:

```
sim = 1 - D(a, b)/D_max
```

Paper link: doi:10.1109/IEMBS.2006.259235.

There is a parameter `distance` which takes value of "longest_distances_via_LCA" (the default) or "shortest_distances_via_NCA":

```
term_sim(dag, terms, method = "Sim_AlMubaid_2006",
    control = list(distance = "shortest_distances_via_NCA"))
```

Sim_Li_2003:

It is similar to the *Sim_AlMubaid_2006* method, but uses a non-linear form:

```
sim = exp(0.2*d(a, b)) * atan(0.6*depth(c))
```

where `d(a, b)` can be the shortest distance or the longest distance via LCA.

Paper link: doi:10.1109/TKDE.2003.1209005.

There is a parameter `distance` which takes value of "longest_distances_via_LCA" (the default) or "shortest_distances_via_NCA":

```
term_sim(dag, terms, method = "Sim_Li_2003",
    control = list(distance = "shortest_distances_via_NCA"))
```

Sim_RSS_2013:

The similarity is adjusted by the positions of term `a`, `b` and the LCA term `c` in the DAG. The similarity is defined as:

```
sim = max_depth/(max_depth + d(a, b)) * alpha/(alpha + beta)
```

where `d(a, b)` is the distance between `a` and `b` which can be the shortest distance or the longest distance via LCA.

In the tuning factor, `alpha` is the distance of LCA to root, which is `depth(c)`. `beta` is the distance to leaves, which is the minimal distance (or the minimal height) of term `a` and `b`:

```
alpha/(alpha + beta) = depth(c)/(depth(c) + min(height(a), height(b)))
```

Paper link: doi:10.1371/journal.pone.0066745.

There is a parameter `distance` which takes value of "longest_distances_via_LCA" (the default) or "shortest_distances_via_NCA":

```
term_sim(dag, terms, method = "Sim_RSS_2013",
    control = list(distance = "shortest_distances_via_NCA"))
```

Sim_HRSS_2013:

It is similar as the *Sim_RSS_2013* method, but it uses information content instead of the distance to adjust the similarity.

It first defines the semantic distance between term `a` and `b` as the sum of the distance to their MICA term `c`:

```
D(a, b) = D(c, a) + D(c, b)
```

And the distance between an ancestor to a term is:

```
D(c, a) = IC(a) - IC(c)  # if c is an ancestor of a
D(a, b) = D(c, a) + D(c, b) = IC(a) + IC(b) - 2*IC(c) # if c is the MICA of a and b
```

Similarly, the similarity is also corrected by the position of MICA term and a and b in the DAG:

```
1/(1 + D(a, b)) * alpha/(alph + beta)
```

Now alpha is the IC of the MICA term:

```
alpha = IC(c)
```

And beta is the average of the maximal semantic distance of a and b to leaves.

```
beta = 0.5*(IC(l_a) - IC(a) + IC(l_b) - IC(b))
```

where l_a is the leaf that a can reach with the highest IC (i.e. most informative leaf), and so is l_b.
Paper link: doi:10.1371/journal.pone.0066745.

Sim_Shen_2010:
It is based on the information content of terms on the path connecting term a and b via their MICA term c.
Denote a list of terms a, ..., c, ..., b which are composed by the shortest path from a to c and from b to c, the difference between a and b is the sum of 1/IC of the terms on the path:

```
sum_{x in the path}(1/IC(x))
```

Then the distance is scaled into [0, 1] by an arctangent tarnsformation:

```
atan(sum_{x in the path}(1/IC(x)))/(pi/2)
```

And finally the similarity is:

```
sim = 1 - atan(sum_{x in the path}(1/IC(x)))/(pi/2)
```

Paper link: doi:10.1109/BIBM.2010.5706623.

Sim_SSDD_2013:
It is similar as the *Sim_Shen_2010* which also sums content along the path passing through LCA term. Instead of summing the information content, the *Sim_SSDD_2013* sums up a so-called "T-value":

```
sim = 1 - atan(sum_{x in the path}(T(x)))/(pi/2)
```

Each term has a T-value and it measures the semantic content a term averagely inherited from its parents and distributed to its offsprings. The T-value of root is 1. Assume a term t has two parents p1 and p1, The T-value for term t is averaged from its

```
(w1*T(p1) + w2*T(p2))/2
```

Since the parent may have other child terms, a factor `w1` or `w2` is multiplied to `T(p1)` and `T(p2)`. Taking `p1` as an example, it has `n_p` offsprings (including itself) and `t` has `n_t` offsprings (including itself), this means `n_t/n_p` of information is transmitted from `p1` to downstream via `t`, thus `w1` is defined as `n_t/n_p`.

Paper link: doi:10.1016/j.ygeno.2013.04.010.

Sim_Jiang_1997:

First semantic distance between term `a` and `b` via MICA term `c` is defined as:

```
D(a, b) = IC(a) + IC(b) - 2*IC(c)
```

Then there are several normalization method to change the distance to similarity and to scale it into the range of `[0, 1]`.

- max: `1 - D(a, b)/2/IC_max`
- Couto: `min(1, D(a, b)/IC_max)`
- Lin: `1 - D(a, b)/(IC(a) + IC(b))` which is the same as the *Sim_Lin_1998* method
- Garla: `1 - log(D(a, b) + 1)/log(2*IC_max + 1)`
- log-Lin: `1 - log(D(a, b) + 1)/log(IC(a) + IC(b) + 1)`
- Rada: `1/(1 + D(a, b))`

Paper link: https://aclanthology.org/O97-1002/.

There is a parameter `norm_method` which takes value in "max", "Couto", "Lin", "Carla", "log-Lin", "Rada":

```
term_sim(dag, terms, method = "Sim_Jiang_1997",
    control = list(norm_method = "Lin"))
```

Sim_Kappa:

Denote two sets `A` and `B` as the items annotated to term `a` and `b`. The similarity value is the kappa coeffcient of the two sets.

The universe or the background can be set via parameter `anno_universe`:

```
term_sim(dag, terms, method = "Sim_kappa",
    control = list(anno_universe = ...))
```

Sim_Jaccard:

Denote two sets `A` and `B` as the items annotated to term `a` and `b`. The similarity value is the Jaccard coeffcient of the two sets, defined as `length(intersect(A, B))/length(union(A, B))`.

The universe or the background can be set via parameter `anno_universe`:

```
term_sim(dag, terms, method = "Sim_Jaccard",
    control = list(anno_universe = ...))
```

Sim_Dice:

Denote two sets `A` and `B` as the items annotated to term `a` and `b`. The similarity value is the Dice coeffcient of the two sets, defined as `2*length(intersect(A, B))/(length(A) + length(B))`.

The universe or the background can be set via parameter `anno_universe`:

```
term_sim(dag, terms, method = "Sim_Dice",
    control = list(anno_universe = ...))
```

Sim_Overlap:

Denote two sets `A` and `B` as the items annotated to term `a` and `b`. The similarity value is the overlap coeffcient of the two sets, defined as `length(intersect(A, B))/min(length(A), length(B))`. The universe or the background can be set via parameter `anno_universe`:

```
term_sim(dag, terms, method = "Sim_Overlap",
    control = list(anno_universe = ...))
```

Sim_Ancestor:

Denote `S_a` and `S_b` are two sets of ancestor terms of term `a` and `b` (including `a` and `b`), the semantic similarity is defined as:

```
length(intersect(S_a, S_b))/length(union(S_a, S_b))
```

```
term_sim(dag, terms, method = "Sim_Ancestor")
```

Examples

```
parents  = c("a", "a", "b", "b", "c", "d")
children = c("b", "c", "c", "d", "e", "f")
annotation = list(
    "a" = 1:3,
    "b" = 3:4,
    "c" = 5,
    "d" = 7,
    "e" = 4:7,
    "f" = 8
)
dag = create_ontology_DAG(parents, children, annotation = annotation)
term_sim(dag, dag_all_terms(dag), method = "Sim_Lin_1998")
```

[,ontology_DAG,ANY,ANY,missing-method

Create sub-DAGs

Description

Create sub-DAGs

Usage

```
## S4 method for signature 'ontology_DAG,ANY,ANY,missing'
x[i, j, ..., drop = FALSE]

## S4 method for signature 'ontology_DAG,ANY,ANY,ANY'
x[i, j, ..., drop = FALSE]

## S4 method for signature 'ontology_DAG,ANY,missing,missing'
x[i, j, ..., drop = FALSE]
```

```
## S4 method for signature 'ontology_DAG,ANY,missing,ANY'
x[i, j, ..., drop = FALSE]

## S4 method for signature 'ontology_DAG,missing,ANY,missing'
x[i, j, ..., drop = FALSE]

## S4 method for signature 'ontology_DAG,missing,ANY,ANY'
x[i, j, ..., drop = FALSE]

## S4 method for signature 'ontology_DAG,missing,missing,missing'
x[i, j, ..., drop = FALSE]

## S4 method for signature 'ontology_DAG,missing,missing,ANY'
x[i, j, ..., drop = FALSE]

## S4 method for signature 'ontology_DAG,character,missing'
x[[i, j, ...]]
```

Arguments

x	An ontology_DAG object.
i	A single term name. The value should be a character vector. It corresponds to the roots.
j	A single term name. The value should be a character vector. It corresponds to the leaves.
...	Ignored.
drop	Ignored.

Details

It returns a sub-DAG taking node i as the root and j as the leaves. If i is a vector, a super root will be added.

Value

An ontology_DAG object.

Examples

```
parents  = c("a", "a", "b", "b", "c", "d")
children = c("b", "c", "c", "d", "e", "f")
dag = create_ontology_DAG(parents, children)
dag["b"]
dag[["b"]]
dag["b", "f"]
dag[, "f"]
```

Index

[,ontology_DAG,ANY,ANY,ANY-method (`[,ontology_DAG,ANY,ANY,missing-method`), 66
[,ontology_DAG,ANY,ANY,missing-method, 66
[,ontology_DAG,ANY,missing,ANY-method (`[,ontology_DAG,ANY,ANY,missing-method`), 66
[,ontology_DAG,ANY,missing,missing-method (`[,ontology_DAG,ANY,ANY,missing-method`), 66
[,ontology_DAG,missing,ANY,ANY-method (`[,ontology_DAG,ANY,ANY,missing-method`), 66
[,ontology_DAG,missing,ANY,missing-method (`[,ontology_DAG,ANY,ANY,missing-method`), 66
[,ontology_DAG,missing,missing,ANY-method (`[,ontology_DAG,ANY,ANY,missing-method`), 66
[,ontology_DAG,missing,missing,missing-method (`[,ontology_DAG,ANY,ANY,missing-method`), 66
[[,ontology_DAG,character,missing-method (`[,ontology_DAG,ANY,ANY,missing-method`), 66

add_annotation, 3
all_group_sim_methods (`all_term_IC_methods`), 3
all_group_sim_methods(), *27*
all_term_IC_methods, 3
all_term_IC_methods(), *38*, *52*
all_term_sim_methods (`all_term_IC_methods`), 3
all_term_sim_methods(), *56*
alternative_GO_terms (`create_ontology_DAG_from_GO_db`), 6
annotated_terms (`term_annotations`), 51
annotated_terms(), *27*
avg_children (`n_offspring`), 41
avg_parents (`n_offspring`), 41

CA_terms (`MICA_term`), 37
ComplexHeatmap::Legend(), *11*
create_ontology_DAG, 4
create_ontology_DAG(), *9*, *19*, *34*, *45*, *55*, *59*, *60*
create_ontology_DAG_from_GO_db, 6
create_ontology_DAG_from_GO_db(), *45*
create_ontology_DAG_from_igraph, 7

dag_add_random_children (`dag_random_tree`), 22
dag_all_terms, 8
dag_all_terms(), *11*, *24*
dag_ancestors (`dag_parents`), 21
dag_as_dendrogram (`dag_treelize`), 26
dag_as_DOT (`dag_circular_viz`), 9
dag_as_igraph, 9
dag_children (`dag_parents`), 21
dag_circular_viz, 9
dag_depth, 12
dag_distinct_ancestors, 13
dag_enrich_on_genes (`dag_enrich_on_items`), 14
dag_enrich_on_items, 14
dag_enrich_on_offsprings, 15
dag_enrich_on_offsprings(), *17*
dag_enrich_on_offsprings_by_permutation, 17
dag_filter, 18
dag_graphviz (`dag_circular_viz`), 9
dag_has_terms, 19
dag_height (`dag_depth`), 12
dag_is_leaf (`dag_root`), 24
dag_leaves (`dag_root`), 24
dag_leaves(), *24*

dag_longest_dist_from_ancestors (dag_longest_dist_to_offspring), 20
dag_longest_dist_to_offspring, 20
dag_n_leaves (dag_all_terms), 8
dag_n_relations (dag_all_terms), 8
dag_n_terms (dag_all_terms), 8
dag_offspring (dag_parents), 21
dag_parents, 21
dag_permutate_children (dag_reorder), 23
dag_random (dag_random_tree), 22
dag_random_tree, 22
dag_reorder, 23
dag_root, 24
dag_shiny, 25
dag_shortest_dist_from_ancestors (dag_longest_dist_to_offspring), 20
dag_shortest_dist_from_root (dag_depth), 12
dag_shortest_dist_to_leaves (dag_depth), 12
dag_shortest_dist_to_offspring (dag_longest_dist_to_offspring), 20
dag_siblings (dag_parents), 21
dag_treelize, 26
DiagrammeR::grViz(), *11*

grid::grid.newpage(), *11*
group_sim, 27

has_annotation (n_annotations), 40

igraph::igraph, *7*, *9*
import_obo, 33
import_ontology (import_obo), 33
import_owl (import_obo), 33
import_ttl (import_obo), 33

LCA_depth (MICA_term), 37
LCA_term (MICA_term), 37
longest_distances_directed (shortest_distances_via_NCA), 48
longest_distances_via_LCA (shortest_distances_via_NCA), 48

max_ancestor_id (MICA_term), 37
max_ancestor_path_sum (MICA_term), 37
max_ancestor_v (MICA_term), 37
mcols,ontology_DAG-method, 36
mcols<-,ontology_DAG-method (mcols,ontology_DAG-method), 36
method_param, 37
MICA_IC (MICA_term), 37
MICA_term, 37

n_ancestors (n_offspring), 41
n_annotations, 40
n_annotations(), *53*
n_children (n_offspring), 41
n_connected_leaves (n_offspring), 41
n_offspring, 41
n_parents (n_offspring), 41
NCA_term (MICA_term), 37

ontology_chebi (ontology_kw), 43
ontology_DAG (ontology_DAG-class), 42
ontology_DAG-class, 42
ontology_go (ontology_kw), 43
ontology_hp (ontology_kw), 43
ontology_kw, 43
ontology_pw (ontology_kw), 43
ontology_rdo (ontology_kw), 43
ontology_reactome (ontology_kw), 43
ontology_vt (ontology_kw), 43
org.Hs.eg.db::org.Hs.eg.db, *7*

partition_by_level, 45
partition_by_level(), *10*
partition_by_size (partition_by_level), 45
partition_by_size(), *10*
print.ontology_tree (dag_treelize), 26
print.print_source, 46

random_items (random_terms), 47
random_terms, 47

shortest_distances_directed (shortest_distances_via_NCA), 48
shortest_distances_via_NCA, 48
shortest_distances_via_NCA(), *39*
show,ontology_DAG-method, 49
simona_opt, 49

term_annotations, 51

term_IC, 52
term_IC(), *59*
term_sim, 56

UniProtKeywords::load_keyword_genesets(), *45*