PyBEL-Tools Documentation

Release 0.7.3-dev

Charles Tapley Hoyt

Apr 16, 2019

GETTING STARTED

PyBEL-Tools is a suite of tools built on top of PyBEL to facilitate data management, integration, and analysis. For further examples, see the PyBEL-Notebooks repository.

CHAPTER

ONE

CITATION

If you use PyBEL and PyBEL Tools in your work, please cite [Hoyt2017]:

CHAPTER

TWO

LINKS

- Documented on Read the Docs
- Versioned on GitHub
- Tested on Travis CI
- Distributed by PyPI
- Chat on Gitter

2.1 Installation

A cool pool of tools for PyBEL.

2.1.1 Installation

Easiest

Download the latest stable code from PyPI with:

```
$ pip install pybel_tools
```

Get the Latest

Download the most recent code from GitHub with:

```
$ pip install git+https://github.com/pybel/pybel-tools.git
```

For Developers

Clone the repository from GitHub and install in editable mode with:

```
$ git clone https://github.com/pybel/pybel-tools.git
$ cd pybel-tools
$ pip install -e .
```

Caveats

PyBEL Tools contains many dependencies, including the scientific Python Stack (numpy, scipy, etc.). This makes installation difficult for Windows users, for whom Python cannot easily build C extensions. We recommend using an Anaconda distribution of Python, which includes these precompiled.

2.1.2 Testing

PyBEL-Tools is tested on Python3 on Linux on Travis CI.

2.2 Command Line Interface

2.2.1 pybel-tools

PyBEL-Tools v0.7.3-dev Command Line Interface on /home/docs/checkouts/readthedocs.org/user_builds/pybel-tools/envs/latest/bin/python with PyBEL v0.13.1

```
pybel-tools [OPTIONS] COMMAND [ARGS]...
```

Options

--version
Show the version and exit.

annotation

Annotation file utilities.

```
pybel-tools annotation [OPTIONS] COMMAND [ARGS]...
```

convert-to-namespace

Convert an annotation file to a namespace file.

```
pybel-tools annotation convert-to-namespace [OPTIONS]
```

Options

-f, --file <file>
Path to input BEL Namespace file

-o, --output <output>
Path to output converted BEL Namespace file

--keyword <keyword>
Set custom keyword. useful if the annotation keyword is too long

document

BEL document utilities.

```
pybel-tools document [OPTIONS] COMMAND [ARGS]...
```

boilerplate

Build a template BEL document with the given PubMed identifiers.

```
pybel-tools document boilerplate [OPTIONS] NAME CONTACT DESCRIPTION [PMIDS]...
```

Options

--version <version>

--copyright <copyright>

--authors <authors>

--licenses <licenses>

--disclaimer <disclaimer>

--output <output>

Arguments

NAME
Required argument

CONTACT
Required argument

DESCRIPTION
Required argument

PMIDS
Optional argument(s)

serialize-namespaces

Parse a BEL document then serializes the given namespaces (errors and all) to the given directory.

```
pybel-tools document serialize-namespaces [OPTIONS] [NAMESPACES]...
```

Options

-c, --connection <connection>
Database connection string. [default: sqlite:////home/docs/.pybel/pybel_0.13.0_cache.db]

-p, --path <path>
Input BEL file path. Defaults to stdin.

-d, --directory <directory>
Output folder. Defaults to current working directory /home/docs/checkouts/readthedocs.org/user_builds/pybel-tools/checkouts/latest/docs/source)

Arguments

NAMESPACES
Optional argument(s)

io

Upload and conversion utilities.

```
pybel-tools io [OPTIONS] COMMAND [ARGS]...
```

Options

-c, --connection <connection>
Database connection string. [default: sqlite:////home/docs/.pybel/pybel_0.13.0_cache.db]

get-pmids

Output PubMed identifiers from a graph to a stream.

```
pybel-tools io get-pmids [OPTIONS] path
```

Options

-o, --output <output>

Arguments

path
Required argument

namespace

Namespace file utilities.

```
pybel-tools namespace [OPTIONS] COMMAND [ARGS]...
```

convert-to-annotation

Convert a namespace file to an annotation file.

```
pybel-tools namespace convert-to-annotation [OPTIONS]
```

Options

-f, --file <file>
Path to input BEL Namespace file

-o, --output <output>
Path to output converted BEL Annotation file

write

Build a namespace from items.

```
pybel-tools namespace write [OPTIONS] NAME KEYWORD DOMAIN CITATION
```

Options

--author <author>

--description <description>

--species <species>

--version <version>

--contact <contact>

--license <license>

--values <values>
A file containing the list of names

--output <output>

Arguments

NAME
Required argument

KEYWORD
Required argument

DOMAIN
Required argument

CITATION
Required argument

2.3 Summary

These scripts are designed to assist in the analysis of errors within BEL documents and provide some suggestions for fixes.

pybel_tools.summary.**count_relations**(*graph*)
Return a histogram over all relationships in a graph.

Parameters **graph** (*pybel.BELGraph*) – A BEL graph

Returns A Counter from {relation type: frequency}

Return type collections.Counter

pybel_tools.summary.**get_edge_relations**(*graph*)
Build a dictionary of {node pair: set of edge types}.

Return type Mapping[Tuple[BaseEntity, BaseEntity], Set[str]]

pybel_tools.summary.**count_unique_relations**(*graph*)
Return a histogram of the different types of relations present in a graph.

Note: this operation only counts each type of edge once for each pair of nodes

Return type Counter

pybel_tools.summary.**count_annotations**(*graph*)
Count how many times each annotation is used in the graph.

Parameters **graph** (*pybel.BELGraph*) – A BEL graph

Returns A Counter from {annotation key: frequency}

Return type collections.Counter

pybel_tools.summary.**get_annotations**(*graph*)
Get the set of annotations used in the graph.

Parameters **graph** (*pybel.BELGraph*) – A BEL graph

Returns A set of annotation keys

Return type set[str]

pybel_tools.summary.**get_annotations_containing_keyword**(*graph*, *keyword*)
Get annotation/value pairs for values for whom the search string is a substring

Parameters

- **graph** (BELGraph) – A BEL graph
- **keyword** (str) – Search for annotations whose values have this as a substring

Return type List[Mapping[str, str]]

pybel_tools.summary.**count_annotation_values**(*graph*, *annotation*)
Count in how many edges each annotation appears in a graph

Parameters

- **graph** (BELGraph) – A BEL graph
- **annotation** (str) – The annotation to count

Return type Counter

Returns A Counter from {annotation value: frequency}

pybel_tools.summary.**count_annotation_values_filtered**(*graph*, *annotation*, *source_predicate=None*, *target_predicate=None*)
Count in how many edges each annotation appears in a graph, but filter out source nodes and target nodes.

See pybel_tools.utils.keep_node() for a basic filter.

Parameters

- **graph** (BELGraph) – A BEL graph

- **annotation** (str) – The annotation to count
- **source_predicate** (Optional[Callable[[BELGraph, BaseEntity], bool]]) – A predicate (graph, node) -> bool for keeping source nodes
- **target_predicate** (Optional[Callable[[BELGraph, BaseEntity], bool]]) – A predicate (graph, node) -> bool for keeping target nodes

Return type Counter

Returns A Counter from {annotation value: frequency}

pybel_tools.summary.**pair_is_consistent**(*graph*, *u*, *v*)
Return if the edges between the given nodes are consistent, meaning they all have the same relation.

Return type Optional[str]

Returns If the edges aren't consistent, return false, otherwise return the relation type

pybel_tools.summary.**get_consistent_edges**(*graph*)
Yield pairs of (source node, target node) for which all of their edges have the same type of relation.

Return type Iterable[Tuple[BaseEntity, BaseEntity]]

Returns An iterator over (source, target) node pairs corresponding to edges with many inconsistent relations

pybel_tools.summary.**get_contradictory_pairs**(*graph*)
Iterates over contradictory node pairs in the graph based on their causal relationships

Return type Iterable[Tuple[BaseEntity, BaseEntity]]

Returns An iterator over (source, target) node pairs that have contradictory causal edges

pybel_tools.summary.**count_pathologies**(*graph*)
Count the number of edges in which each pathology is incident.

Parameters **graph** (*pybel.BELGraph*) – A BEL graph

Return type Counter

pybel_tools.summary.**get_unused_annotations**(*graph*)
Get the set of all annotations that are defined in a graph, but are never used.

Parameters **graph** (*pybel.BELGraph*) – A BEL graph

Returns A set of annotations

Return type set[str]

pybel_tools.summary.**get_unused_list_annotation_values**(*graph*)
Get all of the unused values for list annotations.

Parameters **graph** (*pybel.BELGraph*) – A BEL graph

Returns A dictionary of {str annotation: set of str values that aren't used}

Return type dict[str,set[str]]

pybel_tools.summary.**count_error_types**(*graph*)
Count the occurrence of each type of error in a graph.

Return type Counter

Returns A Counter of {error type: frequency}

pybel_tools.summary.**count_naked_names**(*graph*)
Count the frequency of each naked name (names without namespaces).

Return type Counter

Returns A Counter from {name: frequency}

pybel_tools.summary.**get_naked_names**(*graph*)
Get the set of naked names in the graph.

Return type Set[str]

pybel_tools.summary.**get_incorrect_names_by_namespace**(*graph*, *namespace*)
Return the set of all incorrect names from the given namespace in the graph.

Return type Set[str]

Returns The set of all incorrect names from the given namespace in the graph

pybel_tools.summary.**get_incorrect_names**(*graph*)
Return the dict of the sets of all incorrect names from the given namespace in the graph.

Return type Mapping[str, Set[str]]

Returns The set of all incorrect names from the given namespace in the graph

pybel_tools.summary.**get_undefined_namespaces**(*graph*)
Get all namespaces that are used in the BEL graph aren't actually defined.

Return type Set[str]

pybel_tools.summary.**get_undefined_namespace_names**(*graph*, *namespace*)
Get the names from a namespace that wasn't actually defined.

Return type Set[str]

Returns The set of all names from the undefined namespace

pybel_tools.summary.**calculate_incorrect_name_dict**(*graph*)
Group all of the incorrect identifiers in a dict of {namespace: list of erroneous names}.

Return type Mapping[str, str]

Returns A dictionary of {namespace: list of erroneous names}

pybel_tools.summary.**calculate_error_by_annotation**(*graph*, *annotation*)
Group the graph by a given annotation and builds lists of errors for each.

Return type Mapping[str, List[str]]

Returns A dictionary of {annotation value: list of errors}

pybel_tools.summary.**group_errors**(*graph*)
Group the errors together for analysis of the most frequent error.

Return type Mapping[str, List[int]]

Returns A dictionary of {error string: list of line numbers}

pybel_tools.summary.**get_names_including_errors**(*graph*)
Takes the names from the graph in a given namespace and the erroneous names from the same namespace and returns them together as a unioned set

Return type Mapping[str, Set[str]]

Returns The dict of the sets of all correct and incorrect names from the given namespace in the graph

pybel_tools.summary.**get_names_including_errors_by_namespace**(*graph*, *namespace*)
Takes the names from the graph in a given namespace (pybel.struct.summary.get_names_by_namespace()) and the erroneous names from the same namespace (*get_incorrect_names_by_namespace()*) and returns them together as a unioned set

Return type Set[str]

Returns The set of all correct and incorrect names from the given namespace in the graph

pybel_tools.summary.**get_undefined_annotations**(*graph*)
Get all annotations that aren't actually defined.

Return type Set[str]

Returns The set of all undefined annotations

pybel_tools.summary.**get_namespaces_with_incorrect_names**(*graph*)
Return the set of all namespaces with incorrect names in the graph.

Return type Set[str]

pybel_tools.summary.**get_most_common_errors**(*graph*, *n=20*)
Get the (n) most common errors in a graph.

pybel_tools.summary.**plot_summary_axes**(*graph*, *lax*, *rax*, *logx=True*)
Plots your graph summary statistics on the given axes.

After, you should run plt.tight_layout() and you must run plt.show() to view.

Shows: 1. Count of nodes, grouped by function type 2. Count of edges, grouped by relation type

Parameters

- **graph** (*pybel.BELGraph*) – A BEL graph
- **lax** – An axis object from matplotlib
- **rax** – An axis object from matplotlib

Example usage:

```
>>> import matplotlib.pyplot as plt
>>> from pybel import from_pickle
>>> from pybel_tools.summary import plot_summary_axes
>>> graph = from_pickle('~/dev/bms/aetionomy/parkinsons.gpickle')
>>> fig, axes = plt.subplots(1, 2, figsize=(10, 4))
>>> plot_summary_axes(graph, axes[0], axes[1])
>>> plt.tight_layout()
>>> plt.show()
```

pybel_tools.summary.**plot_summary**(*graph*, *plt*, *logx=True*, ***kwargs*)
Plots your graph summary statistics. This function is a thin wrapper around plot_summary_axis(). It automatically takes care of building figures given matplotlib's pyplot module as an argument. After, you need to run plt.show().

plt is given as an argument to avoid needing matplotlib as a dependency for this function

Shows:

1. Count of nodes, grouped by function type
2. Count of edges, grouped by relation type

Parameters

- **plt** – Give `matplotlib.pyplot` to this parameter
- **kwargs** – keyword arguments to give to `plt.subplots()`

Example usage:

```
>>> import matplotlib.pyplot as plt
>>> from pybel import from_pickle
>>> from pybel_tools.summary import plot_summary
>>> graph = from_pickle('~/dev/bms/aetionomy/parkinsons.gpickle')
>>> plot_summary(graph, plt, figsize=(10, 4))
>>> plt.show()
```

pybel_tools.summary.**is_causal_relation**(*edge_data*)
Check if the given relation is causal.

Return type bool

pybel_tools.summary.**get_causal_out_edges**(*graph*, *nbunch*)
Get the out-edges to the given node that are causal.

Return type Set[Tuple[BaseEntity, BaseEntity]]

Returns A set of (source, target) pairs where the source is the given node

pybel_tools.summary.**get_causal_in_edges**(*graph*, *nbunch*)
Get the in-edges to the given node that are causal.

Return type Set[Tuple[BaseEntity, BaseEntity]]

Returns A set of (source, target) pairs where the target is the given node

pybel_tools.summary.**is_causal_source**(*graph*, *node*)
Return true of the node is a causal source.

- Doesn't have any causal in edge(s)
- Does have causal out edge(s)

Return type bool

pybel_tools.summary.**is_causal_central**(*graph*, *node*)
Return true if the node is neither a causal sink nor a causal source.

- Does have causal in edges(s)
- Does have causal out edge(s)

Return type bool

pybel_tools.summary.**is_causal_sink**(*graph*, *node*)
Return true if the node is a causal sink.

- Does have causal in edge(s)
- Doesn't have any causal out edge(s)

Return type bool

pybel_tools.summary.**get_causal_source_nodes**(*graph*, *func*)
Return a set of all nodes that have an in-degree of 0.

This likely means that it is an external perturbagen and is not known to have any causal origin from within the biological system. These nodes are useful to identify because they generally don't provide any mechanistic insight.

Return type Set[BaseEntity]

pybel_tools.summary.**get_causal_central_nodes**(*graph*, *func*)
Return a set of all nodes that have both an in-degree > 0 and out-degree > 0.

This means that they are an integral part of a pathway, since they are both produced and consumed.

Return type Set[BaseEntity]

pybel_tools.summary.**get_causal_sink_nodes**(*graph*, *func*)
Returns a set of all ABUNDANCE nodes that have an causal out-degree of 0.

This likely means that the knowledge assembly is incomplete, or there is a curation error.

Return type Set[BaseEntity]

pybel_tools.summary.**get_degradations**(*graph*)
Get all nodes that are degraded.

Return type Set[BaseEntity]

pybel_tools.summary.**get_activities**(*graph*)
Get all nodes that have molecular activities.

Return type Set[BaseEntity]

pybel_tools.summary.**get_translocated**(*graph*)
Get all nodes that are translocated.

Return type Set[BaseEntity]

pybel_tools.summary.**count_top_centrality**(*graph*, *number=30*)
Get top centrality dictionary.

Return type Mapping[BaseEntity, int]

pybel_tools.summary.**get_modifications_count**(*graph*)
Get a modifications count dictionary.

Return type Mapping[str, int]

pybel_tools.summary.**count_subgraph_sizes**(*graph*, *annotation='Subgraph'*)
Count the number of nodes in each subgraph induced by an annotation.

Parameters **annotation** (str) – The annotation to group by and compare. Defaults to 'Subgraph'

Return type Counter[int]

Returns A dictionary from {annotation value: number of nodes}

pybel_tools.summary.**calculate_subgraph_edge_overlap**(*graph*, *annotation='Subgraph'*)
Build a DatafFame to show the overlap between different sub-graphs.

Options: 1. Total number of edges overlap (intersection) 2. Percentage overlap (tanimoto similarity)

Parameters

- **graph** (BELGraph) – A BEL graph

- **annotation** (str) – The annotation to group by and compare. Defaults to 'Subgraph'

Return type Tuple[Mapping[str, Set[Tuple[BaseEntity, BaseEntity]]], Mapping[str, Mapping[str, Set[Tuple[BaseEntity, BaseEntity]]]], Mapping[str, Mapping[str, Set[Tuple[BaseEntity, BaseEntity]]]], Mapping[str, Mapping[str, float]]]

Returns {subgraph: set of edges}, {(subgraph 1, subgraph2): set of intersecting edges}, {(subgraph 1, subgraph2): set of unioned edges}, {(subgraph 1, subgraph2): tanimoto similarity},

pybel_tools.summary.**summarize_subgraph_edge_overlap**(*graph*, *annotation='Subgraph'*)

Return a similarity matrix between all subgraphs (or other given annotation).

Parameters **annotation** (str) – The annotation to group by and compare. Defaults to "Subgraph"

Returns A similarity matrix in a dict of dicts

Return type dict

pybel_tools.summary.**rank_subgraph_by_node_filter**(*graph*, *node_predicates*, *annotation='Subgraph'*, *reverse=True*)

Rank sub-graphs by which have the most nodes matching an given filter.

A use case for this function would be to identify which subgraphs contain the most differentially expressed genes.

```
>>> from pybel import from_pickle
>>> from pybel.constants import GENE
>>> from pybel_tools.integration import overlay_type_data
>>> from pybel_tools.summary import rank_subgraph_by_node_filter
>>> import pandas as pd
>>> graph = from_pickle('~/dev/bms/aetionomy/alzheimers.gpickle')
>>> df = pd.read_csv('~/dev/bananas/data/alzheimers_dgxp.csv', columns=['Gene',
↪'log2fc'])
>>> data = {gene: log2fc for _, gene, log2fc in df.itertuples()}
>>> overlay_type_data(graph, data, 'log2fc', GENE, 'HGNC', impute=0.0)
>>> results = rank_subgraph_by_node_filter(graph, lambda g, n: 1.3 < abs(g[n][
↪'log2fc']))
```

Return type List[Tuple[str, int]]

pybel_tools.summary.**summarize_subgraph_node_overlap**(*graph*, *node_predicates=None*, *annotation='Subgraph'*)

Calculate the subgraph similarity tanimoto similarity in nodes passing the given filter.

Provides an alternate view on subgraph similarity, from a more node-centric view

pybel_tools.summary.**count_pmids**(*graph*)

Count the frequency of PubMed documents in a graph.

Return type Counter

Returns A Counter from {(pmid, name): frequency}

pybel_tools.summary.**get_pmid_by_keyword**(*keyword*, *graph=None*, *pubmed_identifiers=None*)

Get the set of PubMed identifiers beginning with the given keyword string.

Parameters

- **keyword** (str) – The beginning of a PubMed identifier
- **graph** (Optional[BELGraph]) – A BEL graph

- **pubmed_identifiers** (Optional[Set[str]]) – A set of pre-cached PubMed identifiers

Return type Set[str]

Returns A set of PubMed identifiers starting with the given string

pybel_tools.summary.**count_citations**(*graph*, ***annotations*)
Counts the citations in a graph based on a given filter

Parameters

- **graph** (BELGraph) – A BEL graph
- **annotations** (*dict*) – The annotation filters to use

Return type Counter

Returns A counter from {(citation type, citation reference): frequency}

pybel_tools.summary.**count_citations_by_annotation**(*graph*, *annotation*)
Group the citation counters by subgraphs induced by the annotation.

Parameters

- **graph** (BELGraph) – A BEL graph
- **annotation** (str) – The annotation to use to group the graph

Return type Mapping[str, Counter[str]]

Returns A dictionary of Counters {subgraph name: Counter from {citation: frequency}}

pybel_tools.summary.**count_authors**(*graph*)
Count the number of edges in which each author appears.

Return type Counter[str]

pybel_tools.summary.**count_author_publications**(*graph*)
Count the number of publications of each author to the given graph.

Return type Counter[str]

pybel_tools.summary.**get_authors**(*graph*)
Get the set of all authors in the given graph.

Return type Set[str]

pybel_tools.summary.**get_authors_by_keyword**(*keyword*, *graph=None*, *authors=None*)
Get authors for whom the search term is a substring.

Parameters

- **graph** (*pybel.BELGraph*) – A BEL graph
- **keyword** (str) – The keyword to search the author strings for
- **authors** (*set[str]*) – An optional set of pre-cached authors calculated from the graph

Return type Set[str]

Returns A set of authors with the keyword as a substring

pybel_tools.summary.**count_authors_by_annotation**(*graph*, *annotation='Subgraph'*)
Group the author counters by sub-graphs induced by the annotation.

Parameters

- **graph** (BELGraph) – A BEL graph

- **annotation** (str) – The annotation to use to group the graph

Return type Mapping[str, Counter[str]]

Returns A dictionary of Counters {subgraph name: Counter from {author: frequency}}

pybel_tools.summary.**get_evidences_by_pmid**(*graph*, *pmids*)
Get a dictionary from the given PubMed identifiers to the sets of all evidence strings associated with each in the graph.

Parameters

- **graph** (BELGraph) – A BEL graph
- **pmids** (Union[str, Iterable[str]]) – An iterable of PubMed identifiers, as strings. Is consumed and converted to a set.

Returns A dictionary of {pmid: set of all evidence strings}

Return type dict

pybel_tools.summary.**count_citation_years**(*graph*)
Count the number of citations from each year.

Return type Counter[int]

pybel_tools.summary.**create_timeline**(*year_counter*)
Complete the Counter timeline.

Parameters **year_counter** (*Counter*) – counter dict for each year

Return type List[Tuple[int, int]]

Returns complete timeline

pybel_tools.summary.**get_citation_years**(*graph*)
Create a citation timeline counter from the graph.

Return type List[Tuple[int, int]]

pybel_tools.summary.**count_confidences**(*graph*)
Count the confidences in the graph.

Return type Counter[str]

2.4 Filters

Filters to supplement pybel.struct.filters.

Node filters to supplement pybel.struct.filters.node_filters.

pybel_tools.filters.node_filters.**summarize_node_filter**(*graph*, *node_filters*)
Print a summary of the number of nodes passing a given set of filters.

Parameters

- **graph** (BELGraph) – A BEL graph
- **node_filters** (Union[Callable[[BELGraph, BaseEntity], bool], Iterable[Callable[[BELGraph, BaseEntity], bool]]]) – A node filter or list/tuple of node filters

Return type None

pybel_tools.filters.node_filters.**node_inclusion_filter_builder**(*nodes*)
Build a filter that only passes on nodes in the given list.

Parameters **nodes** (Iterable[BaseEntity]) – An iterable of BEL nodes

Return type Callable[[BELGraph, BaseEntity], bool]

pybel_tools.filters.node_filters.**node_exclusion_filter_builder**(*nodes*)
Build a filter that fails on nodes in the given list.

Return type Callable[[BELGraph, BaseEntity], bool]

pybel_tools.filters.node_filters.**function_inclusion_filter_builder**(*func*)
Build a filter that only passes on nodes of the given function(s).

Parameters **func** (Union[str, Iterable[str]]) – A BEL Function or list/set/tuple of BEL functions

Return type Callable[[BELGraph, BaseEntity], bool]

pybel_tools.filters.node_filters.**function_exclusion_filter_builder**(*func*)
Build a filter that fails on nodes of the given function(s).

Parameters **func** (Union[str, Iterable[str]]) – A BEL Function or list/set/tuple of BEL functions

Return type Callable[[BELGraph, BaseEntity], bool]

pybel_tools.filters.node_filters.**function_namespace_inclusion_builder**(*func*, *namespace*)
Build a filter function for matching the given BEL function with the given namespace or namespaces.

Parameters

- **func** (str) – A BEL function
- **namespace** (Union[str, Iterable[str]]) – The namespace to search by

Return type Callable[[BELGraph, BaseEntity], bool]

pybel_tools.filters.node_filters.**data_contains_key_builder**(*key*)
Build a filter that passes only on nodes that have the given key in their data dictionary.

Parameters **key** (str) – A key for the node's data dictionary

Return type Callable[[BELGraph, BaseEntity], bool]

pybel_tools.filters.node_filters.**node_has_label**(*_*, *node*)
Passes for nodes that have been annotated with a label

Return type bool

pybel_tools.filters.node_filters.**node_missing_label**(*graph*, *node*)
Fails for nodes that have been annotated with a label

Return type bool

pybel_tools.filters.node_filters.**include_pathology_filter**(*_*, *node*)
A filter that passes for nodes that are pybel.constants.PATHOLOGY

Return type bool

pybel_tools.filters.node_filters.**exclude_pathology_filter**(*_*, *node*)
A filter that fails for nodes that are pybel.constants.PATHOLOGY

Return type bool

pybel_tools.filters.node_filters.**variants_of**(*graph*, *node*, *modifications=None*)
Returns all variants of the given node.

Return type Set[Protein]

pybel_tools.filters.node_filters.**get_variants_to_controllers**(*graph*, *node*, *modifications=None*)
Get a mapping from variants of the given node to all of its upstream controllers.

Return type Mapping[Protein, Set[Protein]]

pybel_tools.filters.node_filters.**data_missing_key_builder**(*key*)
Build a filter that passes only on nodes that don't have the given key in their data dictionary.

Parameters **key** (*str*) – A key for the node's data dictionary

Return type Callable[[BELGraph, BaseEntity], bool]

pybel_tools.filters.node_filters.**build_node_data_search**(*key*, *data_predicate*)
Build a filter for nodes whose associated data with the given key passes the given predicate.

Parameters

- **key** (str) – The node data dictionary key to check
- **data_predicate** (Callable[[Any], bool]) – The filter to apply to the node data dictionary

Return type Callable[[BELGraph, BaseEntity], bool]

pybel_tools.filters.node_filters.**build_node_key_search**(*query*, *key*)
Build a node filter for nodes whose values for the given key are superstrings of the query string(s).

Parameters

- **query** (*str or iter[str]*) – The query string or strings to check if they're in the node name
- **key** (*str*) – The key for the node data dictionary. Should refer only to entries that have str values

Return type Callable[[BELGraph, BaseEntity], bool]

Edge filters to supplement pybel.struct.filters.edge_filters.

pybel_tools.filters.edge_filters.**summarize_edge_filter**(*graph*, *edge_predicates*)
Print a summary of the number of edges passing a given set of filters.

Return type None

pybel_tools.filters.edge_filters.**build_edge_data_filter**(*annotations*, *partial_match=True*)
Build a filter that keeps edges whose data dictionaries are super-dictionaries to the given dictionary.

Parameters

- **annotations** (Mapping[~KT, +VT_co]) – The annotation query dict to match
- **partial_match** (bool) – Should the query values be used as partial or exact matches? Defaults to True.

Return type Callable[[BELGraph, BaseEntity, BaseEntity, str], bool]

pybel_tools.filters.edge_filters.**build_pmid_inclusion_filter**(*pmids*)
Pass for edges with citations whose references are one of the given PubMed identifiers.

Parameters **pmids** (Union[str, Iterable[str]]) – A PubMed identifier or list of PubMed identifiers to filter for

Return type Callable[[BELGraph, BaseEntity, BaseEntity, str], bool]

pybel_tools.filters.edge_filters.**build_pmid_exclusion_filter**(*pmids*)
Fail for edges with citations whose references are one of the given PubMed identifiers.

Parameters **pmids** (Union[str, Iterable[str]]) – A PubMed identifier or list of PubMed identifiers to filter against

Return type Callable[[BELGraph, BaseEntity, BaseEntity, str], bool]

pybel_tools.filters.edge_filters.**build_author_inclusion_filter**(*authors*)
Pass only for edges with author information that matches one of the given authors.

Parameters **authors** (Union[str, Iterable[str]]) – The author or list of authors to filter by

Return type Callable[[BELGraph, BaseEntity, BaseEntity, str], bool]

pybel_tools.filters.edge_filters.**build_source_namespace_filter**(*namespaces*)
Pass for edges whose source nodes have the given namespace or one of the given namespaces.

Parameters **namespaces** (Union[str, Iterable[str]]) – The namespace or namespaces to filter by

Return type Callable[[BELGraph, BaseEntity, BaseEntity, str], bool]

pybel_tools.filters.edge_filters.**build_target_namespace_filter**(*namespaces*)
Only passes for edges whose target nodes have the given namespace or one of the given namespaces

Parameters **namespaces** (Union[str, Iterable[str]]) – The namespace or namespaces to filter by

Return type Callable[[BELGraph, BaseEntity, BaseEntity, str], bool]

pybel_tools.filters.edge_filters.**build_annotation_dict_all_filter**(*annotations*)
Build an edge predicate for edges whose annotations are super-dictionaries of the given dictionary.

If no annotations are given, will always evaluate to true.

Parameters **annotations** (Mapping[str, Iterable[str]]) – The annotation query dict to match

Return type Callable[[BELGraph, BaseEntity, BaseEntity, str], bool]

pybel_tools.filters.edge_filters.**build_annotation_dict_any_filter**(*annotations*)
Build an edge predicate that passes for edges whose data dictionaries match the given dictionary.

If the given dictionary is empty, will always evaluate to true.

Parameters **annotations** (Mapping[str, Iterable[str]]) – The annotation query dict to match

Return type Callable[[BELGraph, BaseEntity, BaseEntity, str], bool]

pybel_tools.filters.edge_filters.**has_pathology_causal**(*graph*, *u*, *v*, *k*)
Check if the subject is a pathology and has a causal relationship with a non bioprocess/pathology.

Return type bool

Returns If the subject of this edge is a pathology and it participates in a causal reaction.

2.5 Selection

This module contains functions to help select data from networks

pybel_tools.selection.**group_nodes_by_annotation**(*graph*, *annotation='Subgraph'*)
Group the nodes occurring in edges by the given annotation.

Return type Mapping[str, Set[BaseEntity]]

pybel_tools.selection.**average_node_annotation**(*graph*, *key*, *annotation='Subgraph'*, *aggregator=None*)
Groups graph into subgraphs and assigns each subgraph a score based on the average of all nodes values for the given node key

Parameters

- **graph** (*pybel.BELGraph*) – A BEL graph
- **key** (str) – The key in the node data dictionary representing the experimental data
- **annotation** (str) – A BEL annotation to use to group nodes
- **aggregator** (*lambda*) – A function from list of values -> aggregate value. Defaults to taking the average of a list of floats.

Return type Mapping[str, ~X]

pybel_tools.selection.**group_nodes_by_annotation_filtered**(*graph*, *node_predicates=None*, *annotation='Subgraph'*)
Group the nodes occurring in edges by the given annotation, with a node filter applied.

Parameters

- **graph** (BELGraph) – A BEL graph
- **node_predicates** (Union[Callable[[BELGraph, BaseEntity], bool], Iterable[Callable[[BELGraph, BaseEntity], bool]], None]) – A predicate or list of predicates (graph, node) -> bool
- **annotation** (str) – The annotation to use for grouping

Return type Mapping[str, Set[BaseEntity]]

Returns A dictionary of {annotation value: set of nodes}

pybel_tools.selection.**get_subgraph_by_node_filter**(*graph*, *node_predicates*)
Induce a sub-graph on the nodes that pass the given predicate(s).

Return type BELGraph

pybel_tools.selection.**get_causal_subgraph**(*graph*)
Build a new sub-graph induced over the causal edges.

Return type BELGraph

pybel_tools.selection.**get_subgraph_by_node_search**(*graph*, *query*)
Get a sub-graph induced over all nodes matching the query string.

Parameters

- **graph** (BELGraph) – A BEL Graph
- **query** (Union[str, Iterable[str]]) – A query string or iterable of query strings for node names

Thinly wraps *search_node_names()* and get_subgraph_by_induction().

Return type BELGraph

pybel_tools.selection.**get_largest_component**(*graph*)
Get the giant component of a graph.

Return type BELGraph

pybel_tools.selection.**get_leaves_by_type**(*graph*, *func=None*, *prune_threshold=1*)

Returns an iterable over all nodes in graph (in-place) with only a connection to one node. Useful for gene and RNA. Allows for optional filter by function type.

Parameters

- **graph** (*pybel.BELGraph*) – A BEL graph
- **func** (*str*) – If set, filters by the node's function from pybel.constants like pybel.constants.GENE, pybel.constants.RNA, pybel.constants.PROTEIN, or pybel.constants.BIOPROCESS
- **prune_threshold** (*int*) – Removes nodes with less than or equal to this number of connections. Defaults to 1

Returns An iterable over nodes with only a connection to one node

Return type iter[tuple]

pybel_tools.selection.**get_nodes_in_all_shortest_paths**(*graph*, *nodes*, *weight=None*, *remove_pathologies=False*)
Get a set of nodes in all shortest paths between the given nodes.

Thinly wraps networkx.all_shortest_paths().

Parameters

- **graph** (*pybel.BELGraph*) – A BEL graph
- **nodes** (*iter[tuple]*) – The list of nodes to use to use to find all shortest paths
- **weight** (*Optional[str]*) – Edge data key corresponding to the edge weight. If none, uses unweighted search.
- **remove_pathologies** (*bool*) – Should pathology nodes be removed first?

Returns A set of nodes appearing in the shortest paths between nodes in the BEL graph

Return type set[tuple]

Note: This can be trivially parallelized using networkx.single_source_shortest_path()

---

pybel_tools.selection.**get_shortest_directed_path_between_subgraphs**(*graph*, *a*, *b*)
Calculate the shortest path that occurs between two disconnected subgraphs A and B going through nodes in the source graph

Parameters

- **graph** (*pybel.BELGraph*) – A BEL graph
- **a** (*pybel.BELGraph*) – A subgraph of graph, disjoint from b
- **b** (*pybel.BELGraph*) – A subgraph of graph, disjoint from a

Returns A list of the shortest paths between the two subgraphs

Return type list

pybel_tools.selection.**get_shortest_undirected_path_between_subgraphs**(*graph*, *a*, *b*)

Get the shortest path between two disconnected subgraphs A and B, disregarding directionality of edges in graph

Parameters

- **graph** (*pybel.BELGraph*) – A BEL graph
- **a** (*pybel.BELGraph*) – A subgraph of `graph`, disjoint from `b`
- **b** (*pybel.BELGraph*) – A subgraph of `graph`, disjoint from `a`

Returns A list of the shortest paths between the two subgraphs

Return type list

pybel_tools.selection.**search_node_names**(*graph*, *query*)

Search for nodes containing a given string(s).

Parameters

- **graph** (*pybel.BELGraph*) – A BEL graph
- **query** (*str or iter[str]*) – The search query

Returns An iterator over nodes whose names match the search query

Return type iter

Example:

```
>>> from pybel.examples import sialic_acid_graph
>>> from pybel_tools.selection import search_node_names
>>> list(search_node_names(sialic_acid_graph, 'CD33'))
[('Protein', 'HGNC', 'CD33'), ('Protein', 'HGNC', 'CD33', ('pmod', ('bel', 'Ph
↪'')))]
```

pybel_tools.selection.**search_node_namespace_names**(*graph*, *query*, *namespace*)

Search for nodes with the given namespace(s) and whose names containing a given string(s).

Parameters

- **graph** (*pybel.BELGraph*) – A BEL graph
- **query** (*str or iter[str]*) – The search query
- **namespace** (*str or iter[str]*) – The namespace(s) to filter

Returns An iterator over nodes whose names match the search query

Return type iter

pybel_tools.selection.**search_node_hgnc_names**(*graph*, *query*)

Search for nodes with the HGNC namespace and whose names containing a given string(s).

Parameters

- **graph** (*pybel.BELGraph*) – A BEL graph
- **query** (*str or iter[str]*) – The search query

Returns An iterator over nodes whose names match the search query

Return type iter

pybel_tools.selection.**convert_path_to_metapath**(*graph*, *nodes*)
Converts a list of nodes to their corresponding functions

Parameters **nodes** (*list[tuple]*) – A list of BEL node tuples

Return type list[str]

pybel_tools.selection.**get_walks_exhaustive**
Gets all walks under a given length starting at a given node

Parameters

- **graph** (*networkx.Graph*) – A graph
- **node** – Starting node
- **length** (*int*) – The length of walks to get

Returns A list of paths

Return type list[tuple]

pybel_tools.selection.**match_simple_metapath**(*graph*, *node*, *simple_metapath*)
Matches a simple metapath starting at the given node

Parameters

- **graph** (*pybel.BELGraph*) – A BEL graph
- **node** (*tuple*) – A BEL node
- **simple_metapath** (*list[str]*) – A list of BEL Functions

Returns An iterable over paths from the node matching the metapath

Return type iter[tuple]

2.6 Integration

This module contains functions that help add more data to the network

pybel_tools.integration.**overlay_data**(*graph*, *data*, *label=None*, *overwrite=False*)
Overlays tabular data on the network

Parameters

- **graph** (BELGraph) – A BEL Graph
- **data** (Mapping[BaseEntity, Any]) – A dictionary of {tuple node: data for that node}
- **label** (Optional[str]) – The annotation label to put in the node dictionary
- **overwrite** (bool) – Should old annotations be overwritten?

Return type None

pybel_tools.integration.**overlay_type_data**(*graph*, *data*, *func*, *namespace*, *label=None*, *overwrite=False*, *impute=None*)
Overlay tabular data on the network for data that comes from an data set with identifiers that lack namespaces.

For example, if you want to overlay differential gene expression data from a table, that table probably has HGNC identifiers, but no specific annotations that they are in the HGNC namespace or that the entities to which they refer are RNA.

Parameters

- **graph** (BELGraph) – A BEL Graph
- **data** (*dict*) – A dictionary of {name: data}
- **func** (str) – The function of the keys in the data dictionary
- **namespace** (str) – The namespace of the keys in the data dictionary
- **label** (Optional[str]) – The annotation label to put in the node dictionary
- **overwrite** (bool) – Should old annotations be overwritten?
- **impute** (Optional[float]) – The value to use for missing data

Return type None

pybel_tools.integration.**load_differential_gene_expression**(*path*, *gene_symbol_column='Gene.symbol'*, *logfc_column='logFC'*, *aggregator=None*)

Load and pre-process a differential gene expression data.

Parameters

- **path** (str) – The path to the CSV
- **gene_symbol_column** (str) – The header of the gene symbol column in the data frame
- **logfc_column** (str) – The header of the log-fold-change column in the data frame
- **aggregator** (Optional[Callable[[List[float]], float]]) – A function that aggregates a list of differential gene expression values. Defaults to `numpy.median()`. Could also use: `numpy.mean()`, `numpy.average()`, `numpy.min()`, or `numpy.max()`

Return type Mapping[str, float]

Returns A dictionary of {gene symbol: log fold change}

2.7 Mutation

Mutation functions to supplement pybel.struct.mutation.

pybel_tools.mutation.**collapse_nodes**(*graph*, *survivor_mapping*)

Collapse all nodes in values to the key nodes, in place.

Parameters

- **graph** (*pybel.BELGraph*) – A BEL graph
- **survivor_mapping** (Mapping[BaseEntity, Set[BaseEntity]]) – A dictionary with survivors as their keys, and iterables of the corresponding victims as values.

Return type None

pybel_tools.mutation.**rewire_variants_to_genes**(*graph*)

Find all protein variants that are pointing to a gene and not a protein and fixes them by changing their function to be pybel.constants.GENE, in place

A use case is after running `collapse_to_genes()`.

Return type None

pybel_tools.mutation.**collapse_gene_variants**(*graph*)
Collapse all gene's variants' edges to their parents, in-place.

Return type None

pybel_tools.mutation.**collapse_protein_variants**(*graph*)
Collapse all protein's variants' edges to their parents, in-place.

Return type None

pybel_tools.mutation.**collapse_consistent_edges**(*graph*)
Collapse consistent edges together.

Warning: This operation doesn't preserve evidences or other annotations

pybel_tools.mutation.**collapse_equivalencies_by_namespace**(*graph*, *victim_namespace*, *survivor_namespace*)
Collapse pairs of nodes with the given namespaces that have equivalence relationships.

Parameters

- **graph** (BELGraph) – A BEL graph
- **victim_namespace** (Union[str, Iterable[str]]) – The namespace(s) of the node to collapse
- **survivor_namespace** (str) – The namespace of the node to keep

To convert all ChEBI names to InChI keys, assuming there are appropriate equivalence relations between nodes with those namespaces:

```
>>> collapse_equivalencies_by_namespace(graph, 'CHEBI', 'CHEBIID')
>>> collapse_equivalencies_by_namespace(graph, 'CHEBIID', 'INCHI')
```

Return type None

pybel_tools.mutation.**collapse_orthologies_by_namespace**(*graph*, *victim_namespace*, *survivor_namespace*)
Collapse pairs of nodes with the given namespaces that have orthology relationships.

Parameters

- **graph** (BELGraph) – A BEL Graph
- **victim_namespace** (Union[str, Iterable[str]]) – The namespace(s) of the node to collapse
- **survivor_namespace** (str) – The namespace of the node to keep

To collapse all MGI nodes to their HGNC orthologs, use: >>> collapse_orthologies_by_namespace('MGI', 'HGNC')

To collapse collapse both MGI and RGD nodes to their HGNC orthologs, use: >>> collapse_orthologies_by_namespace(['MGI', 'RGD'], 'HGNC')

Return type None

pybel_tools.mutation.**collapse_to_protein_interactions**(*graph*)
Collapse to a graph made of only causal gene/protein edges.

Return type BELGraph

pybel_tools.mutation.**collapse_nodes_with_same_names**(*graph*)
Collapse all nodes with the same name, merging namespaces by picking first alphabetical one.

Return type None

pybel_tools.mutation.**remove_inconsistent_edges**(*graph*)
Remove all edges between node pairs with inconsistent edges.

This is the all-or-nothing approach. It would be better to do more careful investigation of the evidences during curation.

Return type None

pybel_tools.mutation.**get_peripheral_successor_edges**(*graph*, *subgraph*)
Get the set of possible successor edges peripheral to the sub-graph.

The source nodes in this iterable are all inside the sub-graph, while the targets are outside.

Return type Iterable[Tuple[BaseEntity, BaseEntity, str]]

pybel_tools.mutation.**get_peripheral_predecessor_edges**(*graph*, *subgraph*)
Get the set of possible predecessor edges peripheral to the sub-graph.

The target nodes in this iterable are all inside the sub-graph, while the sources are outside.

Return type Iterable[Tuple[BaseEntity, BaseEntity, str]]

pybel_tools.mutation.**count_sources**(*edge_iter*)
Count the source nodes in an edge iterator with keys and data.

Return type Counter

Returns A counter of source nodes in the iterable

pybel_tools.mutation.**count_targets**(*edge_iter*)
Count the target nodes in an edge iterator with keys and data.

Return type Counter

Returns A counter of target nodes in the iterable

pybel_tools.mutation.**count_possible_successors**(*graph*, *subgraph*)

Parameters

- **graph** (BELGraph) – A BEL graph
- **subgraph** (BELGraph) – An iterator of BEL nodes

Return type Counter

Returns A counter of possible successor nodes

pybel_tools.mutation.**count_possible_predecessors**(*graph*, *subgraph*)

Parameters

- **graph** (BELGraph) – A BEL graph
- **subgraph** (BELGraph) – An iterator of BEL nodes

Return type Counter

Returns A counter of possible predecessor nodes

pybel_tools.mutation.**get_subgraph_edges**(*graph*, *annotation*, *value*, *source_filter=None*, *target_filter=None*)
Gets all edges from a given subgraph whose source and target nodes pass all of the given filters

Parameters

- **graph** (*pybel.BELGraph*) – A BEL graph
- **annotation** (*str*) – The annotation to search
- **value** (*str*) – The annotation value to search by
- **source_filter** – Optional filter for source nodes (graph, node) -> bool
- **target_filter** – Optional filter for target nodes (graph, node) -> bool

Returns An iterable of (source node, target node, key, data) for all edges that match the annotation/value and node filters

Return type iter[tuple]

pybel_tools.mutation.**get_subgraph_peripheral_nodes**(*graph*, *subgraph*, *node_predicates=None*, *edge_predicates=None*)

Get a summary dictionary of all peripheral nodes to a given sub-graph.

Returns A dictionary of {external node: {'successor': {internal node: list of (key, dict)}, 'predecessor': {internal node: list of (key, dict)}}}

Return type dict

For example, it might be useful to quantify the number of predecessors and successors:

```
>>> from pybel.struct.filters import exclude_pathology_filter
>>> value = 'Blood vessel dilation subgraph'
>>> sg = get_subgraph_by_annotation_value(graph, annotation='Subgraph',␣
↪value=value)
>>> p = get_subgraph_peripheral_nodes(graph, sg, node_predicates=exclude_
↪pathology_filter)
>>> for node in sorted(p, key=lambda n: len(set(p[n]['successor']) | set(p[n][
↪'predecessor'])), reverse=True):
>>>     if 1 == len(p[value][node]['successor']) or 1 == len(p[value][node][
↪'predecessor']):
>>>         continue
>>>     print(node,
>>>           len(p[node]['successor']),
>>>           len(p[node]['predecessor']),
>>>           len(set(p[node]['successor']) | set(p[node]['predecessor'])))
```

pybel_tools.mutation.**expand_periphery**(*universe*, *graph*, *node_predicates=None*, *edge_predicates=None*, *threshold=2*)

Iterates over all possible edges, peripheral to a given subgraph, that could be added from the given graph. Edges could be added if they go to nodes that are involved in relationships that occur with more than the threshold (default 2) number of nodes in the subgraph.

Parameters

- **universe** (BELGraph) – The universe of BEL knowledge
- **graph** (BELGraph) – The (sub)graph to expand
- **threshold** (int) – Minimum frequency of betweenness occurrence to add a gap node

A reasonable edge filter to use is `pybel_tools.filters.keep_causal_edges()` because this function can allow for huge expansions if there happen to be hub nodes.

Return type None

pybel_tools.mutation.**enrich_complexes**(*graph*)
Add all of the members of the complex abundances to the graph.

Return type None

pybel_tools.mutation.**enrich_composites**(*graph*)
Adds all of the members of the composite abundances to the graph.

pybel_tools.mutation.**enrich_reactions**(*graph*)
Adds all of the reactants and products of reactions to the graph.

pybel_tools.mutation.**enrich_variants**(*graph*, *func=None*)
Add the reference nodes for all variants of the given function.

Parameters

- **graph** (BELGraph) – The target BEL graph to enrich
- **func** (Union[None, str, Iterable[str]]) – The function by which the subject of each triple is filtered. Defaults to the set of protein, rna, mirna, and gene.

pybel_tools.mutation.**enrich_unqualified**(*graph*)
Enrich the sub-graph with the unqualified edges from the graph.

The reason you might want to do this is you induce a sub-graph from the original graph based on an annotation filter, but the unqualified edges that don't have annotations that most likely connect elements within your graph are not included.

See also:

This function thinly wraps the successive application of the following functions:

- *enrich_complexes()*
- *enrich_composites()*
- *enrich_reactions()*
- *enrich_variants()*

Equivalent to:

```
>>> enrich_complexes(graph)
>>> enrich_composites(graph)
>>> enrich_reactions(graph)
>>> enrich_variants(graph)
```

pybel_tools.mutation.**expand_internal**(*universe*, *graph*, *edge_predicates=None*)
Edges between entities in the sub-graph that pass the given filters.

Parameters

- **universe** (BELGraph) – The full graph
- **graph** (BELGraph) – A sub-graph to find the upstream information
- **edge_predicates** (Union[Callable[[BELGraph, BaseEntity, BaseEntity, str], bool], Iterable[Callable[[BELGraph, BaseEntity, BaseEntity, str], bool]], None]) – Optional list of edge filter functions (graph, node, node, key, data) -> bool

Return type None

pybel_tools.mutation.**expand_internal_causal**(*universe*, *graph*)
Add causal edges between entities in the sub-graph.

Is an extremely thin wrapper around *expand_internal()*.

Parameters

- **universe** (BELGraph) – A BEL graph representing the universe of all knowledge
- **graph** (BELGraph) – The target BEL graph to enrich with causal relations between contained nodes

Equivalent to:

```
>>> from pybel_tools.mutation import expand_internal
>>> from pybel.struct.filters.edge_predicates import is_causal_relation
>>> expand_internal(universe, graph, edge_predicates=is_causal_relation)
```

Return type None

pybel_tools.mutation.**is_node_highlighted**(*graph*, *node*)
Returns if the given node is highlighted.

Parameters

- **graph** (BELGraph) – A BEL graph
- **node** (*tuple*) – A BEL node

Returns Does the node contain highlight information?

Return type bool

pybel_tools.mutation.**highlight_nodes**(*graph*, *nodes=None*, *color=None*)
Adds a highlight tag to the given nodes.

Parameters

- **graph** (BELGraph) – A BEL graph
- **nodes** (Optional[Iterable[BaseEntity]]) – The nodes to add a highlight tag on
- **color** (Optional[str]) – The color to highlight (use something that works with CSS)

pybel_tools.mutation.**remove_highlight_nodes**(*graph*, *nodes=None*)
Removes the highlight from the given nodes, or all nodes if none given.

Parameters

- **graph** (BELGraph) – A BEL graph
- **nodes** (Optional[Iterable[BaseEntity]]) – The list of nodes to un-highlight

Return type None

pybel_tools.mutation.**is_edge_highlighted**(*graph*, *u*, *v*, *k*)
Returns if the given edge is highlighted.

Parameters **graph** (BELGraph) – A BEL graph

Returns Does the edge contain highlight information?

Return type bool

pybel_tools.mutation.**highlight_edges**(*graph*, *edges=None*, *color=None*)
Adds a highlight tag to the given edges.

Parameters

- **graph** (BELGraph) – A BEL graph

- **edges** (*iter[tuple]*) – The edges (4-tuples of u, v, k, d) to add a highlight tag on
- **color** (*str*) – The color to highlight (use something that works with CSS)

Return type None

pybel_tools.mutation.**remove_highlight_edges**(*graph*, *edges=None*)
Remove the highlight from the given edges, or all edges if none given.

Parameters

- **graph** (BELGraph) – A BEL graph
- **edges** (*iter[tuple]*) – The edges (4-tuple of u,v,k,d) to remove the highlight from)

pybel_tools.mutation.**highlight_subgraph**(*universe*, *graph*)
Highlight all nodes/edges in the universe that in the given graph.

Parameters

- **universe** (BELGraph) – The universe of knowledge
- **graph** (BELGraph) – The BEL graph to mutate

pybel_tools.mutation.**remove_highlight_subgraph**(*graph*, *subgraph*)
Remove the highlight from all nodes/edges in the graph that are in the subgraph.

Parameters

- **graph** (BELGraph) – The BEL graph to mutate
- **subgraph** (BELGraph) – The subgraph from which to remove the highlighting

pybel_tools.mutation.**enrich_protein_and_rna_origins**(*graph*)
Add the corresponding RNA for each protein then the corresponding gene for each RNA/miRNA.

Parameters **graph** (*pybel.BELGraph*) – A BEL graph

pybel_tools.mutation.**infer_missing_two_way_edges**(*graph*)
Add edges to the graph when a two way edge exists, and the opposite direction doesn't exist.

Use: two way edges from BEL definition and/or axiomatic inverses of membership relations

Parameters **graph** (*pybel.BELGraph*) – A BEL graph

pybel_tools.mutation.**infer_missing_backwards_edge**(*graph*, *u*, *v*, *k*)
Add the same edge, but in the opposite direction if not already present.

pybel_tools.mutation.**enrich_internal_unqualified_edges**(*graph*, *subgraph*)
Add the missing unqualified edges between entities in the subgraph that are contained within the full graph.

Parameters

- **graph** (*pybel.BELGraph*) – The full BEL graph
- **subgraph** (*pybel.BELGraph*) – The query BEL subgraph

pybel_tools.mutation.**enrich_pubmed_citations**(*graph*, *manager*)
Overwrite all PubMed citations with values from NCBI's eUtils lookup service.

Return type Set[str]

Returns A set of PMIDs for which the eUtils service crashed

pybel_tools.mutation.**random_by_nodes**(*graph*, *percentage=None*)
Get a random graph by inducing over a percentage of the original nodes.

Parameters

- **graph** (BELGraph) – A BEL graph
- **percentage** (Optional[float]) – The percentage of edges to keep

Return type BELGraph

pybel_tools.mutation.**random_by_edges**(*graph*, *percentage=None*)
Get a random graph by keeping a certain percentage of original edges.

Parameters

- **graph** (BELGraph) – A BEL graph
- **percentage** (Optional[float]) – What percentage of eges to take

Return type BELGraph

pybel_tools.mutation.**shuffle_node_data**(*graph*, *key*, *percentage=None*)
Shuffle the node's data.

Useful for permutation testing.

Parameters

- **graph** (BELGraph) – A BEL graph
- **key** (str) – The node data dictionary key
- **percentage** (Optional[float]) – What percentage of possible swaps to make

Return type BELGraph

pybel_tools.mutation.**shuffle_relations**(*graph*, *percentage=None*)
Shuffle the relations.

Useful for permutation testing.

Parameters

- **graph** (BELGraph) – A BEL graph
- **percentage** (Optional[str]) – What percentage of possible swaps to make

Return type BELGraph

pybel_tools.mutation.**build_expand_node_neighborhood_by_hash**(*manager*)
Make an expand function that's bound to the manager.

Return type Callable[[BELGraph, BELGraph, str], None]

pybel_tools.mutation.**build_delete_node_by_hash**(*manager*)
Make a delete function that's bound to the manager.

Return type Callable[[BELGraph, str], None]

2.8 Reverse Causal Reasoning

An implementation of Reverse Causal Reasoning (RCR) described by [Catlett2013].

pybel_tools.analysis.rcr.**run_rcr**(*graph*, *tag='dgxp'*)
Run the reverse causal reasoning algorithm on a graph.

Steps:

1. Get all downstream controlled things into map (that have at least 4 downstream things)
2. calculate population of all things that are downstream controlled

Note: Assumes all nodes have been pre-tagged with data

Parameters

- **graph** (*pybel.BELGraph*) –
- **tag** (*str*) – The key for the nodes' data dictionaries that corresponds to the integer value for its differential expression.

2.9 CausalR

An implementation of the CausalR algorithm described by [Bradley2017].

pybel_tools.analysis.causalr.**rank_causalr_hypothesis**(*graph*, *node_to_regulation*, *regulator_node*)

Test the regulator hypothesis of the given node on the input data using the algorithm.

Note: this method returns both +/- signed hypotheses evaluated

Algorithm:

1. Calculate the shortest path between the regulator node and each node in observed_regulation
2. Calculate the concordance of the causal network and the observed regulation when there is path between target node and regulator node

Parameters

- **graph** (*networkx.DiGraph*) – A causal graph
- **node_to_regulation** (*dict*) – Nodes to score (1,-1,0)

Return Dictionaries with hypothesis results (keys score, correct, incorrect, ambiguous)

Return type dict

2.10 SPIA

An exporter for signaling pathway impact analysis (SPIA) described by [Tarca2009].

To run this module on an arbitrary BEL graph, use the command `python -m pybel_tools.analysis.spia`.

See also:

https://bioconductor.org/packages/release/bioc/html/SPIA.html

pybel_tools.analysis.spia.**bel_to_spia_matrices**(*graph*)

Create an excel sheet ready to be used in SPIA software.

Parameters **graph** (BELGraph) – BELGraph

Return type Mapping[str, DataFrame]

Returns dictionary with matrices

pybel_tools.analysis.spia.**spia_matrices_to_excel**(*spia_matrices*, *path*)
Export a SPIA data dictionary into an Excel sheet at the given path.

Note: # The R import should add the values: # ["nodes"] from the columns # ["title"] from the name of the file # ["NumberOfReactions"] set to "0"

Return type None

pybel_tools.analysis.spia.**spia_matrices_to_tsvs**(*spia_matrices*, *directory*)
Export a SPIA data dictionary into a directory as several TSV documents.

Return type None

2.11 NeuroMMSig

An implementation of the NeuroMMSig mechanism enrichment algorithm [DomingoFernandez2017].

pybel_tools.analysis.neurommsig.algorithm.**get_neurommsig_scores**(*graph*, *genes*, *annotation='Subgraph'*, *ora_weight=None*, *hub_weight=None*, *top_percent=None*, *topology_weight=None*, *preprocess=False*)
Preprocess the graph, stratify by the given annotation, then run the NeuroMMSig algorithm on each.

Parameters

- **graph** (BELGraph) – A BEL graph
- **genes** (List[Gene]) – A list of gene nodes
- **annotation** (str) – The annotation to use to stratify the graph to subgraphs
- **ora_weight** (Optional[float]) – The relative weight of the over-enrichment analysis score from neurommsig_gene_ora(). Defaults to 1.0.
- **hub_weight** (Optional[float]) – The relative weight of the hub analysis score from neurommsig_hubs(). Defaults to 1.0.
- **top_percent** (Optional[float]) – The percentage of top genes to use as hubs. Defaults to 5% (0.05).
- **topology_weight** (Optional[float]) – The relative weight of the topolgical analysis core from neurommsig_topology(). Defaults to 1.0.
- **preprocess** (bool) – If true, preprocess the graph.

Return type Optional[Mapping[str, float]]

Returns A dictionary from {annotation value: NeuroMMSig composite score}

Pre-processing steps:

1. Infer the central dogma with :func:‘‘

2. Collapse all proteins, RNAs and miRNAs to genes with :func:‘‘
3. Collapse variants to genes with :func:‘‘

pybel_tools.analysis.neurommsig.algorithm.**get_neurommsig_score**(*graph*, *genes*, *ora_weight=None*, *hub_weight=None*, *top_percent=None*, *topology_weight=None*)

Calculate the composite NeuroMMSig Score for a given list of genes.

Parameters

- **graph** (BELGraph) – A BEL graph
- **genes** (List[Gene]) – A list of gene nodes
- **ora_weight** (Optional[float]) – The relative weight of the over-enrichment analysis score from neurommsig_gene_ora(). Defaults to 1.0.
- **hub_weight** (Optional[float]) – The relative weight of the hub analysis score from neurommsig_hubs(). Defaults to 1.0.
- **top_percent** (Optional[float]) – The percentage of top genes to use as hubs. Defaults to 5% (0.05).
- **topology_weight** (Optional[float]) – The relative weight of the topolgical analysis core from neurommsig_topology(). Defaults to 1.0.

Return type float

Returns The NeuroMMSig composite score

2.12 EpiCom

An implementation of chemical-based mechanism enrichment with NeuroMMSig described by [Hoyt2018].

This algorithm has multiple steps:

1. Select NeuroMMSig networks for AD, PD, and epilepsy
2. Select drugs from DrugBank, and their targets
3. Run NeuroMMSig algorithm on target list for each network and each mechanism
4. Store in database

pybel_tools.analysis.epicom.**multi_run_epicom**(*graphs*, *path*)

Run EpiCom analysis on many graphs.

Return type None

2.13 Stability Analysis

pybel_tools.analysis.stability.**get_contradiction_summary**(*graph*)

Yield triplets of (source node, target node, set of relations) for (source node, target node) pairs that have multiple, contradictory relations.

Return type Iterable[Tuple[BaseEntity, BaseEntity, str]]

pybel_tools.analysis.stability.**get_regulatory_pairs**(*graph*)
Find pairs of nodes that have mutual causal edges that are regulating each other such that `A -> B` and `B -| A`.

Return type Set[Tuple[BaseEntity, BaseEntity]]

Returns A set of pairs of nodes with mutual causal edges

pybel_tools.analysis.stability.**get_chaotic_pairs**(*graph*)
Find pairs of nodes that have mutual causal edges that are increasing each other such that `A -> B` and `B -> A`.

Return type Set[Tuple[BaseEntity, BaseEntity]]

Returns A set of pairs of nodes with mutual causal edges

pybel_tools.analysis.stability.**get_dampened_pairs**(*graph*)
Find pairs of nodes that have mutual causal edges that are decreasing each other such that `A -| B` and `B -| A`.

Return type Set[Tuple[BaseEntity, BaseEntity]]

Returns A set of pairs of nodes with mutual causal edges

pybel_tools.analysis.stability.**get_correlation_graph**(*graph*)
Extract an undirected graph of only correlative relationships.

Return type Graph

pybel_tools.analysis.stability.**get_correlation_triangles**(*graph*)
Return a set of all triangles pointed by the given node.

Return type Set[Tuple[BaseEntity, BaseEntity, BaseEntity]]

pybel_tools.analysis.stability.**get_triangles**(*graph*)
Get a set of triples representing the 3-cycles from a directional graph.

Each 3-cycle is returned once, with nodes in sorted order.

Return type Set[Tuple[BaseEntity, BaseEntity, BaseEntity]]

pybel_tools.analysis.stability.**get_separate_unstable_correlation_triples**(*graph*)
Yield all triples of nodes A, B, C such that `A pos B`, `A pos C`, and `B neg C`.

Return type Iterable[Tuple[BaseEntity, BaseEntity, BaseEntity]]

Returns An iterator over triples of unstable graphs, where the second two are negative

pybel_tools.analysis.stability.**get_mutually_unstable_correlation_triples**(*graph*)
Yield triples of nodes (A, B, C) such that `A neg B`, `B neg C`, and `C neg A`.

Return type Iterable[Tuple[BaseEntity, BaseEntity, BaseEntity]]

pybel_tools.analysis.stability.**jens_transformation_alpha**(*graph*)
Apply Jens' transformation (Type 1) to the graph.

1. Induce a sub-graph over causal + correlative edges
2. **Transform edges by the following rules:**
   - increases => increases
   - decreases => backwards increases
   - positive correlation => two way increases
   - negative correlation => delete

The resulting graph can be used to search for 3-cycles, which now symbolize unstable triplets where `A -> B`, `A -| C` and `B positiveCorrelation C`.

Return type `DiGraph`

`pybel_tools.analysis.stability.`**`jens_transformation_beta`**(*graph*)
Apply Jens' Transformation (Type 2) to the graph.

1. Induce a sub-graph over causal and correlative relations
2. **Transform edges with the following rules:**
   - increases => backwards decreases
   - decreases => decreases
   - positive correlation => delete
   - negative correlation => two way decreases

The resulting graph can be used to search for 3-cycles, which now symbolize stable triples where `A -> B`, `A -| C` and `B negativeCorrelation C`.

Return type `DiGraph`

`pybel_tools.analysis.stability.`**`get_jens_unstable`**(*graph*)
Yield triples of nodes (A, B, C) where `A -> B`, `A -| C`, and `C positiveCorrelation A`.

Calculated efficiently using the Jens Transformation.

Return type `Iterable`[`Tuple`[`BaseEntity`, `BaseEntity`, `BaseEntity`]]

`pybel_tools.analysis.stability.`**`get_increase_mismatch_triplets`**(*graph*)
Yield triples of nodes (A, B, C) where `A -> B`, `A -> C`, and `C negativeCorrelation A`.

Return type `Iterable`[`Tuple`[`BaseEntity`, `BaseEntity`, `BaseEntity`]]

`pybel_tools.analysis.stability.`**`get_decrease_mismatch_triplets`**(*graph*)
Yield triples of nodes (A, B, C) where `A -| B`, `A -| C`, and `C negativeCorrelation A`.

Return type `Iterable`[`Tuple`[`BaseEntity`, `BaseEntity`, `BaseEntity`]]

`pybel_tools.analysis.stability.`**`get_chaotic_triplets`**(*graph*)
Yield triples of nodes (A, B, C) that mutually increase each other, such as when `A -> B`, `B -> C`, and `C -> A`.

Return type `Iterable`[`Tuple`[`BaseEntity`, `BaseEntity`, `BaseEntity`]]

`pybel_tools.analysis.stability.`**`get_dampened_triplets`**(*graph*)
Yield triples of nodes (A, B, C) that mutually decreases each other, such as when `A -| B`, `B -| C`, and `C -| A`.

Return type `Iterable`[`Tuple`[`BaseEntity`, `BaseEntity`, `BaseEntity`]]

`pybel_tools.analysis.stability.`**`summarize_stability`**(*graph*)
Summarize the stability of the graph.

Return type `Mapping`[`str`, `int`]

2.14 Subgraph Expansion Workflow

Deletion functions to supplement `pybel.struct.mutation.expansion`.

pybel_tools.mutation.expansion.**get_peripheral_successor_edges**(*graph*, *subgraph*)
Get the set of possible successor edges peripheral to the sub-graph.

The source nodes in this iterable are all inside the sub-graph, while the targets are outside.

Return type Iterable[Tuple[BaseEntity, BaseEntity, str]]

pybel_tools.mutation.expansion.**get_peripheral_predecessor_edges**(*graph*, *subgraph*)
Get the set of possible predecessor edges peripheral to the sub-graph.

The target nodes in this iterable are all inside the sub-graph, while the sources are outside.

Return type Iterable[Tuple[BaseEntity, BaseEntity, str]]

pybel_tools.mutation.expansion.**count_sources**(*edge_iter*)
Count the source nodes in an edge iterator with keys and data.

Return type Counter

Returns A counter of source nodes in the iterable

pybel_tools.mutation.expansion.**count_targets**(*edge_iter*)
Count the target nodes in an edge iterator with keys and data.

Return type Counter

Returns A counter of target nodes in the iterable

pybel_tools.mutation.expansion.**count_possible_successors**(*graph*, *subgraph*)

Parameters

- **graph** (BELGraph) – A BEL graph
- **subgraph** (BELGraph) – An iterator of BEL nodes

Return type Counter

Returns A counter of possible successor nodes

pybel_tools.mutation.expansion.**count_possible_predecessors**(*graph*, *subgraph*)

Parameters

- **graph** (BELGraph) – A BEL graph
- **subgraph** (BELGraph) – An iterator of BEL nodes

Return type Counter

Returns A counter of possible predecessor nodes

pybel_tools.mutation.expansion.**get_subgraph_edges**(*graph*, *annotation*, *value*, *source_filter=None*, *target_filter=None*)
Gets all edges from a given subgraph whose source and target nodes pass all of the given filters

Parameters

- **graph** (*pybel.BELGraph*) – A BEL graph
- **annotation** (*str*) – The annotation to search
- **value** (*str*) – The annotation value to search by
- **source_filter** – Optional filter for source nodes (graph, node) -> bool
- **target_filter** – Optional filter for target nodes (graph, node) -> bool

Returns An iterable of (source node, target node, key, data) for all edges that match the annotation/value and node filters

Return type iter[tuple]

pybel_tools.mutation.expansion.**get_subgraph_peripheral_nodes**(*graph*, *subgraph*, *node_predicates=None*, *edge_predicates=None*)

Get a summary dictionary of all peripheral nodes to a given sub-graph.

Returns A dictionary of {external node: {'successor': {internal node: list of (key, dict)}, 'predecessor': {internal node: list of (key, dict)}}}

Return type dict

For example, it might be useful to quantify the number of predecessors and successors:

```
>>> from pybel.struct.filters import exclude_pathology_filter
>>> value = 'Blood vessel dilation subgraph'
>>> sg = get_subgraph_by_annotation_value(graph, annotation='Subgraph',␣
↪value=value)
>>> p = get_subgraph_peripheral_nodes(graph, sg, node_predicates=exclude_
↪pathology_filter)
>>> for node in sorted(p, key=lambda n: len(set(p[n]['successor']) | set(p[n][
↪'predecessor'])), reverse=True):
>>>     if 1 == len(p[value][node]['successor']) or 1 == len(p[value][node][
↪'predecessor']):
>>>         continue
>>>     print(node,
>>>           len(p[node]['successor']),
>>>           len(p[node]['predecessor']),
>>>           len(set(p[node]['successor']) | set(p[node]['predecessor'])))
```

pybel_tools.mutation.expansion.**expand_periphery**(*universe*, *graph*, *node_predicates=None*, *edge_predicates=None*, *threshold=2*)

Iterates over all possible edges, peripheral to a given subgraph, that could be added from the given graph. Edges could be added if they go to nodes that are involved in relationships that occur with more than the threshold (default 2) number of nodes in the subgraph.

Parameters

- **universe** (BELGraph) – The universe of BEL knowledge
- **graph** (BELGraph) – The (sub)graph to expand
- **threshold** (int) – Minimum frequency of betweenness occurrence to add a gap node

A reasonable edge filter to use is pybel_tools.filters.keep_causal_edges() because this function can allow for huge expansions if there happen to be hub nodes.

Return type None

pybel_tools.mutation.expansion.**enrich_complexes**(*graph*)

Add all of the members of the complex abundances to the graph.

Return type None

pybel_tools.mutation.expansion.**enrich_composites**(*graph*)

Adds all of the members of the composite abundances to the graph.

pybel_tools.mutation.expansion.**enrich_reactions**(*graph*)

Adds all of the reactants and products of reactions to the graph.

pybel_tools.mutation.expansion.**enrich_variants**(*graph*, *func=None*)
Add the reference nodes for all variants of the given function.

Parameters

- **graph** (BELGraph) – The target BEL graph to enrich
- **func** (Union[None, str, Iterable[str]]) – The function by which the subject of each triple is filtered. Defaults to the set of protein, rna, mirna, and gene.

pybel_tools.mutation.expansion.**enrich_unqualified**(*graph*)
Enrich the sub-graph with the unqualified edges from the graph.

The reason you might want to do this is you induce a sub-graph from the original graph based on an annotation filter, but the unqualified edges that don't have annotations that most likely connect elements within your graph are not included.

See also:

This function thinly wraps the successive application of the following functions:

- *enrich_complexes()*
- *enrich_composites()*
- *enrich_reactions()*
- *enrich_variants()*

Equivalent to:

```
>>> enrich_complexes(graph)
>>> enrich_composites(graph)
>>> enrich_reactions(graph)
>>> enrich_variants(graph)
```

pybel_tools.mutation.expansion.**expand_internal**(*universe*, *graph*, *edge_predicates=None*)
Edges between entities in the sub-graph that pass the given filters.

Parameters

- **universe** (BELGraph) – The full graph
- **graph** (BELGraph) – A sub-graph to find the upstream information
- **edge_predicates** (Union[Callable[[BELGraph, BaseEntity, BaseEntity, str], bool], Iterable[Callable[[BELGraph, BaseEntity, BaseEntity, str], bool]], None]) – Optional list of edge filter functions (graph, node, node, key, data) -> bool

Return type None

pybel_tools.mutation.expansion.**expand_internal_causal**(*universe*, *graph*)
Add causal edges between entities in the sub-graph.

Is an extremely thin wrapper around *expand_internal()*.

Parameters

- **universe** (BELGraph) – A BEL graph representing the universe of all knowledge
- **graph** (BELGraph) – The target BEL graph to enrich with causal relations between contained nodes

Equivalent to:

```
>>> from pybel_tools.mutation import expand_internal
>>> from pybel.struct.filters.edge_predicates import is_causal_relation
>>> expand_internal(universe, graph, edge_predicates=is_causal_relation)
```

Return type None

2.15 Unbiased Candidate Mechanism Generation

An implementation of the unbiased candidate mechanism (UCM) generation workflow.

This workflow can be used to address the inconsistency in the definitions of the boundaries of pathways, mechanisms, sub-graphs, etc. in networks and systems biology that are introduced during curation due to a variety of reasons.

A simple approach for generating unbiased candidate mechanisms is to take the upstream controllers.

This module provides functions for generating sub-graphs based around a single node, most likely a biological process.

Sub-graphs induced around biological processes should prove to be sub-graphs of the NeuroMMSig/canonical mechanisms and provide an even more rich mechanism inventory.

This method has been applied in the following Jupyter Notebooks:

- Generating Unbiased Candidate Mechanisms

pybel_tools.generation.**remove_unweighted_leaves**(*graph*, *key=None*)
Remove nodes that are leaves and that don't have a weight (or other key) attribute set.

Parameters

- **graph** (BELGraph) – A BEL graph
- **key** (Optional[str]) – The key in the node data dictionary representing the experimental data. Defaults to pybel_tools.constants.WEIGHT.

Return type None

pybel_tools.generation.**is_unweighted_source**(*graph*, *node*, *key*)
Check if the node is both a source and also has an annotation.

Parameters

- **graph** (BELGraph) – A BEL graph
- **node** (BaseEntity) – A BEL node
- **key** (str) – The key in the node data dictionary representing the experimental data

Return type bool

pybel_tools.generation.**get_unweighted_sources**(*graph*, *key=None*)
Get nodes on the periphery of the sub-graph that do not have a annotation for the given key.

Parameters

- **graph** (BELGraph) – A BEL graph
- **key** (Optional[str]) – The key in the node data dictionary representing the experimental data

Return type Iterable[BaseEntity]

Returns An iterator over BEL nodes that are unannotated and on the periphery of this subgraph

pybel_tools.generation.**remove_unweighted_sources**(*graph*, *key=None*)
Prune unannotated nodes on the periphery of the sub-graph.

Parameters

- **graph** (BELGraph) – A BEL graph
- **key** (Optional[str]) – The key in the node data dictionary representing the experimental data. Defaults to pybel_tools.constants.WEIGHT.

Return type None

pybel_tools.generation.**prune_mechanism_by_data**(*graph*, *key=None*)
Remove all leaves and source nodes that don't have weights.

Is a thin wrapper around *remove_unweighted_leaves()* and *remove_unweighted_sources()*

Parameters

- **graph** – A BEL graph
- **key** (Optional[str]) – The key in the node data dictionary representing the experimental data. Defaults to pybel_tools.constants.WEIGHT.

Equivalent to:

```
>>> remove_unweighted_leaves(graph)
>>> remove_unweighted_sources(graph)
```

Return type None

pybel_tools.generation.**generate_mechanism**(*graph*, *node*, *key=None*)
Generate a mechanistic sub-graph upstream of the given node.

Parameters

- **graph** (BELGraph) – A BEL graph
- **node** (BaseEntity) – A BEL node
- **key** (Optional[str]) – The key in the node data dictionary representing the experimental data.

Return type BELGraph

Returns A sub-graph grown around the target BEL node

pybel_tools.generation.**generate_bioprocess_mechanisms**(*graph*, *key=None*)
Generate a mechanistic sub-graph for each biological process in the graph using *generate_mechanism()*.

Parameters

- **graph** – A BEL graph
- **key** (Optional[str]) – The key in the node data dictionary representing the experimental data.

Return type Mapping[BiologicalProcess, BELGraph]

2.16 Heat Diffusion Workflow

This module describes a heat diffusion workflow for analyzing BEL networks with differential gene expression[0].

It has four parts:

1) Assembling a network, pre-processing, and overlaying data
2) Generating unbiased candidate mechanisms from the network
3) Generating random sub-graphs from each unbiased candidate mechanism
4) Applying standard heat diffusion to each sub-graph and calculating scores for each unbiased candidate mechanism based on the distribution of scores for its sub-graph

In this algorithm, heat is applied to the nodes based on the data set. For the differential gene expression experiment, the log-fold-change values are used instead of the corrected p-values to allow for the effects of up- and down-regulation to be admitted in the analysis. Finally, heat diffusion inspired by previous algorithms published in systems and networks biology[1][2] is run with the constraint that decreases edges cause the sign of the heat to be flipped. Because of the construction of unbiased candidate mechanisms, all heat will flow towards their seed biological process nodes. The amount of heat on the biological process node after heat diffusion stops becomes the score for the whole unbiased candidate mechanism.

The issue of inconsistent causal networks addressed by SST[3] does not affect heat diffusion algorithms since it can quantify multiple conflicting pathways. However, it does not address the possibility of contradictory edges, for example, when `A increases B` and `A decreases B` are both true. A random sampling approach is used on networks with contradictory edges and aggregate statistics over multiple trials are used to assess the robustness of the scores as a function of the topology of the underlying unbiases candidate mechanisms.

2.16.1 Invariants

- Because heat always flows towards the biological process node, it is possible to remove leaf nodes (nodes with no incoming edges) after each step, since their heat will never change.

2.16.2 Examples

This workflow has been applied in several Jupyter notebooks:

- Heat Diffusion Workflow
- Time Series Heat Diffusion

2.16.3 Future Work

This algorithm can be tuned to allow the use of correlative relationships. Because many multi-scale and multi-modal data are often measured with correlations to molecular features, this enables experiments to be run using SNP or brain imaging features, whose experiments often measure their correlation with the activity of gene products.

[0] Hoyt, C. T., *et al.* (2017). PyBEL: a computational framework for Biological Expression Language. Bioinformatics (Oxford, England), 34(4), 703–704.

[1] Bernabo N., *et al.* (2014). The biological networks in studying cell signal transduction complexity: The examples of sperm capacitation and of endocannabinoid system. Computational and Structural Biotechnology Journal, 11 (18), 11–21.

[2] Leiserson, M. D. M., *et al.* (2015). Pan-cancer network analysis identifies combinations of rare somatic mutations across pathways and protein complexes. Nature Genetics, 47 (2), 106–14.

[3] Vasilyev, D. M., *et al.* (2014). An algorithm for score aggregation over causal biological networks based on random walk sampling. BMC Research Notes, 7, 516.

pybel_tools.analysis.heat.**RESULT_LABELS = ['avg', 'stddev', 'normality', 'median', 'neighbo**
The columns in the score tuples

pybel_tools.analysis.heat.**calculate_average_scores_on_graph**(*graph*, *key=None*, *tag=None*, *default_score=None*, *runs=None*, *use_tqdm=False*)

Calculate the scores over all biological processes in the sub-graph.

As an implementation, it simply computes the sub-graphs then calls *calculate_average_scores_on_subgraphs()* as described in that function's documentation.

Parameters

- **graph** (BELGraph) – A BEL graph with heats already on the nodes
- **key** (Optional[str]) – The key in the node data dictionary representing the experimental data. Defaults to pybel_tools.constants.WEIGHT.
- **tag** (Optional[str]) – The key for the nodes' data dictionaries where the scores will be put. Defaults to 'score'
- **default_score** (Optional[float]) – The initial score for all nodes. This number can go up or down.
- **runs** (Optional[int]) – The number of times to run the heat diffusion workflow. Defaults to 100.
- **use_tqdm** (bool) – Should there be a progress bar for runners?

Returns A dictionary of {pybel node tuple: results tuple}

Return type dict[tuple, tuple]

Suggested usage with pandas:

```
>>> import pandas as pd
>>> from pybel_tools.analysis.heat import calculate_average_scores_on_graph
>>> graph = ...  # load graph and data
>>> scores = calculate_average_scores_on_graph(graph)
>>> pd.DataFrame.from_items(scores.items(), orient='index', columns=RESULT_LABELS)
```

pybel_tools.analysis.heat.**calculate_average_scores_on_subgraphs**(*subgraphs*, *key=None*, *tag=None*, *default_score=None*, *runs=None*, *use_tqdm=False*, *tqdm_kwargs=None*)

Calculate the scores over precomputed candidate mechanisms.

Parameters

- **subgraphs** (Mapping[~H, BELGraph]) – A dictionary of keys to their corresponding subgraphs
- **key** (Optional[str]) – The key in the node data dictionary representing the experimental data. Defaults to pybel_tools.constants.WEIGHT.
- **tag** (Optional[str]) – The key for the nodes' data dictionaries where the scores will be put. Defaults to 'score'

- **default_score** (Optional[float]) – The initial score for all nodes. This number can go up or down.
- **runs** (Optional[int]) – The number of times to run the heat diffusion workflow. Defaults to 100.
- **use_tqdm** (bool) – Should there be a progress bar for runners?

Return type Mapping[~H, Tuple[float, float, float, float, int, int]]

Returns A dictionary of keys to results tuples

Example Usage:

```
>>> import pandas as pd
>>> from pybel_tools.generation import generate_bioprocess_mechanisms
>>> from pybel_tools.analysis.heat import calculate_average_scores_on_subgraphs
>>> # load graph and data
>>> graph = ...
>>> candidate_mechanisms = generate_bioprocess_mechanisms(graph)
>>> scores = calculate_average_scores_on_subgraphs(candidate_mechanisms)
>>> pd.DataFrame.from_items(scores.items(), orient='index', columns=RESULT_LABELS)
```

pybel_tools.analysis.heat.**workflow**(*graph*, *node*, *key=None*, *tag=None*, *default_score=None*, *runs=None*, *minimum_nodes=1*)

Generate candidate mechanisms and run the heat diffusion workflow.

Parameters

- **graph** (BELGraph) – A BEL graph
- **node** (BaseEntity) – The BEL node that is the focus of this analysis
- **key** (Optional[str]) – The key in the node data dictionary representing the experimental data. Defaults to pybel_tools.constants.WEIGHT.
- **tag** (Optional[str]) – The key for the nodes' data dictionaries where the scores will be put. Defaults to 'score'
- **default_score** (Optional[float]) – The initial score for all nodes. This number can go up or down.
- **runs** (Optional[int]) – The number of times to run the heat diffusion workflow. Defaults to 100.
- **minimum_nodes** (int) – The minimum number of nodes a sub-graph needs to try running heat diffusion

Return type List[*Runner*]

Returns A list of runners

pybel_tools.analysis.heat.**multirun**(*graph*, *node*, *key=None*, *tag=None*, *default_score=None*, *runs=None*, *use_tqdm=False*)

Run the heat diffusion workflow multiple times, each time yielding a *Runner* object upon completion.

Parameters

- **graph** (BELGraph) – A BEL graph
- **node** (BaseEntity) – The BEL node that is the focus of this analysis
- **key** (Optional[str]) – The key in the node data dictionary representing the experimental data. Defaults to pybel_tools.constants.WEIGHT.

- **tag** (Optional[str]) – The key for the nodes' data dictionaries where the scores will be put. Defaults to 'score'
- **default_score** (Optional[float]) – The initial score for all nodes. This number can go up or down.
- **runs** (Optional[int]) – The number of times to run the heat diffusion workflow. Defaults to 100.
- **use_tqdm** (bool) – Should there be a progress bar for runners?

Return type Iterable[*Runner*]

Returns An iterable over the runners after each iteration

class pybel_tools.analysis.heat.**Runner**(*graph*, *target_node*, *key=None*, *tag=None*, *default_score=None*)

This class houses the data related to a single run of the heat diffusion workflow.

Initialize the heat diffusion runner class.

Parameters

- **graph** (BELGraph) – A BEL graph
- **target_node** (BaseEntity) – The BEL node that is the focus of this analysis
- **key** (Optional[str]) – The key in the node data dictionary representing the experimental data. Defaults to pybel_tools.constants.WEIGHT.
- **tag** (Optional[str]) – The key for the nodes' data dictionaries where the scores will be put. Defaults to 'score'
- **default_score** (Optional[float]) – The initial score for all nodes. This number can go up or down.

iter_leaves()

Return an iterable over all nodes that are leaves.

A node is a leaf if either:

- it doesn't have any predecessors, OR
- all of its predecessors have a score in their data dictionaries

Return type Iterable[BaseEntity]

has_leaves()

Return if the current graph has any leaves.

Implementation is not that smart currently, and does a full sweep.

Return type List[BaseEntity]

in_out_ratio(*node*)

Calculate the ratio of in-degree / out-degree of a node.

Return type float

unscored_nodes_iter()

Iterate over all nodes without a score.

Return type BaseEntity

get_random_edge()
This function should be run when there are no leaves, but there are still unscored nodes. It will introduce a probabilistic element to the algorithm, where some edges are disregarded randomly to eventually get a score for the network. This means that the score can be averaged over many runs for a given graph, and a better data structure will have to be later developed that doesn't destroy the graph (instead, annotates which edges have been disregarded, later)

1. get all un-scored
2. rank by in-degree
3. weighted probability over all in-edges where lower in-degree means higher probability
4. pick randomly which edge

Returns A random in-edge to the lowest in/out degree ratio node. This is a 3-tuple of (node, node, key)

Return type tuple

remove_random_edge()
Remove a random in-edge from the node with the lowest in/out degree ratio.

remove_random_edge_until_has_leaves()
Remove random edges until there is at least one leaf node.

Return type None

score_leaves()
Calculate the score for all leaves.

Return type Set[BaseEntity]

Returns The set of leaf nodes that were scored

run()
Calculate scores for all leaves until there are none, removes edges until there are, and repeats until all nodes have been scored.

Return type None

run_with_graph_transformation()
Calculate scores for all leaves until there are none, removes edges until there are, and repeats until all nodes have been scored. Also, yields the current graph at every step so you can make a cool animation of how the graph changes throughout the course of the algorithm

Return type Iterable[BELGraph]

Returns An iterable of BEL graphs

done_chomping()
Determines if the algorithm is complete by checking if the target node of this analysis has been scored yet. Because the algorithm removes edges when it gets stuck until it is un-stuck, it is always guaranteed to finish.

Return type bool

Returns Is the algorithm done running?

get_final_score()
Return the final score for the target node.

Return type float

Returns The final score for the target node

calculate_score(*node*)
Calculate the new score of the given node.

Return type float

get_remaining_graph()
Allows for introspection on the algorithm at a given point by returning the sub-graph induced by all un-scored nodes

Return type BELGraph

Returns The remaining un-scored BEL graph

pybel_tools.analysis.heat.**workflow_aggregate**(*graph*, *node*, *key=None*, *tag=None*, *default_score=None*, *runs=None*, *aggregator=None*)
Get the average score over multiple runs.

This function is very simple, and can be copied to do more interesting statistics over the *Runner* instances. To iterate over the runners themselves, see *workflow()*

Parameters

- **graph** (BELGraph) – A BEL graph
- **node** (BaseEntity) – The BEL node that is the focus of this analysis
- **key** (Optional[str]) – The key in the node data dictionary representing the experimental data. Defaults to pybel_tools.constants.WEIGHT.
- **tag** (Optional[str]) – The key for the nodes' data dictionaries where the scores will be put. Defaults to 'score'
- **default_score** (Optional[float]) – The initial score for all nodes. This number can go up or down.
- **runs** (Optional[int]) – The number of times to run the heat diffusion workflow. Defaults to 100.
- **aggregator** (Optional[Callable[[Iterable[float]], float]]) – A function that aggregates a list of scores. Defaults to numpy.average(). Could also use: numpy.mean(), numpy.median(), numpy.min(), numpy.max()

Return type Optional[float]

Returns The average score for the target node

pybel_tools.analysis.heat.**workflow_all**(*graph*, *key=None*, *tag=None*, *default_score=None*, *runs=None*)
Run the heat diffusion workflow and get runners for every possible candidate mechanism

1. Get all biological processes
2. Get candidate mechanism induced two level back from each biological process
3. Heat diffusion workflow for each candidate mechanism for multiple runs
4. Return all runner results

Parameters

- **graph** (BELGraph) – A BEL graph

- **key** (Optional[str]) – The key in the node data dictionary representing the experimental data. Defaults to pybel_tools.constants.WEIGHT.
- **tag** (Optional[str]) – The key for the nodes' data dictionaries where the scores will be put. Defaults to 'score'
- **default_score** (Optional[float]) – The initial score for all nodes. This number can go up or down.
- **runs** (Optional[int]) – The number of times to run the heat diffusion workflow. Defaults to 100.

Return type Mapping[BaseEntity, List[*Runner*]]

Returns A dictionary of {node: list of runners}

pybel_tools.analysis.heat.**workflow_all_aggregate**(*graph*, *key=None*, *tag=None*, *default_score=None*, *runs=None*, *aggregator=None*)

Run the heat diffusion workflow to get average score for every possible candidate mechanism.

1. Get all biological processes
2. Get candidate mechanism induced two level back from each biological process
3. Heat diffusion workflow on each candidate mechanism for multiple runs
4. Report average scores for each candidate mechanism

Parameters

- **graph** (BELGraph) – A BEL graph
- **key** (Optional[str]) – The key in the node data dictionary representing the experimental data. Defaults to pybel_tools.constants.WEIGHT.
- **tag** (Optional[str]) – The key for the nodes' data dictionaries where the scores will be put. Defaults to 'score'
- **default_score** (Optional[float]) – The initial score for all nodes. This number can go up or down.
- **runs** (Optional[int]) – The number of times to run the heat diffusion workflow. Defaults to 100.
- **aggregator** (Optional[Callable[[Iterable[float]], float]]) – A function that aggregates a list of scores. Defaults to numpy.average(). Could also use: numpy.mean(), numpy.median(), numpy.min(), numpy.max()

Returns A dictionary of {node: upstream causal subgraph}

pybel_tools.analysis.heat.**calculate_average_score_by_annotation**(*graph*, *annotation*, *key=None*, *runs=None*, *use_tqdm=False*)

For each sub-graph induced over the edges matching the annotation, calculate the average score for all of the contained biological processes

Assumes you haven't done anything yet

1. Generates biological process upstream candidate mechanistic sub-graphs with generate_bioprocess_mechanisms()

2. Calculates scores for each sub-graph with `calculate_average_scores_on_sub-graphs()`
3. Overlays data with pbt.integration.overlay_data
4. Calculates averages with pbt.selection.group_nodes.average_node_annotation

Parameters

- **graph** (`BELGraph`) – A BEL graph
- **annotation** (`str`) – A BEL annotation
- **key** (`Optional[str]`) – The key in the node data dictionary representing the experimental data. Defaults to `pybel_tools.constants.WEIGHT`.
- **runs** (`Optional[int]`) – The number of times to run the heat diffusion workflow. Defaults to 100.
- **use_tqdm** (`bool`) – Should there be a progress bar for runners?

Return type `Mapping[str, float]`

Returns A dictionary from {str annotation value: tuple scores}

Example Usage:

```
>>> import pybel
>>> from pybel_tools.integration import overlay_data
>>> from pybel_tools.analysis.heat import calculate_average_score_by_annotation
>>> graph = pybel.from_path(...)
>>> scores = calculate_average_score_by_annotation(graph, 'subgraph')
```

2.17 HTML Assembler

Generate summary pages of BEL graphs in HTML.

`pybel_tools.assembler.html.`**`to_html`**(*graph*)

Render the graph as an HTML string.

Common usage may involve writing to a file like:

```
>>> from pybel.examples import sialic_acid_graph
>>> with open('html_output.html', 'w') as file:
...     print(to_html(sialic_acid_graph), file=file)
```

Return type `str`

2.18 Ideogram Assembler

Assemble a BEL graph as an ideogram chart in HTML..

`pybel_tools.assembler.ideogram.`**`to_html`**(*graph*, *chart=None*)

Render the graph as an HTML string.

Common usage may involve writing to a file like:

```
>>> from pybel.examples import sialic_acid_graph
>>> with open('ideogram_output.html', 'w') as file:
...     print(to_html(sialic_acid_graph), file=file)
```

Return type str

pybel_tools.assembler.ideogram.**to_jupyter**(*graph*, *chart=None*)

Render the graph as JavaScript in a Jupyter Notebook.

Return type Javascript

2.19 Document Utilities

2.19.1 Creating Definition Documents

pybel_tools.definition_utils.**get_merged_namespace_names**(*locations*, *check_keywords=True*)

Loads many namespaces and combines their names.

Parameters

- **locations** (*iter[str]*) – An iterable of URLs or file paths pointing to BEL namespaces.
- **check_keywords** (*bool*) – Should all the keywords be the same? Defaults to True

Returns A dictionary of {names: labels}

Return type dict[str, str]

Example Usage

```
>>> from pybel.resources import write_namespace
>>> from pybel_tools.definition_utils import export_namespace, get_merged_
↪namespace_names
>>> graph = ...
>>> original_ns_url = ...
>>> export_namespace(graph, 'MBS') # Outputs in current directory to MBS.belns
>>> value_dict = get_merged_namespace_names([original_ns_url, 'MBS.belns'])
>>> with open('merged_namespace.belns', 'w') as f:
>>> ...  write_namespace('MyBrokenNamespace', 'MBS', 'Other', 'Charles Hoyt',
↪'PyBEL Citation', value_dict, file=f)
```

pybel_tools.definition_utils.**merge_namespaces**(*input_locations*, *output_path*, *namespace_name*, *namespace_keyword*, *namespace_domain*, *author_name*, *citation_name*, *namespace_description=None*, *namespace_species=None*, *namespace_version=None*, *namespace_query_url=None*, *namespace_created=None*, *author_contact=None*, *author_copyright=None*, *citation_description=None*, *citation_url=None*, *citation_version=None*, *citation_date=None*, *case_sensitive=True*, *delimiter='|'*, *cacheable=True*, *functions=None*, *value_prefix=''*, *sort_key=None*, *check_keywords=True*)

Merges namespaces from multiple locations to one.

Parameters

- **input_locations** (*iter*) – An iterable of URLs or file paths pointing to BEL namespaces.
- **output_path** (*str*) – The path to the file to write the merged namespace
- **namespace_name** (*str*) – The namespace name
- **namespace_keyword** (*str*) – Preferred BEL Keyword, maximum length of 8
- **namespace_domain** (*str*) – One of: `pybel.constants.NAMESPACE_DOMAIN_BIOPROCESS`, `pybel.constants.NAMESPACE_DOMAIN_CHEMICAL`, `pybel.constants.NAMESPACE_DOMAIN_GENE`, or `pybel.constants.NAMESPACE_DOMAIN_OTHER`
- **author_name** (*str*) – The namespace's authors
- **citation_name** (*str*) – The name of the citation
- **namespace_query_url** (*str*) – HTTP URL to query for details on namespace values (must be valid URL)
- **namespace_description** (*str*) – Namespace description
- **namespace_species** (*str*) – Comma-separated list of species taxonomy id's
- **namespace_version** (*str*) – Namespace version
- **namespace_created** (*str*) – Namespace public timestamp, ISO 8601 datetime
- **author_contact** (*str*) – Namespace author's contact info/email address
- **author_copyright** (*str*) – Namespace's copyright/license information
- **citation_description** (*str*) – Citation description
- **citation_url** (*str*) – URL to more citation information
- **citation_version** (*str*) – Citation version
- **citation_date** (*str*) – Citation publish timestamp, ISO 8601 Date

- **case_sensitive** (*bool*) – Should this config file be interpreted as case-sensitive?
- **delimiter** (*str*) – The delimiter between names and labels in this config file
- **cacheable** (*bool*) – Should this config file be cached?
- **functions** (*iterable of characters*) – The encoding for the elements in this namespace
- **value_prefix** (*str*) – a prefix for each name
- **sort_key** – A function to sort the values with `sorted()`
- **check_keywords** (*bool*) – Should all the keywords be the same? Defaults to `True`

pybel_tools.definition_utils.**export_namespace**(*graph*, *namespace*, *directory=None*, *cacheable=False*)

Exports all names and missing names from the given namespace to its own BEL Namespace files in the given directory.

Could be useful during quick and dirty curation, where planned namespace building is not a priority.

Parameters

- **graph** (*pybel.BELGraph*) – A BEL graph
- **namespace** (*str*) – The namespace to process
- **directory** (*str*) – The path to the directory where to output the namespace. Defaults to the current working directory returned by `os.getcwd()`
- **cacheable** (*bool*) – Should the namespace be cacheable? Defaults to `False` because, in general, this operation will probably be used for evil, and users won't want to reload their entire cache after each iteration of curation.

pybel_tools.definition_utils.**export_namespaces**(*graph*, *namespaces*, *directory=None*, *cacheable=False*)

Thinly wraps *export_namespace()* for an iterable of namespaces.

Parameters

- **graph** (*pybel.BELGraph*) – A BEL graph
- **namespaces** (*iter[str]*) – An iterable of strings for the namespaces to process
- **directory** (*str*) – The path to the directory where to output the namespaces. Defaults to the current working directory returned by `os.getcwd()`
- **cacheable** (*bool*) – Should the namespaces be cacheable? Defaults to `False` because, in general, this operation will probably be used for evil, and users won't want to reload their entire cache after each iteration of curation.

2.19.2 Creating Knowledge Documents

pybel_tools.document_utils.**write_boilerplate**(*name*, *version=None*, *description=None*, *authors=None*, *contact=None*, *copyright=None*, *licenses=None*, *disclaimer=None*, *namespace_url=None*, *namespace_patterns=None*, *annotation_url=None*, *annotation_patterns=None*, *annotation_list=None*, *pmids=None*, *entrez_ids=None*, *file=None*)

Write a boilerplate BEL document, with standard document metadata, definitions.

Parameters

- **name** (str) – The unique name for this BEL document
- **contact** (Optional[str]) – The email address of the maintainer
- **description** (Optional[str]) – A description of the contents of this document
- **authors** (Optional[str]) – The authors of this document
- **version** (Optional[str]) – The version. Defaults to current date in format YYYYMMDD.
- **copyright** (Optional[str]) – Copyright information about this document
- **licenses** (Optional[str]) – The license applied to this document
- **disclaimer** (Optional[str]) – The disclaimer for this document
- **namespace_url** (Optional[Mapping[str, str]]) – an optional dictionary of {str name: str URL} of namespaces
- **namespace_patterns** (Optional[Mapping[str, str]]) – An optional dictionary of {str name: str regex} namespaces
- **annotation_url** (Optional[Mapping[str, str]]) – An optional dictionary of {str name: str URL} of annotations
- **annotation_patterns** (Optional[Mapping[str, str]]) – An optional dictionary of {str name: str regex} of regex annotations
- **annotation_list** (Optional[Mapping[str, Set[str]]]) – An optional dictionary of {str name: set of names} of list annotations
- **pmids** (Optional[Iterable[Union[str, int]]]) – A list of PubMed identifiers to auto-populate with citation and abstract
- **entrez_ids** (Optional[Iterable[Union[str, int]]]) – A list of Entrez identifiers to autopopulate the gene summary as evidence
- **file** (Optional[Textio]) – A writable file or file-like. If None, defaults to sys.stdout

Return type None

pybel_tools.document_utils.**lint_file**(*in_file*, *out_file=None*)

Helps remove extraneous whitespace from the lines of a file

Parameters

- **in_file** (*file*) – A readable file or file-like

- **out_file** (*file*) – A writable file or file-like

pybel_tools.document_utils.**lint_directory**(*source*, *target*)
Adds a linted version of each document in the source directory to the target directory

Parameters

- **source** (*str*) – Path to directory to lint
- **target** (*str*) – Path to directory to output

2.20 Utilities

This module contains functions useful throughout PyBEL Tools

pybel_tools.utils.**pairwise**(*iterable*)
Iterate over pairs in list s -> (s0,s1), (s1,s2), (s2, s3), ...

Return type Iterable[Tuple[~X, ~X]]

pybel_tools.utils.**count_defaultdict**(*dict_of_lists*)
Count the number of elements in each value of the dictionary.

Return type Mapping[~X, Counter[~Y]]

pybel_tools.utils.**count_dict_values**(*dict_of_counters*)
Count the number of elements in each value (can be list, Counter, etc).

Parameters **dict_of_counters** (Mapping[~X, Sized]) – A dictionary of things whose lengths can be measured (lists, Counters, dicts)

Return type Counter[~X]

Returns A Counter with the same keys as the input but the count of the length of the values list/tuple/set/Counter

pybel_tools.utils.**set_percentage**(*x*, *y*)
What percentage of x is contained within y?

Parameters

- **x** (*set*) – A set
- **y** (*set*) – Another set

Return type float

Returns The percentage of x contained within y

pybel_tools.utils.**tanimoto_set_similarity**(*x*, *y*)
Calculate the tanimoto set similarity.

Return type float

pybel_tools.utils.**min_tanimoto_set_similarity**(*x*, *y*)
Calculate the tanimoto set similarity using the minimum size.

Parameters

- **x** (*set*) – A set
- **y** (*set*) – Another set

Return type float

Returns The similarity between

pybel_tools.utils.**calculate_single_tanimoto_set_distances**(*target*, *dict_of_sets*)
Return a dictionary of distances keyed by the keys in the given dict.

Distances are calculated based on pairwise tanimoto similarity of the sets contained

Parameters

- **target** (*set*) – A set
- **dict_of_sets** (*dict*) – A dict of {x: set of y}

Returns A similarity dicationary based on the set overlap (tanimoto) score between the target set and the sets in dos

Return type dict

pybel_tools.utils.**calculate_tanimoto_set_distances**(*dict_of_sets*)
Return a distance matrix keyed by the keys in the given dict.

Distances are calculated based on pairwise tanimoto similarity of the sets contained.

Parameters **dict_of_sets** (Mapping[~X, Set[~T]]) – A dict of {x: set of y}

Return type Mapping[~X, Mapping[~X, float]]

Returns A similarity matrix based on the set overlap (tanimoto) score between each x as a dict of dicts

pybel_tools.utils.**calculate_global_tanimoto_set_distances**(*dict_of_sets*)
Calculate an alternative distance matrix based on the following equation.

$$distance(A, B) = 1 - \|A \cup B\| / \| \cup_{s \in S} s\|$$

Parameters **dict_of_sets** (Mapping[~X, Set[~T]]) – A dict of {x: set of y}

Return type Mapping[~X, Mapping[~X, float]]

Returns A similarity matrix based on the alternative tanimoto distance as a dict of dicts

pybel_tools.utils.**barh**(*d*, *plt*, *title=None*)
A convenience function for plotting a horizontal bar plot from a Counter

pybel_tools.utils.**barv**(*d*, *plt*, *title=None*, *rotation='vertical'*)
A convenience function for plotting a vertical bar plot from a Counter

pybel_tools.utils.**safe_add_edge**(*graph*, *u*, *v*, *key*, *attr_dict*, ***attr*)
Adds an edge while preserving negative keys, and paying no respect to positive ones

Parameters

- **graph** (*pybel.BELGraph*) – A BEL Graph
- **u** (*tuple*) – The source BEL node
- **v** (*tuple*) – The target BEL node
- **key** (*int*) – The edge key. If less than zero, corresponds to an unqualified edge, else is disregarded
- **attr_dict** (*dict*) – The edge data dictionary
- **attr** (*dict*) – Edge data to assign via keyword arguments

pybel_tools.utils.**prepare_c3**(*data*, *y_axis_label='y'*, *x_axis_label='x'*)
Prepares C3 JSON for making a bar chart from a Counter

Parameters

- **data** (Union[List[Tuple[str, int]], Mapping[str, int]]) – A dictionary of {str: int} to display as bar chart
- **y_axis_label** (str) – The Y axis label
- **x_axis_label** (str) – X axis internal label. Should be left as default 'x')

Return type str

Returns A JSON dictionary for making a C3 bar chart

pybel_tools.utils.**prepare_c3_time_series**(*data*, *y_axis_label='y'*, *x_axis_label='x'*)
Prepare C3 JSON string dump for a time series.

Parameters

- **data** (List[Tuple[int, int]]) – A list of tuples [(year, count)]
- **y_axis_label** (str) – The Y axis label
- **x_axis_label** (str) – X axis internal label. Should be left as default 'x')

Return type str

pybel_tools.utils.**calculate_betweenness_centality**(*graph*, *number_samples=200*)
Calculate the betweenness centrality over nodes in the graph.

Tries to do it with a certain number of samples, but then tries a complete approach if it fails.

Return type Counter

pybel_tools.utils.**get_circulations**(*elements*)
Iterate over all possible circulations of an ordered collection (tuple or list).

Example:

```
>>> list(get_circulations([1, 2, 3]))
[[1, 2, 3], [2, 3, 1], [3, 1, 2]]
```

Return type Iterable[~T]

pybel_tools.utils.**canonical_circulation**(*elements*, *key=None*)
Get get a canonical representation of the ordered collection by finding its minimum circulation with the given sort key

Return type ~T

pybel_tools.utils.**get_version**()
Get the current PyBEL Tools version.

Return type str

CHAPTER

THREE

INDICES AND TABLES

- genindex
- modindex
- search

BIBLIOGRAPHY

[Hoyt2017] Hoyt, C. T., *et al.* (2017). PyBEL: a Computational Framework for Biological Expression Language. Bioinformatics, 34(December), 1–2.

[Catlett2013] Catlett, N. L., *et al* (2013). Reverse causal reasoning: applying qualitative causal knowledge to the interpretation of high-throughput data. BMC Bioinformatics, 14(1), 340.

[Bradley2017] Bradley, G., & Barrett, S. J. (2017). CausalR - extracting mechanistic sense from genome scale data. Bioinformatics, (June), 1–3.

[Tarca2009] Tarca, A. L., *et al* (2009). A novel signaling pathway impact analysis. Bioinformatics, 25(1), 75–82.

[DomingoFernandez2017] Domingo-Fernández, D., *et al* (2017). Multimodal mechanistic signatures for neurodegenerative diseases (NeuroMMSig): A web server for mechanism enrichment. Bioinformatics, 33(22), 3679–3681.

[Hoyt2018] Hoyt, C. T., *et al*. (2018) A systematic approach for identifying shared mechanisms in epilepsy and its comorbidities, Database, Volume 2018, 1 January 2018, bay050

PYTHON MODULE INDEX

p

INDEX

Symbols

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

R

S

T

U

V

W