graphid Documentation

Release 0.1.0

Jon Crall

Aug 30, 2023

CONTENTS

CHAPTER

ONE

GRAPHID PACKAGE

1.1 Subpackages

1.1.1 graphid.core package

1.1.1.1 Submodules

1.1.1.1.1 graphid.core.__main__ module

1.1.1.1.2 graphid.core._rhomb_dist module

This is very ibeis-specific, and likely does not belong here.

graphid.core._rhomb_dist.**RhombicuboctahedronDistanceDemo**()

CommandLine

```
python -m graphid.core._rhomb_dist RhombicuboctahedronDistanceDemo --show
```

Returns
face

Return type
?

CommandLine

```
python -m graphid.core._rhomb_dist RhombicuboctahedronDistanceDemo --show
```

Example

```
>>> # xdoctest: +REQUIRES(module:pygraphviz)
>>> from graphid import util
>>> RhombicuboctahedronDistanceDemo()
>>> util.show_if_requested()
```

1.1.1.1.3 graphid.core.annot_inference module

graphid.core.annot_inference.**_rectify_decision**(*evidence_decision*, *meta_decision*)

If evidence decision is not explicitly set, then meta decision is used to make a guess. Raises a ValueError if decisions are in incompatible states.

class graphid.core.annot_inference.**Consistency**

Bases: object

is_consistent(*cc*)

Determines if a PCC contains inconsistencies

Parameters
cc (*set*) – nodes in a PCC

Returns
bool: returns True unless cc contains any negative edges

Return type
flag

Example

```
>>> from graphid import demo
>>> infr = demo.demodata_infr(num_pccs=1, p_incon=1)
>>> assert not infr.is_consistent(next(infr.positive_components()))
>>> infr = demo.demodata_infr(num_pccs=1, p_incon=0)
>>> assert infr.is_consistent(next(infr.positive_components()))
```

positive_components(*graph=None*)

Generates the positive connected compoments (PCCs) in the graph These will contain both consistent and inconsinstent PCCs.

Yields
cc – set: nodes within the PCC

inconsistent_components(*graph=None*)

Generates inconsistent PCCs. These PCCs contain internal negative edges indicating an error exists.

consistent_components(*graph=None*)

Generates consistent PCCs. These PCCs contain no internal negative edges.

Yields
cc – set: nodes within the PCC

class graphid.core.annot_inference.**Feedback**

Bases: object

_check_edge(*edge*)

add_feedback_from(*items*, *verbose=None*, ***kwargs*)

Parameters
items (*List[Edge]*) – each edge is a dictionary with aid1, aid2, evidence_decision, meta_decision, etc..

edge_decision(*edge*)

Gets a decision on an edge, either explicitly or implicitly

edge_decision_from(*edges*)

Gets a decision for multiple edges

add_node_feedback(*aid*, ***attrs*)

add_feedback(*edge*, *evidence_decision=None*, *tags=None*, *user_id=None*, *meta_decision=None*, *confidence=None*, *timestamp_c1=None*, *timestamp_c2=None*, *timestamp_s1=None*, *timestamp=None*, *verbose=None*, *priority=None*)

Primary method for adding feedback and review edges to the graph.

Parameters

- **edge** (*tuple*) – an undirected edge represented as a pair of aids
- **evidence_decision** (*str*) – decision made based on visual evidence between the two photos. Can be POSTV, NEGTV, INCMP, or UNKWN. Note: POSTV etc… are the variables not the strings.
- **tags** (*list of str*) – additional information to specify
- **user_id** (*str*) – who is doing this review. This can identify a human or algorithm reviewer (e.g. 'user:joncrall' or 'algo:vamp').
- **meta_decision** (*str*) – decision made based on external knowledge. Perhaps the photographer knows that two animals are the same because all photos are of the same animal. This constrains the identity problem, but does not impact the computer vision learning algorithms, which aren't given the info needed to make this sort of decision.
- **confidence** (*str*) – how sure is the user of this decision.
- **timestamp_c1** (*int*) – Time that the review client started
- **timestamp_c2** (*int*) – Time that the review client ended
- **timestamp_s1** (*int*) – Time that the review server started
- **timestamp** (*int*) – Time that the review server ended
- **verbose** (*bool*) – verbosity
- **priority** (*float, optional*) – the priority assigned to this edge before review. This is only relevant for the termination criterion.

Notes

If *infr.params['inference.enabled']* is True, then the edge is inserted into the graph and its properties are updated dynamically. Otherwise it is only added to the internal feedback dictionary and the *apply_feedback_edges* method must be called.

Example

```
>>> from graphid import demo
>>> infr = demo.demodata_infr(num_pccs=5)
>>> infr.add_feedback((5, 6), POSTV)
>>> infr.add_feedback((5, 6), NEGTV, tags=['photobomb'])
>>> infr.add_feedback((1, 2), INCMP)
>>> print(ub.urepr(infr.internal_feedback, nl=3, sk=1))
>>> assert len(infr.external_feedback) == 0
>>> assert len(infr.internal_feedback) == 2
>>> assert len(infr.internal_feedback[(5, 6)]) == 2
>>> assert len(infr.internal_feedback[(1, 2)]) == 1
```

_print_debug_ccs()

feedback_keys = ['evidence_decision', 'tags', 'user_id', 'meta_decision', 'timestamp_c1', 'timestamp_c2', 'timestamp_s1', 'timestamp', 'confidence', 'num_reviews', 'review_id']

feedback_data_keys = ['evidence_decision', 'tags', 'user_id', 'meta_decision', 'timestamp_c1', 'timestamp_c2', 'timestamp_s1', 'timestamp', 'confidence']

apply_feedback_edges()

Transforms the feedback dictionaries into nx graph edge attributes. This

_rectify_feedback(*feedback*)

_rectify_feedback_item(*vals*)

uses most recently use strategy

all_feedback_items()

all_feedback()

clear_feedback(*edges=None*)

Delete all edges properties related to feedback

clear_edges()

Removes all edges from the graph

reset(*state='empty'*)

Removes all edges from graph and resets name labels.

Example

```
>>> from graphid.core.annot_inference import *  # NOQA
>>> from graphid import demo
>>> infr = demo.demodata_infr(num_pccs=5)
>>> assert len(list(infr.edges())) > 0
>>> infr.reset(state='empty')
>>> assert len(list(infr.edges())) == 0
```

reset_name_labels()

Resets all annotation node name labels to their initial values

clear_name_labels()

Sets all annotation node name labels to be unknown

class graphid.core.annot_inference.**NameRelabel**

Bases: object

node_label(*aid*)

node_labels(**aids*)

_next_nid()

_rectify_names(*old_names*, *new_labels*)

Finds the best assignment of old names based on the new groups each is assigned to.

old_names = [None, None, None, 1, 2, 3, 3, 4, 4, 4, 5, None] new_labels = [1, 2, 2, 3, 4, 5, 5, 6, 3, 3, 7, 7]

_rectified_relabel(*cc_subgraphs*)

Reuses as many names as possible

relabel_using_reviews(*graph=None*, *rectify=True*)

Relabels nodes in graph based on positive connected components

This will change the 'name_label' of the nodes to be consistent while preserving any existing names as best as possible. If rectify=False, this will be faster, but the old names may not be preserved and each PCC will be assigned an arbitrary name.

Note: if something messes up you can call infr.reset_labels_to_ibeis() to reset node labels to their original values — this will almost always put the graph in an inconsistent state — but then you can this with rectify=True to fix everything up.

Parameters

- **graph** (*nx.Graph, optional*) – only edges in *graph* are relabeled defaults to current graph.
- **rectify** (*bool, optional*) – if True names attempt to remain consistent otherwise there are no restrictions on name labels other than that they are distinct.

Example

```
>>> from graphid import demo, util
>>> infr = demo.demodata_infr(num_pccs=5, pos_redun=1)
>>> names0 = set(infr.get_node_attrs('name_label').values())
>>> infr.relabel_using_reviews(rectify=True)
>>> names1 = set(infr.get_node_attrs('name_label').values())
>>> assert names0 == names1
>>> # wont change because its the entire graph
>>> #infr.relabel_using_reviews(rectify=False)
>>> #names2 = set(infr.get_node_attrs('name_label').values())
```

class graphid.core.annot_inference.**MiscHelpers**

Bases: object

_rectify_nids(*aids*, *nids*)

remove_aids(*aids*)

Remove annotations from the graph.

Returns
split: indicates which PCCs were split by this action.

Return type
dict

Note: This may cause unintended splits!

CommandLine

```
xdoctest -m graphid.core.annot_inference MiscHelpers.remove_aids
```

Example

```
>>> from graphid import demo, util
>>> infr = demo.demodata_infr(num_pccs=5, pos_redun=1)
>>> infr.refresh_candidate_edges()
>>> infr.pin_node_layout()
>>> before = infr.copy()
>>> aids = infr.aids[::5]
>>> splits = infr.remove_aids(aids)
>>> assert len(splits['old']) > 0
>>> infr.assert_invariants()
>>> # xdoc: +REQUIRES(--show)
>>> util.qtensure()
>>> after = infr
>>> before.show(fnum=1, pnum=(1, 2, 1), pickable=True)
>>> after.show(fnum=1, pnum=(1, 2, 2), pickable=True)
```

add_aids(*aids*, *nids=None*)

CommandLine

```
python -m graphid.core.annot_inference MiscHelpers.add_aids
```

Doctest

```
>>> aids_ = [1, 2, 3, 4, 5, 6, 7, 9]
>>> infr = AnnotInference(aids=aids_, autoinit=True)
>>> aids = [2, 22, 7, 9, 8]
>>> nids = None
>>> infr.add_aids(aids, nids)
>>> result = infr.aids
>>> print(result)
>>> assert len(infr.graph) == len(infr.aids)
[1, 2, 3, 4, 5, 6, 7, 9, 22, 8]
```

update_node_attributes(*aids=None*, *nids=None*)

initialize_graph(*graph=None*)

Constructs the internal networkx Graph objects

print(*msg*, *level=1*, *color=None*)

latest_logs(*colored=False*)

dump_logs()

class graphid.core.annot_inference.**AltConstructors**

Bases: object

_graph_cls

alias of *NiceGraph*

classmethod from_pairs(*aid_pairs*, *attrs=None*, *verbose=False*)

classmethod from_netx(*G*, *verbose=False*, *infer=True*)

Creates an AnnotInference object from a networkx graph

status(*extended=False*)

Returns information about the state of the graph.

Parameters

extended (*bool*) – if True, adds in extra information that requires an O(|E|) amount of computation, otherwise only O(1) stats that are dynamically tracked are returned.

Returns

a dictionary containing status information. Each of the keys

represents the following information:

nNodes: number of nodes in the graph nEdges: number of edges in the graph nCCs: number of positive connected components nPostvEdges: number of edges labeled as positive nNegtvEdges: number of edges labeled as negative nIncmpEdges: number of edges labeled as incomparable nUnrevEdges: number of edges labeled as unreviewed nPosRedunCCs: the number of PCCs which are currently

k-positive-redundant, i.e. we are confident those PCCs are the same individual.

nNegRedunPairs: the number of PCCs pairs which are
currently k-negative-redundant, i.e. we are confident those PCCs are different individuals.

nInconsistentCCs: the number of inconsistent PCCs that need
to be fixed, i.e. the number of PCCs with an internal negative edges.

If extended is True, then the following keys are also present
nNegEdgesWithin: number of negatives edges inside PCCs nNegEdgesBetween: number of negative edges between PCCs nIncompEdgesWithin: number of incomparable edges inside PCCs nIncompEdgesBetween: number of incomparable edges between PCCs nUnrevEdgesWithin: number of unreviewed edges inside PCCs nUrevEdgesBetween: number of unreviewed edges between PCCs

Return type
dict

Example

```
>>> from graphid import demo
>>> infr = demo.demodata_infr(num_pccs=5, p_incon=0.5, pcc_size=10)
>>> print(ub.urepr(infr.status(extended=True)))
{
    'nNodes': 50,
    'nEdges': 93,
    'nCCs': 5,
    'nPostvEdges': 66,
    'nNegtvEdges': 10,
    'nIncmpEdges': 2,
    'nUnrevEdges': 15,
    'nPosRedunCCs': 1,
    'nNegRedunPairs': 2,
    'nInconsistentCCs': 3,
    'nNegEdgesWithin': 4,
    'nNegEdgesBetween': 6,
    'nIncompEdgesWithin': 0,
    'nIncompEdgesBetween': 2,
    'nUnrevEdgesWithin': 15,
    'nUrevEdgesBetween': 0,
}
```

class graphid.core.annot_inference.**AnnotInference**(*aids=[]*, *nids=None*, *autoinit=True*, *verbose=False*)

Bases: NiceRepr, *AltConstructors*, *MiscHelpers*, *Feedback*, *NameRelabel*, *Consistency*, *NonDynamicUpdate*, *Recovery*, *DynamicUpdate*, *Redundancy*, *Priority*, *AssertInvariants*, *DummyEdges*, *Convenience*, *AttrAccess*, *SimulationHelpers*, *InfrReviewers*, *InfrLoops*, *InfrCallbacks*, *InfrCandidates*, *GraphVisualization*

class for maintaining state of an identification

CommandLine

```
python -m graphid.core.annot_inference AnnotInference
python -m graphid.core.annot_inference AnnotInference --show
```

Example

```
>>> from graphid.core import AnnotInference
>>> import pytest
>>> infr = AnnotInference()
>>> print('infr = {}'.format(infr))
infr = <AnnotInference(nNodes=0, nEdges=0, nCCs=0)>
>>> infr.add_aids(list(range(1, 6)))
>>> print('infr = {}'.format(infr))
infr = <AnnotInference(nNodes=5, nEdges=0, nCCs=5)>
>>> # Add some feedback
>>> infr.params['allow_unseen_nodes'] = False
>>> infr.add_feedback((1, 2), POSTV)
>>> infr.add_feedback((1, 3), INCMP)
>>> infr.add_feedback((1, 4), NEGTV)
>>> with pytest.raises(ValueError):
>>>     infr.add_feedback((1, 10), NEGTV)
>>> with pytest.raises(ValueError):
>>>     infr.add_feedback((11, 12), NEGTV)
>>> print('infr = {}'.format(infr))
infr = <AnnotInference(nNodes=5, nEdges=3, nCCs=4)>
>>> # xdoc: +REQUIRES(--show)
>>> infr.show_graph()
>>> util.show_if_requested()
```

subparams(*prefix*)

Returns dict of params prefixed with <prefix>. The returned dict does not contain the prefix

Example

```
>>> infr = AnnotInference()
>>> result = ub.urepr(infr.subparams('refresh'), nl=0, precision=1, sort=1)
>>> print(result)
{'method': 'binomial', 'patience': 72, 'thresh': 0.1, 'window': 20}
```

copy()

subgraph(*aids*)

Makes a new inference object that is a subset of the original.

Note, this is not robust, be careful. The subgraph should be treated as read only. Do not commit any reviews made from here.

set_config(*config*, ***kw*)

1.1.1.1.4 graphid.core.mixin_callbacks module

class graphid.core.mixin_callbacks.**InfrCallbacks**

Bases: object

Methods relating to callbacks that must be registered with the inference object for it to work properly.

set_ranker(*ranker*)

ranker should be a function that accepts a list of annotation ids and return a list of the top K ranked annotations.

set_verifier(*verifier*, *task='match_state'*)

verifier should be a function that accepts a list of annotation pairs and produces the 3-state match_state probabilities.

refresh_candidate_edges()

CommandLine

```
python -m graphid.core.mixin_callbacks InfrCallbacks.refresh_candidate_edges
```

Example

```
>>> from graphid import demo
>>> kwargs = dict(num_pccs=40, size=2)
>>> infr = demo.demodata_infr(**kwargs)
>>> infr.refresh_candidate_edges()
```

class graphid.core.mixin_callbacks.**InfrCandidates**

Bases: object

Methods that should be used by callbacks to add new edges to be considered as candidates in the priority queue.

add_candidate_edges(*candidate_edges*)

ensure_task_probs(*edges*)

Ensures that probabilities are assigned to the edges. This gaurentees that infr.task_probs contains data for edges. (Currently only the primary task is actually ensured)

CommandLine

```
python -m graphid.core.mixin_callbacks InfrCandidates.ensure_task_probs
```

Doctest

```
>>> from graphid import demo
>>> infr = demo.demodata_infr(num_pccs=6, p_incon=.5, size_std=2)
>>> edges = list(infr.edges())
>>> infr.ensure_task_probs(edges)
>>> assert all([np.isclose(sum(p.values()), 1)
>>>             for p in infr.task_probs['match_state'].values()])
```

ensure_priority_scores(*priority_edges*)

Ensures that priority attributes are assigned to the edges. This does not change the state of the queue.

Doctest

```
>>> from graphid import demo
>>> infr = demo.demodata_infr(num_pccs=6, p_incon=.5, size_std=2)
>>> edges = list(infr.edges())
>>> infr.ensure_priority_scores(edges)
```

1.1.1.1.5 graphid.core.mixin_dynamic module

This file handles dynamically updating the graph state based on new feedback. This involves handling lots of different cases, which can get confusing (it confuses me and I wrote it). To better understand the dynamic case a good first step would be to understand the nondynamic case defined by *apply_nondynamic_update*. This function automatically puts the graph into a state that satisfies the dynamic invariants. Any dynamic operation followed by a call to this function should be a no-op, which you can used to check if a dynamic operation is implemented correctly.

Todo: Negative bookkeeping, needs a small re-organization fix. MOVE FROM neg_redun_metagraph TO neg_metagraph

Instead of maintaining a graph that contains PCCS which are neg redundant to each other, the graph should maintain PCCs that have ANY negative edge between them (aka 1 neg redundant). Then that edge should store a flag indicating the strength / redundancy of that connection. A better idea might be to store both neg_redun_metagraph AND neg_metagraph.

TODO: this (all neg-redun functionality can be easilly consolidated into the neg-metagraph-update. note, we have to allow inconsistent pccs to be in the neg redun graph, we just filter them out afterwords)

---

class graphid.core.mixin_dynamic.**DynamicUpdate**

Bases: `object`

12 total possible states

details of these states. POSITIVE, WITHIN, CONSISTENT

- pos-within never changes PCC status
- never introduces inconsistency
- might add pos-redun

POSITIVE, WITHIN, INCONSISTENT

- pos-within never changes PCC status

- might fix inconsistent edge

POSITIVE, BETWEEN, BOTH_CONSISTENT

- pos-between edge always does merge

POSITIVE, BETWEEN, ANY_INCONSISTENT

- pos-between edge always does merge
- pos-between never fixes inconsistency

NEGATIVE, WITHIN, CONSISTENT

- might split PCC, results will be consistent
- might causes an inconsistency

NEGATIVE, WITHIN, INCONSISTENT

- might split PCC, results may be inconsistent

NEGATIVE, BETWEEN, BOTH_CONSISTENT

- might add neg-redun

NEGATIVE, BETWEEN, ANY_INCONSISTENT

- might add to incon-neg-external
- neg-redun not tracked for incon.

UNINFERABLE, WITHIN, CONSISTENT

- might remove pos-redun
- might split PCC, results will be consistent

UNINFERABLE, WITHIN, INCONSISTENT

- might split PCC, results may be inconsistent

UNINFERABLE, BETWEEN, BOTH_CONSISTENT

- might remove neg-redun

UNINFERABLE, BETWEEN, ANY_INCONSISTENT

- might remove incon-neg-external

`ensure_edges_from`(*edges*)

Finds edges that don't exist and adds them as unreviwed edges. Returns new edges that were added.

`_add_review_edges_from`(*edges*, *decision='UNREV'*)

`_add_review_edge`(*edge*, *decision*)

Adds an edge to the appropriate data structure

`_get_current_decision`(*edge*)

Find if any data structure has the edge

`on_between`(*edge*, *decision*, *prev_decision*, *nid1*, *nid2*, *merge_nid=None*)

Callback when a review is made between two PCCs

on_within(*edge*, *decision*, *prev_decision*, *nid*, *split_nids=None*)

Callback when a review is made inside a PCC

Parameters

- **edge** – the edge reviewed
- **decision** – the new decision
- **prev_decision** – the old decision
- **nid** – the old nid the edge is inside of
- **split_nids** – the tuple of new nids created if this decision splits a PCC

_update_neg_metagraph(*decision*, *prev_decision*, *nid1*, *nid2*, *merge_nid=None*, *split_nids=None*)

Update the negative metagraph based a new review

Todo: we can likely consolidate lots of neg_redun_metagraph functionality into this function. Just check when the weights are above or under the threshold and update accordingly.

_positive_decision(*edge*)

Logic for a dynamic positive decision. A positive decision is evidence that two annots should be in the same PCC

Note, this could be an incomparable edge, but with a meta_decision of same.

_negative_decision(*edge*)

Logic for a dynamic negative decision. A negative decision is evidence that two annots should not be in the same PCC

_uninferable_decision(*edge*, *decision*)

Logic for a dynamic uninferable negative decision An uninferrable decision does not provide any evidence about PCC status and is either:

incomparable, unreviewed, or unknown

class graphid.core.mixin_dynamic.**Recovery**

Bases: object

recovery funcs

is_recovering(*edge=None*)

Checks to see if the graph is inconsinsistent.

Parameters

edge (*None*) – If None, then returns True if the graph contains any inconsistency. Otherwise, returns True if the edge is related to an inconsistent component via a positive or negative connection.

Returns

flag

Return type

bool

CommandLine

```
python -m graphid.core.mixin_dynamic Recovery.is_recovering
```

Doctest

```
>>> from graphid import demo
>>> infr = demo.demodata_infr(num_pccs=4, size=4, ignore_pair=True)
>>> infr.ensure_cliques(meta_decision=SAME)
>>> a, b, c, d = map(list, infr.positive_components())
>>> assert infr.is_recovering() is False
>>> infr.add_feedback((a[0], a[1]), NEGTV)
>>> assert infr.is_recovering() is True
>>> assert infr.is_recovering((a[2], a[3])) is True
>>> assert infr.is_recovering((a[3], b[0])) is True
>>> assert infr.is_recovering((b[0], b[1])) is False
>>> infr.add_feedback((a[3], b[2]), NEGTV)
>>> assert infr.is_recovering((b[0], b[1])) is True
>>> assert infr.is_recovering((c[0], d[0])) is False
>>> infr.add_feedback((b[2], c[0]), NEGTV)
>>> assert infr.is_recovering((c[0], d[0])) is False
>>> result = ub.urepr({
>>>     'iccs': list(infr.inconsistent_components()),
>>>     'pccs': sorted([cc for cc in infr.positive_components()], key=min),
>>> }, nobr=1, sorted=True, si=True, itemsep='', sep='', nl=1)
>>> print(result)
iccs: [{1,2,3,4}],
pccs: [{1,2,3,4},{5,6,7,8},{9,10,11,12},{13,14,15,16}],
```

_purge_error_edges(*nid*)

Removes all error edges associated with a PCC so they can be recomputed or resolved.

_set_error_edges(*nid*, *new_error_edges*)

maybe_error_edges()

_new_inconsistency(*nid*)

_check_inconsistency(*nid*, *cc=None*)

Check if a PCC contains an error

_mincut_edge_weights(*edges_*)

hypothesis_errors(*pos_subgraph*, *neg_edges*)

class graphid.core.mixin_dynamic.**NonDynamicUpdate**

Bases: object

apply_nondynamic_update(*graph=None*)

Recomputes all dynamic bookkeeping for a graph in any state. This ensures that subsequent dyanmic inference can be applied.

Example

```
>>> from graphid import demo
>>> num_pccs = 250
>>> kwargs = dict(num_pccs=100, p_incon=.3)
>>> infr = demo.demodata_infr(infer=False, **kwargs)
>>> graph = None
>>> infr.apply_nondynamic_update()
>>> infr.assert_neg_metagraph()
```

collapsed_meta_edges(*graph=None*)

Collapse the grah such that each PCC is a node. Get a list of edges within/between each PCC.

categorize_edges(*graph=None*, *ne_to_edges=None*)

Non-dynamically computes the status of each edge in the graph. This is can be used to verify the dynamic computations and update when the dynamic state is lost.

Example

```
>>> from graphid import demo
>>> num_pccs = 250 if ub.argflag('--profile') else 100
>>> kwargs = dict(num_pccs=100, p_incon=.3)
>>> infr = demo.demodata_infr(infer=False, **kwargs)
>>> graph = None
>>> cat = infr.categorize_edges()
```

1.1.1.1.6 graphid.core.mixin_helpers module

class graphid.core.mixin_helpers.**AttrAccess**

Bases: object

Contains non-core helper functions

gen_node_attrs(*key*, *nodes=None*, *default=NoParam*)

gen_edge_attrs(*key*, *edges=None*, *default=NoParam*, *on_missing=None*)

maybe change to gen edge items

gen_node_values(*key*, *nodes*, *default=NoParam*)

gen_edge_values(*key*, *edges=None*, *default=NoParam*, *on_missing='error'*, *on_keyerr='default'*)

get_node_attrs(*key*, *nodes=None*, *default=NoParam*)

Networkx node getter helper

get_edge_attrs(*key*, *edges=None*, *default=NoParam*, *on_missing=None*)

Networkx edge getter helper

_get_edges_where(*key*, *op*, *val*, *edges=None*, *default=NoParam*, *on_missing=None*)

get_edges_where_eq(*key*, *val*, *edges=None*, *default=NoParam*, *on_missing=None*)

get_edges_where_ne(*key*, *val*, *edges=None*, *default=NoParam*, *on_missing=None*)

set_node_attrs(*key*, *node_to_prop*)

Networkx node setter helper

set_edge_attrs(*key*, *edge_to_prop*)

Networkx edge setter helper

get_edge_attr(*edge*, *key*, *default=NoParam*, *on_missing='error'*)

single edge getter helper

set_edge_attr(*edge*, *attr*)

single edge setter helper

get_annot_attrs(*key*, *aids*)

Wrapper around get_node_attrs specific to annotation nodes

edges(*data=False*)

has_edge(*edge*)

get_edge_data(*edge*)

get_nonvisual_edge_data(*edge*, *on_missing='filter'*)

get_edge_dataframe(*edges=None*, *all=False*)

get_edge_df_text(*edges=None*, *highlight=True*)

class graphid.core.mixin_helpers.**Convenience**

Bases: object

static e_(*u*, *v*)

property pos_graph

property neg_graph

property incomp_graph

property unreviewed_graph

property unknown_graph

print_graph_info()

print_graph_connections(*label='orig_name_label'*)

label = 'orig_name_label'

print_within_connection_info(*edge=None*, *cc=None*, *aid=None*, *nid=None*)

pair_connection_info(*aid1*, *aid2*)

Helps debugging when ibs.nids has info that annotmatch/staging do not Note: the relevant ibs parts were removed. Perhaps this is not useful now or should be moved to the ibeis plugin?

Example

```
>>> from graphid import demo
>>> infr = demo.demodata_infr(num_pccs=3, size=4)
>>> aid1, aid2 = 1, 2
>>> print(infr.pair_connection_info(aid1, aid2))
```

node_tag_hist()

edge_tag_hist()

match_state_df(*index*)

Returns the current matching state of a list of edges.

PERHAPS WE SHOULD DEPRICATE THIS FUNCTION?

Note: This does NOT use the IBEIS database state, where as the original version of this function did.

CommandLine

```
python -m graphid.core.mixin_helpers Convenience.match_state_df
```

Example

```
>>> from graphid import demo
>>> infr = demo.demodata_infr(num_pccs=2, p_incomp=.8, size=4)
>>> index = list(infr.edges())
>>> print(infr.match_state_df(index))
           NEGTV  POSTV  INCMP
aid1 aid2
1    3     False  False   True
     4     False  False   True
     2     False   True  False
2    3     False  False   True
     4     False  False   True
3    4     False   True  False
     5     False  False   True
5    8     False  False   True
     7     False  False   True
     6     False  False   True
6    8     False  False   True
     7     False  False   True
7    8     False  False   True
```

class graphid.core.mixin_helpers.**DummyEdges**

Bases: object

ensure_mst(*label='name_label'*, *meta_decision='same'*)

Ensures that all names are names are connected.

Parameters

- **label** (*str*) – node attribute to use as the group id to form the mst.
- **meta_decision** (*str*) – if specified adds clique edges as feedback items with this decision. Otherwise the edges are only explicitly added to the graph. This makes feedback items with user_id=algo:mst and with a confidence of guessing.

Example

```
>>> from graphid import demo
>>> infr = demo.demodata_infr(num_pccs=3, size=4)
>>> assert infr.status()['nCCs'] == 3
>>> infr.clear_edges()
>>> assert infr.status()['nCCs'] == 12
>>> infr.ensure_mst()
>>> assert infr.status()['nCCs'] == 3
```

ensure_cliques(*label='name_label'*, *meta_decision=None*)

Force each name label to be a clique.

Parameters

- **label** (*str*) – node attribute to use as the group id to form the cliques.
- **meta_decision** (*str*) – if specified adds clique edges as feedback items with this decision. Otherwise the edges are only explicitly added to the graph.

Parameters

- **label** (*str*) – defaults to ‘name_label’
- **meta_decision** (*str*) – if specified, the feedback edges added are added this meta decision and with the *user_id=algo:clique*.

CommandLine

```
python -m graphid.core.mixin_helpers ensure_cliques
```

Example

```
>>> from graphid import demo
>>> label = 'name_label'
>>> infr = demo.demodata_infr(num_pccs=3, size=5)
>>> print(ub.urepr(infr.status()))
>>> assert infr.status()['nEdges'] < 33
>>> infr.ensure_cliques()
>>> print(ub.urepr(infr.status()))
>>> assert infr.status()['nEdges'] == 31
>>> assert infr.status()['nUnrevEdges'] == 12
>>> assert len(list(infr.find_clique_edges(label))) > 0
>>> infr.ensure_cliques(meta_decision=SAME)
>>> assert infr.status()['nUnrevEdges'] == 0
>>> assert len(list(infr.find_clique_edges(label))) == 0
```

ensure_full()

Explicitly places all edges, but does not make any feedback items

find_clique_edges(*label='name_label'*)

Augmenting edges that would complete each the specified cliques. (based on the group inferred from *label*)

Parameters
label (*str*) – node attribute to use as the group id to form the cliques.

find_mst_edges(*label='name_label'*)

Returns edges to augment existing PCCs (by label) in order to ensure they are connected with positive edges.

Example

```
>>> # DISABLE_DOCTEST
>>> from graphid.core.mixin_helpers import *  # NOQA
>>> import ibeis
>>> ibs = ibeis.opendb(defaultdb='PZ_MTEST')
>>> infr = ibeis.AnnotInference(ibs, 'all', autoinit=True)
>>> label = 'orig_name_label'
>>> label = 'name_label'
>>> infr.find_mst_edges()
>>> infr.ensure_mst()
```

find_connecting_edges()

Searches for a small set of edges, which if reviewed as positive would ensure that each PCC is k-connected. Note that in somes cases this is not possible

1.1.1.1.7 graphid.core.mixin_invariants module

These check for certain invariants that should be maintained by the dynamic data structure.

class graphid.core.mixin_invariants.**AssertInvariants**

Bases: object

assert_edge(*edge*)

assert_invariants(*msg=''*)

assert_neg_metagraph()

Checks that the negative metgraph is correctly book-kept.

assert_union_invariant(*msg=''*)

assert_disjoint_invariant(*msg=''*)

assert_consistency_invariant(*msg=''*)

assert_recovery_invariant(*msg=''*)

1.1.1.1.8 graphid.core.mixin_loops module

class graphid.core.mixin_loops.**InfrLoops**

Bases: object

Algorithm control flow loops

main_gen(*max_loops=None*, *use_refresh=True*)

The main outer loop.

This function is designed as an iterator that will execute the graph algorithm main loop as automatically as possible, but if user input is needed, it will pause and yield the decision it needs help with. Once feedback is given for this item, you can continue the main loop by calling next. StopIteration is raised once the algorithm is complete.

Parameters

- **max_loops** (*int*) – maximum number of times to run the outer loop, i.e. ranking is run at most this many times.
- **use_refresh** (*bool*) – allow the refresh criterion to stop the algo

Notes

Different phases of the main loop are implemented as subiterators

CommandLine

```
python -m graphid.core.mixin_loops InfrLoops.main_gen
```

Example

```
>>> from graphid.core.mixin_simulation import UserOracle
>>> from graphid import demo
>>> infr = demo.demodata_infr(num_pccs=3, size=5)
>>> infr.params['manual.n_peek'] = 10
>>> infr.params['ranking.ntop'] = 1
>>> infr.oracle = UserOracle(.99, rng=0)
>>> infr.simulation_mode = False
>>> infr.reset()
>>> gen = infr.main_gen()
>>> while True:
>>>     try:
>>>         reviews = next(gen)
>>>         edge, priority, data = reviews[0]
>>>         feedback = infr.request_oracle_review(edge)
>>>         infr.add_feedback(edge, **feedback)
>>>     except StopIteration:
>>>         break
```

hardcase_review_gen()

Subiterator for hardcase review

Re-review non-confident edges that vsone did not classify correctly

ranked_list_gen(*use_refresh=True*)

Subiterator for phase1 of the main algorithm

Calls the underlying ranking algorithm and prioritizes the results

incon_recovery_gen()

Subiterator for recovery mode of the mainm algorithm

Iterates until the graph is consistent

Note: inconsistency recovery is implicitly handled by the main algorithm, so other phases do not need to call this explicitly. This exists for the case where the only mode we wish to run is inconsistency recovery.

---

pos_redun_gen()

Subiterator for phase2 of the main algorithm.

Searches for decisions that would commplete positive redundancy

CommandLine

```
python -m graphid.core.mixin_loops InfrLoops.pos_redun_gen
```

Example

```
>>> from graphid.core.mixin_loops import *
>>> from graphid import demo
>>> infr = demo.demodata_infr(num_pccs=3, size=5, pos_redun=1)
>>> gen = infr.pos_redun_gen()
>>> feedback = next(gen)
>>> edge_ = feedback[0][0]
>>> print(edge_)
(1, 5)
```

neg_redun_gen()

Subiterator for phase3 of the main algorithm.

Searches for decisions that would commplete negative redundancy

_inner_priority_gen(*use_refresh=False*, *only_auto=False*)

Helper function that implements the general inner priority loop.

Executes reviews until the queue is empty or needs refresh

Parameters

- **user_refresh** (*bool*) – if True enables the refresh criteria. (set to True in Phase 1)
- **only_auto** (*bool*) – reviews unless the graph is inconsistent. (set to True in Phase 3)

Notes

The caller is responsible for populating the priority queue. This will iterate until the queue is empty or the refresh critieron is triggered.

init_refresh()

start_id_review(*max_loops=None*, *use_refresh=None*)

main_loop(*max_loops=None*, *use_refresh=True*)

DEPRICATED

use list(infr.main_gen) instead or assert not any(infr.main_gen()) maybe this is fine.

class graphid.core.mixin_loops.**InfrReviewers**

Bases: object

try_auto_review(*edge*)

request_oracle_review(*edge*, ***kw*)

_make_review_tuple(*edge*, *priority=None*)

Makes tuple to be sent back to the user

emit_manual_review(*edge*, *priority=None*)

Emits a signal containing edges that need review. The callback should present them to a user, get feedback, and then call on_accpet.

skip(*edge*)

accept(*feedback*)

Called when user has completed feedback from qt or web

continue_review()

1.1.1.1.9 graphid.core.mixin_priority module

class graphid.core.mixin_priority.**Priority**

Bases: object

Handles prioritization of edges for review.

Example

```
>>> from graphid.core.mixin_priority import *  # NOQA
>>> from graphid import demo
>>> infr = demo.demodata_infr(num_pccs=20)
```

remaining_reviews()

_pop(**args*)

Wraps queue so ordering is determenistic

_push(*edge*, *priority*)

Wraps queue so ordering is determenistic

_peek_many(*n*)

Wraps queue so ordering is determenistic

_remove_edge_priority(*edges*)

_reinstate_edge_priority(*edges*)

_increase_priority(*edges*, *amount=10*)

remove_internal_priority(*cc*)

reinstate_internal_priority(*cc*)

prioritize(*metric=None*, *edges=None*, *scores=None*, *force_inconsistent=True*, *reset=False*)

Adds edges to the priority queue.

Note that these edges must already exist in the *infr.unreviewed_graph* as unreviewed edges. By default the *prob_match* edge attribute is used to sort edges. If you have registered a verification algorithm, then these scores are computed using *infr.ensure_priority_scores(edges)*. However, you can have all this done for you by simply calling *infr.add_candidate_edges(edges)* or *infr.refresh_candidate_edges()*.

Example

```
>>> from graphid.core.mixin_priority import *  # NOQA
>>> from graphid import demo
>>> infr = demo.demodata_infr(num_pccs=7, size=5)
>>> infr.ensure_cliques(meta_decision=SAME)
>>> # Add a negative edge inside a PCC
>>> ccs = list(infr.positive_components())
>>> edge1 = tuple(list(ccs[0])[0:2])
>>> edge2 = tuple(list(ccs[1])[0:2])
>>> infr.add_feedback(edge1, NEGTV)
>>> infr.add_feedback(edge2, NEGTV)
>>> num_new = infr.prioritize(reset=True)
>>> order = infr._peek_many(np.inf)
>>> scores = util.take_column(order, 1)
>>> assert scores[0] > 10
>>> assert len(scores) == num_new, 'should prioritize two hypotheis edges'
>>> unrev_edges = set(infr.unreviewed_graph.edges())
>>> err_edges = set(ub.flatten(infr.nid_to_errors.values()))
>>> edges = set(list(unrev_edges - err_edges)[0:2])
>>> edges.update(list(err_edges)[0:2])
>>> num_new = infr.prioritize(edges=edges, reset=True)
>>> order2 = infr._peek_many(np.inf)
>>> scores2 = np.array(util.take_column(order2, 1))
>>> assert np.all(scores2[0:2] > 10)
>>> assert np.all(scores2[2:] < 10)
```

push(*edge*, *priority=None*)

Push an edge back onto the queue

pop()

Main interface to the priority queue used by the algorithm loops. Pops the highest priority edge from the queue.

peek()

peek_many(*n*)

Peeks at the top n edges in the queue.

Example

```
>>> # ENABLE_DOCTEST
>>> from graphid.core.mixin_priority import *  # NOQA
>>> from graphid import demo
>>> infr = demo.demodata_infr(num_pccs=7, size=5)
>>> infr.refresh_candidate_edges()
>>> infr.peek_many(50)
```

confidently_connected(*u*, *v*, *thresh=2*)

Checks if u and v are conneted by edges above a confidence threshold

confidently_separated(*u*, *v*, *thresh=2*)

Checks if u and v are conneted by edges above a confidence threshold

Example

```
>>> from graphid.core.mixin_priority import *  # NOQA
>>> from graphid import demo
>>> infr = demo.demodata_infr(ccs=[(1, 2), (3, 4), (5, 6), (7, 8)])
>>> infr.add_feedback((1, 5), NEGTV)
>>> infr.add_feedback((5, 8), NEGTV)
>>> infr.add_feedback((6, 3), NEGTV)
>>> u, v = (1, 4)
>>> thresh = 0
>>> assert not infr.confidently_separated(u, v, thresh)
>>> infr.add_feedback((2, 3), NEGTV)
>>> assert not infr.confidently_separated(u, v, thresh)
```

generate_reviews(*pos_redun=None*, *neg_redun=None*, *data=False*)

Dynamic generator that yeilds high priority reviews

_generate_reviews(*data=False*)

1.1.1.1.10 graphid.core.mixin_redundancy module

Functionality related to the k-edge redundancy measures

class graphid.core.mixin_redundancy.**_RedundancyComputers**

Bases: object

methods for computing redundancy

These are used to compute redundancy bookkeeping structures. Thus, they should not use them in their calculations.

is_pos_redundant(*cc*, *k=None*, *relax=None*, *assume_connected=False*)

Tests if a group of nodes is positive redundant. (ie. if the group is k-edge-connected)

CommandLine

```
python -m graphid.core.mixin_dynamic _RedundancyComputers.is_pos_redundant
```

Example

```
>>> from graphid import demo
>>> infr = demo.demodata_infr(ccs=[(1, 2, 3)], pos_redun=1)
>>> cc = infr.pos_graph.connected_to(1)
>>> flag1 = infr.is_pos_redundant(cc)
>>> infr.add_feedback((1, 3), POSTV)
>>> flag2 = infr.is_pos_redundant(cc, k=2)
>>> flags = [flag1, flag2]
>>> print('flags = %r' % (flags,))
flags = [False, True]
>>> # xdoc: +REQUIRES(--show)
>>> from graphid import util
>>> infr.show()
>>> util.show_if_requested()
```

is_neg_redundant(*cc1*, *cc2*, *k=None*)

Tests if two disjoint groups of nodes are negative redundant (ie. have at least k negative edges between them).

CommandLine

```
python -m graphid.core.mixin_dynamic _RedundancyComputers.is_neg_redundant --
↪show
```

Example

```
>>> from graphid import demo
>>> infr = demo.demodata_infr(ccs=[(1, 2), (3, 4)], ignore_pair=True)
>>> infr.params['redun.neg'] = 2
>>> cc1 = infr.pos_graph.connected_to(1)
>>> cc2 = infr.pos_graph.connected_to(3)
>>> flag1 = infr.is_neg_redundant(cc1, cc2)
>>> infr.add_feedback((1, 3), NEGTV)
>>> flag2 = infr.is_neg_redundant(cc1, cc2)
>>> infr.add_feedback((2, 4), NEGTV)
>>> flag3 = infr.is_neg_redundant(cc1, cc2)
>>> flags = [flag1, flag2, flag3]
>>> print('flags = %r' % (flags,))
>>> assert flags == [False, False, True]
>>> # xdoc: +REQUIRES(--show)
>>> from graphid import util
>>> infr.show()
>>> util.show_if_requested()
```

find_neg_nids_to(*cc*)

Find the nids with at least one negative edge external to this cc.

find_neg_nid_freq_to(*cc*)

Find the number of edges leaving *cc* and directed towards specific names.

find_neg_redun_nids_to(*cc*)

Get PCCs that are k-negative redundant with *cc*

Example

```
>>> from graphid import demo
>>> infr = demo.demodata_infr()
>>> node = 20
>>> cc = infr.pos_graph.connected_to(node)
>>> infr.params['redun.neg'] = 2
>>> infr.find_neg_redun_nids_to(cc)
```

find_pos_redundant_pccs(*k=None*, *relax=None*)

find_non_pos_redundant_pccs(*k=None*, *relax=None*)

Get PCCs that are not k-positive-redundant

find_non_neg_redun_pccs(*k=None*, *cc=None*)

Get pairs of PCCs that are not complete.

Parameters

- **k** (*int*) – level of redunency to be considered complete
- **cc** (*set, optional*) – if specified only search for other pccs that are not negative redundant to this particular cc

Example

```
>>> from graphid import demo
>>> infr = demo.demodata_infr(pcc_sizes=[1, 1, 2, 3, 5, 8], ignore_pair=True)
>>> non_neg_pccs = list(infr.find_non_neg_redun_pccs(k=2))
>>> assert len(non_neg_pccs) == (6 * 5) / 2
```

find_pos_redun_nids()

recomputes infr.pos_redun_nids

find_neg_redun_nids()

recomputes edges in infr.neg_redun_metagraph

class graphid.core.mixin_redundancy.**_RedundancyAugmentation**

Bases: object

find_neg_augment_edges(*cc1*, *cc2*, *k=None*)

Find enough edges to between two pccs to make them k-negative complete The two CCs should be disjoint and not have any positive edges between them.

Parameters

- **cc1** (*set*) – nodes in one PCC

- **cc2** (*set*) – nodes in another positive-disjoint PCC
- **k** (*int*) – redundnacy level (if None uses infr.params['redun.neg'])

Example

```
>>> from graphid import demo
>>> k = 2
>>> cc1, cc2 = {1}, {2, 3}
>>> # --- return an augmentation if feasible
>>> infr = demo.demodata_infr(ccs=[cc1, cc2], ignore_pair=True)
>>> edges = set(infr.find_neg_augment_edges(cc1, cc2, k=k))
>>> assert edges == {(1, 2), (1, 3)}
>>> # --- if infeasible return a partial augmentation
>>> infr.add_feedback((1, 2), INCMP)
>>> edges = set(infr.find_neg_augment_edges(cc1, cc2, k=k))
>>> assert edges == {(1, 3)}
```

find_pos_augment_edges(*pcc*, *k=None*)

[[1, 0], [0, 2], [1, 2], [3, 1]] pos_sub = nx.Graph([[0, 1], [1, 2], [0, 2], [1, 3]])

find_pos_redun_candidate_edges(*k=None*, *verbose=False*)

Searches for augmenting edges that would make PCCs k-positive redundant

CommandLine

```
python -m graphid.core.mixin_dynamic _RedundancyAugmentation.find_pos_redun_
↪candidate_edges
```

Doctest

```
>>> from graphid.core.mixin_redundancy import *  # NOQA
>>> from graphid import demo
>>> # FIXME: this behavior seems to change depending on Python version
>>> infr = demo.demodata_infr(ccs=[(1, 2, 3, 4, 5), (7, 8, 9, 10)], pos_
↪redun=1)
>>> infr.add_feedback((2, 5), POSTV)
>>> infr.add_feedback((1, 5), INCMP)
>>> infr.params['redun.pos'] = 2
>>> candidate_edges = sorted(infr.find_pos_redun_candidate_edges())
...
>>> result = ('candidate_edges = ' + ub.urepr(candidate_edges, nl=0))
>>> print(result)
candidate_edges = [(1, 4), ..., (7, 10)]
```

find_neg_redun_candidate_edges(*k=None*)

Get pairs of PCCs that are not complete. Finds edges that might complete them.

Example

```
>>> from graphid import demo
>>> infr = demo.demodata_infr(ccs=[(1,), (2,), (3,)], ignore_pair=True)
>>> edges = list(infr.find_neg_redun_candidate_edges())
>>> assert len(edges) == 3, 'all should be needed here'
>>> infr.add_feedback_from(edges, evidence_decision=NEGTV)
>>> assert len(list(infr.find_neg_redun_candidate_edges())) == 0
```

Example

```
>>> from graphid import demo
>>> infr = demo.demodata_infr(pcc_sizes=[3] * 20, ignore_pair=True)
>>> ccs = list(infr.positive_components())
>>> gen = infr.find_neg_redun_candidate_edges(k=2)
>>> for edge in gen:
>>>     # What happens when we make ccs positive
>>>     print(infr.node_labels(edge))
>>>     infr.add_feedback(edge, evidence_decision=POSTV)
>>> import ubelt as ub
>>> infr = demo.demodata_infr(pcc_sizes=[1] * 30, ignore_pair=True)
>>> ccs = list(infr.positive_components())
>>> gen = infr.find_neg_redun_candidate_edges(k=3)
>>> for chunk in ub.chunks(gen, 2):
>>>     for edge in chunk:
>>>         # What happens when we make ccs positive
>>>         print(infr.node_labels(edge))
>>>         infr.add_feedback(edge, evidence_decision=POSTV)
```

list(gen)

class graphid.core.mixin_redundancy.**Redundancy**

Bases: *_RedundancyComputers*, *_RedundancyAugmentation*

methods for dynamic redundancy book-keeping

is_flagged_as_redun(*edge*)

Tests redundancy against bookkeeping structure against cache

filter_edges_flagged_as_redun(*edges*)

Returns only edges that are not flagged as redundant. Uses bookkeeping structures

Example

```
>>> from graphid import demo
>>> infr = demo.demodata_infr(num_pccs=1, size=4)
>>> infr.clear_edges()
>>> infr.ensure_cliques()
>>> infr.clear_feedback()
>>> print(ub.urepr(infr.status()))
>>> nonredun_edges = list(infr.filter_edges_flagged_as_redun(
```

(continues on next page)

(continued from previous page)

```
>>>     infr.unreviewed_graph.edges()))
>>> assert len(nonredun_edges) == 6
```

update_extern_neg_redun(*nid*, *may_add=True*, *may_remove=True*, *force=False*)

Checks if *nid* is negative redundant to any other *cc* it has at least one negative review to. (TODO: NEG REDUN CAN BE CONSOLIDATED VIA NEG-META-GRAPH)

update_neg_redun_to(*nid1*, *other_nids*, *may_add=True*, *may_remove=True*, *force=False*)

Checks if nid1 is neg redundant to other_nids. Edges are either removed or added to the queue appropriately. (TODO: NEG REDUN CAN BE CONSOLIDATED VIA NEG-META-GRAPH)

update_pos_redun(*nid*, *may_add=True*, *may_remove=True*, *force=False*)

Checks if a PCC is newly, or no longer positive redundant. Edges are either removed or added to the queue appropriately.

_set_pos_redun_flag(*nid*, *flag*)

Flags or unflags an nid as positive redundant.

_set_neg_redun_flags(*nid1*, *other_nids*, *flags*)

Flags or unflags an nid1 as negative redundant with other nids. (TODO: NEG REDUN CAN BE CONSOLIDATED VIA NEG-META-GRAPH)

_purge_redun_flags(*nid*)

Removes positive and negative redundancy from nids and all other PCCs touching nids respectively. Return the external PCC nids.

(TODO: NEG REDUN CAN BE CONSOLIDATED VIA NEG-META-GRAPH)

1.1.1.1.11 graphid.core.mixin_simulation module

Mixin functionality for experiments, tests, and simulations. This includes recordings measures used to generate plots in JC's thesis.

class graphid.core.mixin_simulation.**SimulationHelpers**

Bases: object

init_simulation(*oracle_accuracy=1.0*, *k_redun=2*, *enable_autoreview=True*, *enable_inference=True*, *classifiers=None*, *match_state_thresh=None*, *max_outer_loops=None*, *name=None*)

init_test_mode()

measure_error_edges()

measure_metrics()

_print_previous_loop_statistics(*count*)

_dynamic_test_callback(*edge*, *decision*, *prev_decision*, *user_id*)

class graphid.core.mixin_simulation.**UserOracle**(*accuracy*, *rng*)

Bases: object

review(*edge*, *truth*, *infr*, *accuracy=None*)

1.1.1.1.12 graphid.core.mixin_viz module

class graphid.core.mixin_viz.**GraphVisualization**

Bases: object

contains plotting related code

_get_truth_colors()

property _error_color

_get_cmap()

initialize_visual_node_attrs(*graph=None*)

update_node_image_config(***kwargs*)

update_node_image_attribute(*use_image=False*, *graph=None*)

get_colored_edge_weights(*graph=None*, *highlight_reviews=True*)

get_colored_weights(*weights*)

property visual_edge_attrs

all edge visual attrs

property visual_edge_attrs_appearance

attrs that pertain to edge color and style

property visual_edge_attrs_space

attrs that pertain to edge positioning in a plot

property visual_node_attrs

simplify_graph(*graph=None*, *copy=True*)

pin_node_layout()

Ensures a node layout exists and then sets the pin attribute on each node, which tells graphviz not to change node positions. Useful for making before and after pictures.

update_visual_attrs(*graph=None*, *show_reviewed_edges=True*, *show_unreviewed_edges=False*, *show_inferred_diff=True*, *show_inferred_same=True*, *show_recent_review=False*, *highlight_reviews=True*, *show_inconsistency=True*, *wavy=False*, *simple_labels=False*, *show_labels=True*, *reposition=True*, *use_image=False*, *edge_overrides=None*, *node_overrides=None*, *colorby='name_label'*, ***kwargs*)

show_graph(*graph=None*, *use_image=False*, *update_attrs=True*, *with_colorbar=False*, *pnum=(1, 1, 1)*, *zoomable=True*, *pickable=False*, ***kwargs*)

Parameters

- **infr** (*?*)
- **graph** (*None*) – (default = None)
- **use_image** (*bool*) – (default = False)
- **update_attrs** (*bool*) – (default = True)
- **with_colorbar** (*bool*) – (default = False)

- **pnum** (*tuple*) – plot number(default = (1, 1, 1))
- **zoomable** (*bool*) – (default = True)
- **pickable** (*bool*) – (de = False)
- ****kwargs** – verbose, with_labels, fnum, layout, ax, pos, img_dict, title, layoutkw, framewidth, modify_ax, as_directed, hacknoedge, hacknode, node_labels, arrow_width, fontsize, fontweight, fontname, fontfamilty, fontproperties

Example

```
>>> # xdoctest: +REQUIRES(module:pygraphviz)
>>> from graphid import demo
>>> infr = demo.demodata_infr(ccs=util.estarmap(
>>>    range, [(1, 6), (6, 10), (10, 13), (13, 15), (15, 16),
>>>            (17, 20)]))
>>> pnum_ = util.PlotNums(nRows=1, nCols=3)
>>> infr.show_graph(show_cand=True, simple_labels=True, pickable=True, fnum=1,␣
↪pnum=pnum_())
>>> infr.add_feedback((1, 5), INCMP)
>>> infr.add_feedback((14, 18), INCMP)
>>> infr.refresh_candidate_edges()
>>> infr.show_graph(show_cand=True, simple_labels=True, pickable=True, fnum=1,␣
↪pnum=pnum_())
>>> infr.add_feedback((17, 18), NEGTV)  # add inconsistency
>>> infr.apply_nondynamic_update()
>>> infr.show_graph(show_cand=True, simple_labels=True, pickable=True, fnum=1,␣
↪pnum=pnum_())
>>> util.show_if_requested()
```

show_edge(*edge*, *fnum=None*, *pnum=None*, ***kwargs*)

debug_edge_repr()

repr_edge_data(*all_edge_data*, *visual=True*)

show_error_case(*aids*, *edge=None*, *error_edges=None*, *colorby=None*, *fnum=1*)

Example

show(*graph=None*, *use_image=False*, *update_attrs=True*, *with_colorbar=False*, *pnum=(1, 1, 1)*, *zoomable=True*, *pickable=False*, ***kwargs*)

Parameters

- **infr** (*?*)
- **graph** (*None*) – (default = None)
- **use_image** (*bool*) – (default = False)
- **update_attrs** (*bool*) – (default = True)
- **with_colorbar** (*bool*) – (default = False)
- **pnum** (*tuple*) – plot number(default = (1, 1, 1))
- **zoomable** (*bool*) – (default = True)
- **pickable** (*bool*) – (de = False)

- ****kwargs** – verbose, with_labels, fnum, layout, ax, pos, img_dict, title, layoutkw, framewidth, modify_ax, as_directed, hacknoedge, hacknode, node_labels, arrow_width, fontsize, fontweight, fontname, fontfamilty, fontproperties

Example

```
>>> # xdoctest: +REQUIRES(module:pygraphviz)
>>> from graphid import demo
>>> infr = demo.demodata_infr(ccs=util.estarmap(
>>>    range, [(1, 6), (6, 10), (10, 13), (13, 15), (15, 16),
>>>            (17, 20)]))
>>> pnum_ = util.PlotNums(nRows=1, nCols=3)
>>> infr.show_graph(show_cand=True, simple_labels=True, pickable=True, fnum=1,␣
↪pnum=pnum_())
>>> infr.add_feedback((1, 5), INCMP)
>>> infr.add_feedback((14, 18), INCMP)
>>> infr.refresh_candidate_edges()
>>> infr.show_graph(show_cand=True, simple_labels=True, pickable=True, fnum=1,␣
↪pnum=pnum_())
>>> infr.add_feedback((17, 18), NEGTV)  # add inconsistency
>>> infr.apply_nondynamic_update()
>>> infr.show_graph(show_cand=True, simple_labels=True, pickable=True, fnum=1,␣
↪pnum=pnum_())
>>> util.show_if_requested()
```

graphid.core.mixin_viz.**on_pick**(*event*, *infr=None*)

graphid.core.mixin_viz.**color_nodes**(*graph*, *labelattr='label'*, *brightness=0.878*, *outof=None*, *sat_adjust=None*)

Colors edges and nodes by nid

graphid.core.mixin_viz.**nx_ensure_agraph_color**(*graph*)

changes colors to hex strings on graph attrs

1.1.1.1.13 graphid.core.refresh module

class graphid.core.refresh.**RefreshCriteria**(*window=20*, *patience=72*, *thresh=0.1*, *method='binomial'*)

Bases: object

Determine when to re-query for candidate edges.

Models an upper bound on the probability that any of the next *patience* reviews will be label-changing (meaningful). Once this probability is below a threshold the criterion triggers. The model is either binomial or poisson. They both work about the same. The binomial is a slightly better model.

Does this by maintaining an estimate of the probability any particular review will be label-chaging using an exponentially weighted moving average. This is the rate parameter / individual event probability.

clear()

check()

prob_any_remain(*n_remain_edges=None*)

_prob_none_remain(*n_remain_edges=None*)

pred_num_positives(*n_remain_edges*)

Uses poisson process to estimate remaining positive reviews.

Multipling mu * n_remain_edges gives a probabilistic upper bound on the number of errors remaning. This only provides a real estimate if reviewing in a random order

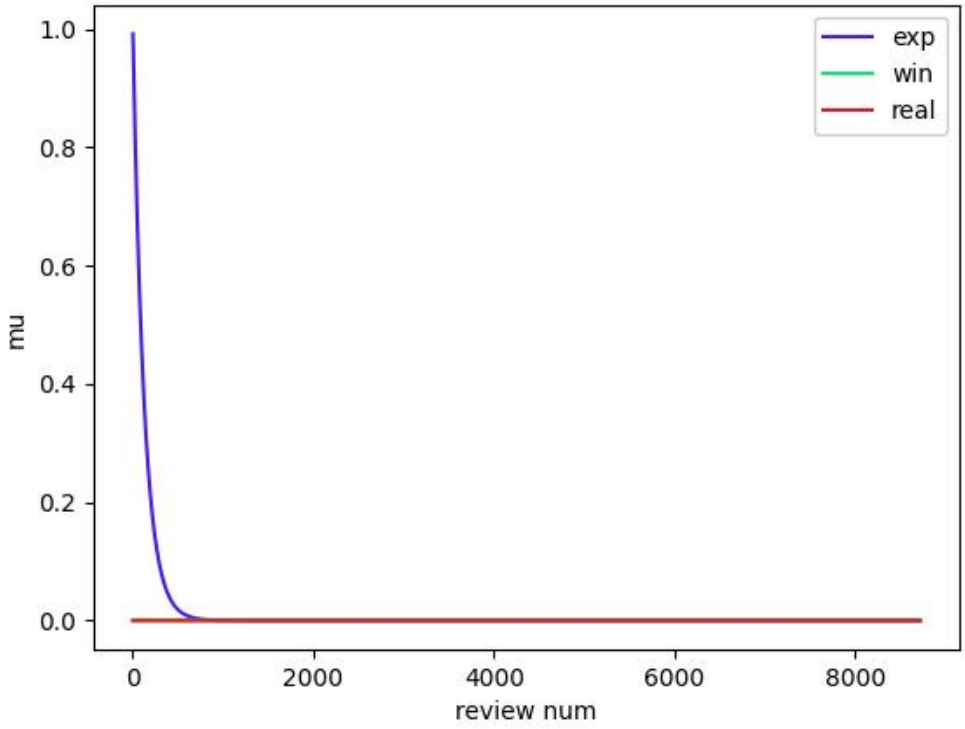
Example

```
>>> # ENABLE_DOCTEST
>>> from graphid.core.refresh import *  # NOQA
>>> from graphid import demo
>>> infr = demo.demodata_infr(num_pccs=50, size=4, size_std=2)
>>> edges = list(infr.ranker.predict_candidate_edges(infr.aids, K=100))
>>> #edges = util.shuffle(sorted(edges), rng=321)
>>> scores = np.array(infr.verifier.predict_edges(edges))
>>> sortx = scores.argsort()[::-1]
>>> edges = list(ub.take(edges, sortx))
>>> scores = scores[sortx]
>>> ys = infr.match_state_df(edges)[POSTV].values
>>> y_remainsum = ys[::-1].cumsum()[::-1]
>>> refresh = RefreshCriteria(window=250)
>>> n_pred_list = []
>>> n_real_list = []
>>> xdata = []
>>> for count, (edge, y) in enumerate(zip(edges, ys)):
>>>     refresh.add(y, user_id='user:oracle')
>>>     n_remain_edges = len(edges) - count
>>>     n_pred = refresh.pred_num_positives(n_remain_edges)
>>>     n_real = y_remainsum[count]
>>>     if count == 2000:
>>>         break
>>>     n_real_list.append(n_real)
>>>     n_pred_list.append(n_pred)
>>>     xdata.append(count + 1)
>>> # xdoctest: +REQUIRES(--show)
>>> import plottool_ibeis as pt
>>> pt.qtensure()
>>> n_pred_list = n_pred_list[10:]
>>> n_real_list = n_real_list[10:]
>>> xdata = xdata[10:]
>>> pt.multi_plot(xdata, [n_pred_list, n_real_list], marker='',
>>>               label_list=['pred', 'real'], xlabel='review num',
>>>               ylabel='pred remaining merges')
>>> stop_point = xdata[np.where(y_remainsum[10:] == 0)[0][0]]
>>> pt.gca().plot([stop_point, stop_point], [0, int(max(n_pred_list))], 'g-')
```

add(*meaningful*, *user_id*, *decision=None*)

ave(*method='exp'*)

```
>>> from graphid import demo
>>> infr = demo.demodata_infr(num_pccs=40, size=4, size_std=2, ignore_
↪pair=True)
>>> edges = list(infr.ranker.predict_candidate_edges(infr.aids, K=100))
>>> scores = np.array(infr.verifier.predict_edges(edges))
>>> #sortx = util.shuffle(np.arange(len(edges)), rng=321)
>>> sortx = scores.argsort()[::-1]
>>> edges = list(ub.take(edges, sortx))
>>> scores = scores[sortx]
>>> ys = infr.match_state_df(edges)[POSTV].values
>>> y_remainsum = ys[::-1].cumsum()[::-1]
>>> refresh = RefreshCriteria(window=250)
>>> ma1 = []
>>> ma2 = []
>>> reals = []
>>> xdata = []
>>> for count, (edge, y) in enumerate(zip(edges, ys)):
>>>     refresh.add(y, user_id='user:oracle')
>>>     ma1.append(refresh._ewma)
>>>     ma2.append(refresh.pos_frac)
>>>     n_real = y_remainsum[count] / (len(edges) - count)
>>>     reals.append(n_real)
>>>     xdata.append(count + 1)
>>> # xdoctest: +REQUIRES(--show)
>>> from graphid import util
>>> util.qtensure()
>>> util.multi_plot(xdata, [ma1, ma2, reals], marker='',
>>>                 label_list=['exp', 'win', 'real'], xlabel='review num',
>>>                 ylabel='mu')
```



property pos_frac

graphid.core.refresh.**demo_refresh**()

CommandLine

```
python -m graphid.core.refresh demo_refresh \
        --num_pccs=40 --size=2 --show
```

Example

```
>>> # ENABLE_DOCTEST
>>> from graphid.core.refresh import *  # NOQA
>>> demo_refresh()
>>> util.show_if_requested()
```

graphid.core.refresh.**_dev_iters_until_threshold**()

INTERACTIVE DEVELOPMENT FUNCTION

How many iterations of ewma until you hit the poisson / biniomal threshold

This establishes a principled way to choose the threshold for the refresh criterion in my thesis. There are paramters — moving parts — that we need to work with: *a* the patience, *s* the span, and *mu* our ewma.

s is a span paramter indicating how far we look back.

mu is the average number of label-changing reviews in roughly the last *s* manual decisions.

These numbers are used to estimate the probability that any of the next *a* manual decisions will be label-chanigng. When that probability falls below a threshold we terminate. The goal is to choose *a*, *s*, and the threshold *t*, such that the probability will fall below the threshold after a maximum of *a* consecutive non-label-chaning reviews. IE we want to tie the patience paramter (how far we look ahead) to how far we actually are willing to go.

1.1.1.1.14 graphid.core.state module

class graphid.core.state.**_Common**

Bases: object

class graphid.core.state.**_ConstHelper**(*name*, *parents*, *dct*)

Bases: type

Adds code and nice constants to an integer version of a class

cls = META_DECISION code_cls = META_DECISION_CODE

class graphid.core.state.**EVIDENCE_DECISION**

Bases: *_Common*

TODO: change to EVIDENCE_DECISION / VISUAL_DECISION Enumerated types of review codes and texts

Notes

Unreviewed: Not comparared yet. nomatch: Visually comparable and the different match: Visually comparable and the same notcomp: Not comparable means it is actually impossible to determine. unknown: means that it was reviewed, but we just can't figure it out.

UNREVIEWED = None

NEGATIVE = 0

POSITIVE = 1

INCOMPARABLE = 2

UNKNOWN = 3

INT_TO_CODE = {0: 'NEGTV', 1: 'POSTV', 2: 'INCMP', 3: 'UNKWN', None: 'UNREV'}

INT_TO_NICE = {0: 'Negative', 1: 'Positive', 2: 'Incomparable', 3: 'Unknown', None: 'Unreviewed'}

CODE_TO_NICE = {'INCMP': 'Incomparable', 'NEGTV': 'Negative', 'POSTV': 'Positive', 'UNKWN': 'Unknown', 'UNREV': 'Unreviewed'}

CODE_TO_INT = {'INCMP': 2, 'NEGTV': 0, 'POSTV': 1, 'UNKWN': 3, 'UNREV': None}

NICE_TO_CODE = {'Incomparable': 'INCMP', 'Negative': 'NEGTV', 'Positive': 'POSTV', 'Unknown': 'UNKWN', 'Unreviewed': 'UNREV'}

NICE_TO_INT = {'Incomparable': 2, 'Negative': 0, 'Positive': 1, 'Unknown': 3, 'Unreviewed': None}

MATCH_CODE = {'INCMP': 2, 'NEGTV': 0, 'POSTV': 1, 'UNKWN': 3, 'UNREV': None}

class CODE

Bases: object

INCOMPARABLE = 'INCMP'

NEGATIVE = 'NEGTV'

POSITIVE = 'POSTV'

UNKNOWN = 'UNKWN'

UNREVIEWED = 'UNREV'

class NICE

Bases: object

INCOMPARABLE = 'Incomparable'

NEGATIVE = 'Negative'

POSITIVE = 'Positive'

UNKNOWN = 'Unknown'

UNREVIEWED = 'Unreviewed'

class graphid.core.state.META_DECISION

Bases: *_Common*

Enumerated types of review codes and texts

Notes

unreviewed: we dont have a meta decision same: we know this is the same animal through non-visual means diff: we know this is the different animal through non-visual means

Example

```
>>> assert hasattr(META_DECISION, 'CODE')
>>> assert hasattr(META_DECISION, 'NICE')
>>> code1 = META_DECISION.INT_TO_CODE[META_DECISION.NULL]
>>> code2 = META_DECISION.CODE.NULL
>>> assert code1 == code2
>>> nice1 = META_DECISION.INT_TO_NICE[META_DECISION.NULL]
>>> nice2 = META_DECISION.NICE.NULL
>>> assert nice1 == nice2
```

NULL = None

DIFF = 0

SAME = 1

INT_TO_CODE = {0: 'diff', 1: 'same', None: 'null'}

INT_TO_NICE = {0: 'Different', 1: 'Same', None: 'NULL'}

CODE_TO_NICE = {'diff': 'Different', 'null': 'NULL', 'same': 'Same'}

CODE_TO_INT = {'diff': 0, 'null': None, 'same': 1}

NICE_TO_CODE = {'Different': 'diff', 'NULL': 'null', 'Same': 'same'}

NICE_TO_INT = {'Different': 0, 'NULL': None, 'Same': 1}

class CODE

Bases: object

DIFF = 'diff'

NULL = 'null'

SAME = 'same'

class NICE

Bases: object

DIFF = 'Different'

NULL = 'NULL'

SAME = 'Same'

class graphid.core.state.CONFIDENCE

Bases: *_Common*

UNKNOWN = None

GUESSING = 1

NOT_SURE = 2

PRETTY_SURE = 3

ABSOLUTELY_SURE = 4

INT_TO_CODE = {1: 'guessing', 2: 'not_sure', 3: 'pretty_sure', 4: 'absolutely_sure', None: 'unspecified'}

INT_TO_NICE = {1: 'Guessing', 2: 'Unsure', 3: 'Sure', 4: 'Doubtless', None: 'Unspecified'}

CODE_TO_NICE = {'absolutely_sure': 'Doubtless', 'guessing': 'Guessing', 'not_sure': 'Unsure', 'pretty_sure': 'Sure', 'unspecified': 'Unspecified'}

CODE_TO_INT = {'absolutely_sure': 4, 'guessing': 1, 'not_sure': 2, 'pretty_sure': 3, 'unspecified': None}

NICE_TO_CODE = {'Doubtless': 'absolutely_sure', 'Guessing': 'guessing', 'Sure': 'pretty_sure', 'Unspecified': 'unspecified', 'Unsure': 'not_sure'}

NICE_TO_INT = {'Doubtless': 4, 'Guessing': 1, 'Sure': 3, 'Unspecified': None, 'Unsure': 2}

class CODE

Bases: object

ABSOLUTELY_SURE = 'absolutely_sure'

GUESSING = 'guessing'

NOT_SURE = 'not_sure'

PRETTY_SURE = 'pretty_sure'

UNKNOWN = 'unspecified'

class NICE

Bases: object

ABSOLUTELY_SURE = 'Doubtless'

GUESSING = 'Guessing'

NOT_SURE = 'Unsure'

PRETTY_SURE = 'Sure'

UNKNOWN = 'Unspecified'

class graphid.core.state.QUAL

Bases: *_Common*

EXCELLENT = 5

GOOD = 4

OK = 3

POOR = 2

JUNK = 1

UNKNOWN = None

INT_TO_CODE = {1: 'junk', 2: 'poor', 3: 'ok', 4: 'good', 5: 'excellent', None: 'unspecified'}

INT_TO_NICE = {1: 'Junk', 2: 'Poor', 3: 'OK', 4: 'Good', 5: 'Excellent', None: 'Unspecified'}

CODE_TO_NICE = {'excellent': 'Excellent', 'good': 'Good', 'junk': 'Junk', 'ok': 'OK', 'poor': 'Poor', 'unspecified': 'Unspecified'}

CODE_TO_INT = {'excellent': 5, 'good': 4, 'junk': 1, 'ok': 3, 'poor': 2, 'unspecified': None}

NICE_TO_CODE = {'Excellent': 'excellent', 'Good': 'good', 'Junk': 'junk', 'OK': 'ok', 'Poor': 'poor', 'Unspecified': 'unspecified'}

NICE_TO_INT = {'Excellent': 5, 'Good': 4, 'Junk': 1, 'OK': 3, 'Poor': 2, 'Unspecified': None}

class CODE

Bases: object

EXCELLENT = 'excellent'

GOOD = 'good'

JUNK = 'junk'

OK = 'ok'

POOR = 'poor'

UNKNOWN = 'unspecified'

class NICE

Bases: object

EXCELLENT = 'Excellent'

GOOD = 'Good'

JUNK = 'Junk'

OK = 'OK'

POOR = 'Poor'

UNKNOWN = 'Unspecified'

class graphid.core.state.VIEW

Bases: *_Common*

categorical viewpoint using the faces of a Rhombicuboctahedron

References

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

UNKNOWN = None

R = 1

FR = 2

F = 3

FL = 4

L = 5

BL = 6

B = 7

BR = 8

U = 9

UF = 10

UB = 11

```
UL = 12

UR = 13

UFL = 14

UFR = 15

UBL = 16

UBR = 17

D = 18

DF = 19

DB = 20

DL = 21

DR = 22

DFL = 23

DFR = 24

DBL = 25

DBR = 26

INT_TO_CODE = {1:  'right', 10:  'upfront', 11:  'upback', 12:  'upleft', 13:
'upright', 14:  'upfrontleft', 15:  'upfrontright', 16:  'upbackleft', 17:
'upbackright', 18:  'down', 19:  'downfront', 2:  'frontright', 20:  'downback', 21:
'downleft', 22:  'downright', 23:  'downfrontleft', 24:  'downfrontright', 25:
'downbackleft', 26:  'downbackright', 3:  'front', 4:  'frontleft', 5:  'left', 6:
'backleft', 7:  'back', 8:  'backright', 9:  'up', None:  'unknown'}

INT_TO_NICE = {1:  'Right', 10:  'Up-Front', 11:  'Up-Back', 12:  'Up-Left', 13:
'Up-Right', 14:  'Up-Front-Left', 15:  'Up-Front-Right', 16:  'Up-Back-Left', 17:
'Up-Back-Right', 18:  'Down', 19:  'Down-Front', 2:  'Front-Right', 20:
'Down-Back', 21:  'Down-Left', 22:  'Down-Right', 23:  'Down-Front-Left', 24:
'Down-Front-Right', 25:  'Down-Back-Left', 26:  'Down-Back-Right', 3:  'Front', 4:
'Front-Left', 5:  'Left', 6:  'Back-Left', 7:  'Back', 8:  'Back-Right', 9:  'Up',
None:  'Unknown'}

CODE_TO_NICE = {'back':  'Back', 'backleft':  'Back-Left', 'backright':
'Back-Right', 'down':  'Down', 'downback':  'Down-Back', 'downbackleft':
'Down-Back-Left', 'downbackright':  'Down-Back-Right', 'downfront':  'Down-Front',
'downfrontleft':  'Down-Front-Left', 'downfrontright':  'Down-Front-Right',
'downleft':  'Down-Left', 'downright':  'Down-Right', 'front':  'Front',
'frontleft':  'Front-Left', 'frontright':  'Front-Right', 'left':  'Left', 'right':
'Right', 'unknown':  'Unknown', 'up':  'Up', 'upback':  'Up-Back', 'upbackleft':
'Up-Back-Left', 'upbackright':  'Up-Back-Right', 'upfront':  'Up-Front',
'upfrontleft':  'Up-Front-Left', 'upfrontright':  'Up-Front-Right', 'upleft':
'Up-Left', 'upright':  'Up-Right'}
```

```
CODE_TO_INT = {'back': 7, 'backleft': 6, 'backright': 8, 'down': 18, 'downback':
20, 'downbackleft': 25, 'downbackright': 26, 'downfront': 19, 'downfrontleft':
23, 'downfrontright': 24, 'downleft': 21, 'downright': 22, 'front': 3,
'frontleft': 4, 'frontright': 2, 'left': 5, 'right': 1, 'unknown': None, 'up':
9, 'upback': 11, 'upbackleft': 16, 'upbackright': 17, 'upfront': 10,
'upfrontleft': 14, 'upfrontright': 15, 'upleft': 12, 'upright': 13}

NICE_TO_CODE = {'Back': 'back', 'Back-Left': 'backleft', 'Back-Right':
'backright', 'Down': 'down', 'Down-Back': 'downback', 'Down-Back-Left':
'downbackleft', 'Down-Back-Right': 'downbackright', 'Down-Front': 'downfront',
'Down-Front-Left': 'downfrontleft', 'Down-Front-Right': 'downfrontright',
'Down-Left': 'downleft', 'Down-Right': 'downright', 'Front': 'front',
'Front-Left': 'frontleft', 'Front-Right': 'frontright', 'Left': 'left', 'Right':
'right', 'Unknown': 'unknown', 'Up': 'up', 'Up-Back': 'upback', 'Up-Back-Left':
'upbackleft', 'Up-Back-Right': 'upbackright', 'Up-Front': 'upfront',
'Up-Front-Left': 'upfrontleft', 'Up-Front-Right': 'upfrontright', 'Up-Left':
'upleft', 'Up-Right': 'upright'}

NICE_TO_INT = {'Back': 7, 'Back-Left': 6, 'Back-Right': 8, 'Down': 18,
'Down-Back': 20, 'Down-Back-Left': 25, 'Down-Back-Right': 26, 'Down-Front': 19,
'Down-Front-Left': 23, 'Down-Front-Right': 24, 'Down-Left': 21, 'Down-Right':
22, 'Front': 3, 'Front-Left': 4, 'Front-Right': 2, 'Left': 5, 'Right': 1,
'Unknown': None, 'Up': 9, 'Up-Back': 11, 'Up-Back-Left': 16, 'Up-Back-Right':
17, 'Up-Front': 10, 'Up-Front-Left': 14, 'Up-Front-Right': 15, 'Up-Left': 12,
'Up-Right': 13}
```

```
DIST = {(1, 1): 0, (1, 10): 2, (1, 11): 2, (1, 12): 3, (1, 13): 1, (1, 14): 3,
(1, 15): 1, (1, 16): 3, (1, 17): 1, (1, 18): 2, (1, 19): 2, (1, 2): 1, (1,
20): 2, (1, 21): 3, (1, 22): 1, (1, 23): 3, (1, 24): 1, (1, 25): 3, (1, 26):
1, (1, 3): 2, (1, 4): 3, (1, 5): 4, (1, 6): 3, (1, 7): 2, (1, 8): 1, (1, 9):
2, (1, None): None, (10, 1): 2, (10, 10): 0, (10, 11): 2, (10, 12): 2, (10,
13): 2, (10, 14): 1, (10, 15): 1, (10, 16): 2, (10, 17): 2, (10, 18): 3, (10,
19): 2, (10, 2): 2, (10, 20): 4, (10, 21): 3, (10, 22): 3, (10, 23): 2, (10,
24): 2, (10, 25): 3, (10, 26): 3, (10, 3): 1, (10, 4): 2, (10, 5): 2, (10, 6):
3, (10, 7): 3, (10, 8): 3, (10, 9): 1, (10, None): None, (11, 1): 2, (11, 10):
2, (11, 11): 0, (11, 12): 2, (11, 13): 2, (11, 14): 2, (11, 15): 2, (11, 16):
1, (11, 17): 1, (11, 18): 3, (11, 19): 4, (11, 2): 3, (11, 20): 2, (11, 21):
3, (11, 22): 3, (11, 23): 3, (11, 24): 3, (11, 25): 2, (11, 26): 2, (11, 3):
3, (11, 4): 3, (11, 5): 2, (11, 6): 2, (11, 7): 1, (11, 8): 2, (11, 9): 1,
(11, None): None, (12, 1): 3, (12, 10): 2, (12, 11): 2, (12, 12): 0, (12, 13):
2, (12, 14): 1, (12, 15): 2, (12, 16): 1, (12, 17): 2, (12, 18): 3, (12, 19):
3, (12, 2): 3, (12, 20): 3, (12, 21): 2, (12, 22): 4, (12, 23): 2, (12, 24):
3, (12, 25): 2, (12, 26): 3, (12, 3): 2, (12, 4): 2, (12, 5): 1, (12, 6): 2,
(12, 7): 2, (12, 8): 3, (12, 9): 1, (12, None): None, (13, 1): 1, (13, 10): 2,
(13, 11): 2, (13, 12): 2, (13, 13): 0, (13, 14): 2, (13, 15): 1, (13, 16): 2,
(13, 17): 1, (13, 18): 3, (13, 19): 3, (13, 2): 2, (13, 20): 3, (13, 21): 4,
(13, 22): 2, (13, 23): 3, (13, 24): 2, (13, 25): 3, (13, 26): 2, (13, 3): 2,
(13, 4): 3, (13, 5): 3, (13, 6): 3, (13, 7): 2, (13, 8): 2, (13, 9): 1, (13,
None): None, (14, 1): 3, (14, 10): 1, (14, 11): 2, (14, 12): 1, (14, 13): 2,
(14, 14): 0, (14, 15): 2, (14, 16): 2, (14, 17): 2, (14, 18): 3, (14, 19): 2,
(14, 2): 2, (14, 20): 3, (14, 21): 2, (14, 22): 3, (14, 23): 2, (14, 24): 2,
(14, 25): 2, (14, 26): 4, (14, 3): 1, (14, 4): 1, (14, 5): 1, (14, 6): 2, (14,
7): 3, (14, 8): 3, (14, 9): 1, (14, None): None, (15, 1): 1, (15, 10): 1, (15,
11): 2, (15, 12): 2, (15, 13): 1, (15, 14): 2, (15, 15): 0, (15, 16): 2, (15,
17): 2, (15, 18): 3, (15, 19): 2, (15, 2): 1, (15, 20): 3, (15, 21): 3, (15,
22): 2, (15, 23): 2, (15, 24): 2, (15, 25): 4, (15, 26): 2, (15, 3): 1, (15,
4): 2, (15, 5): 3, (15, 6): 3, (15, 7): 3, (15, 8): 2, (15, 9): 1, (15, None):
None, (16, 1): 3, (16, 10): 2, (16, 11): 1, (16, 12): 1, (16, 13): 2, (16, 14):
2, (16, 15): 2, (16, 16): 0, (16, 17): 2, (16, 18): 3, (16, 19): 3, (16, 2):
3, (16, 20): 2, (16, 21): 2, (16, 22): 3, (16, 23): 2, (16, 24): 4, (16, 25):
2, (16, 26): 2, (16, 3): 3, (16, 4): 2, (16, 5): 1, (16, 6): 1, (16, 7): 1,
(16, 8): 2, (16, 9): 1, (16, None): None, (17, 1): 1, (17, 10): 2, (17, 11):
1, (17, 12): 2, (17, 13): 1, (17, 14): 2, (17, 15): 2, (17, 16): 2, (17, 17):
0, (17, 18): 3, (17, 19): 3, (17, 2): 2, (17, 20): 2, (17, 21): 3, (17, 22):
2, (17, 23): 4, (17, 24): 2, (17, 25): 2, (17, 26): 2, (17, 3): 3, (17, 4): 3,
(17, 5): 3, (17, 6): 2, (17, 7): 1, (17, 8): 1, (17, 9): 1, (17, None): None,
(18, 1): 2, (18, 10): 3, (18, 11): 3, (18, 12): 3, (18, 13): 3, (18, 14): 3,
(18, 15): 3, (18, 16): 3, (18, 17): 3, (18, 18): 0, (18, 19): 1, (18, 2): 2,
(18, 20): 1, (18, 21): 1, (18, 22): 1, (18, 23): 1, (18, 24): 1, (18, 25): 1,
(18, 26): 1, (18, 3): 2, (18, 4): 2, (18, 5): 2, (18, 6): 2, (18, 7): 2, (18,
8): 2, (18, 9): 4, (18, None): None, (19, 1): 2, (19, 10): 2, (19, 11): 4,
(19, 12): 3, (19, 13): 3, (19, 14): 2, (19, 15): 2, (19, 16): 3, (19, 17): 3,
(19, 18): 1, (19, 19): 0, (19, 2): 2, (19, 20): 2, (19, 21): 2, (19, 22): 2,
(19, 23): 1, (19, 24): 1, (19, 25): 2, (19, 26): 2, (19, 3): 1, (19, 4): 2,
(19, 5): 2, (19, 6): 3, (19, 7): 3, (19, 8): 3, (19, 9): 3, (19, None): None,
(2, 1): 1, (2, 10): 2, (2, 11): 3, (2, 12): 3, (2, 13): 2, (2, 14): 2, (2,
15): 1, (2, 16): 3, (2, 17): 2, (2, 18): 2, (2, 19): 2, (2, 2): 0, (2, 20):
3, (2, 21): 3, (2, 22): 2, (2, 23): 2, (2, 24): 1, (2, 25): 3, (2, 26): 2, (2,
3): 1, (2, 4): 2, (2, 5): 3, (2, 6): 4, (2, 7): 3, (2, 8): 2, (2, 9): 2, (2,
None): None, (20, 1): 2, (20, 10): 4, (20, 11): 2, (20, 12): 3, (20, 13): 3,
(20, 14): 3, (20, 15): 3, (20, 16): 2, (20, 17): 2, (20, 18): 1, (20, 19): 2,
(20, 2): 3, (20, 20): 0, (20, 21): 2, (20, 22): 2, (20, 23): 2, (20, 24): 2,
(20, 25): 1, (20, 26): 1, (20, 3): 3, (20, 4): 3, (20, 5): 2, (20, 6): 2, (20,
7): 1, (20, 8): 2, (20, 9): 3, (20, None): None, (21, 1): 3, (21, 10): 3, (21,
11): 3, (21, 12): 2, (21, 13): 4, (21, 14): 2, (21, 15): 3, (21, 16): 2, (21,
17): 3, (21, 18): 1, (21, 19): 2, (21, 2): 3, (21, 20): 2, (21, 21): 0, (21,
```

class CODE

Bases: object

B = 'back'

BL = 'backleft'

BR = 'backright'

D = 'down'

DB = 'downback'

DBL = 'downbackleft'

DBR = 'downbackright'

DF = 'downfront'

DFL = 'downfrontleft'

DFR = 'downfrontright'

DL = 'downleft'

DR = 'downright'

F = 'front'

FL = 'frontleft'

FR = 'frontright'

L = 'left'

R = 'right'

U = 'up'

UB = 'upback'

UBL = 'upbackleft'

UBR = 'upbackright'

UF = 'upfront'

UFL = 'upfrontleft'

UFR = 'upfrontright'

UL = 'upleft'

UNKNOWN = 'unknown'

UR = 'upright'

class NICE

Bases: object

```
        B = 'Back'
        BL = 'Back-Left'
        BR = 'Back-Right'
        D = 'Down'
        DB = 'Down-Back'
        DBL = 'Down-Back-Left'
        DBR = 'Down-Back-Right'
        DF = 'Down-Front'
        DFL = 'Down-Front-Left'
        DFR = 'Down-Front-Right'
        DL = 'Down-Left'
        DR = 'Down-Right'
        F = 'Front'
        FL = 'Front-Left'
        FR = 'Front-Right'
        L = 'Left'
        R = 'Right'
        U = 'Up'
        UB = 'Up-Back'
        UBL = 'Up-Back-Left'
        UBR = 'Up-Back-Right'
        UF = 'Up-Front'
        UFL = 'Up-Front-Left'
        UFR = 'Up-Front-Right'
        UL = 'Up-Left'
        UNKNOWN = 'Unknown'
        UR = 'Up-Right'
    d = None
    f1 = None
    f2 = None
```

1.1.1.2 Module contents

Regenerate Input Command mkinit ~/code/graphid/graphid/core

1.1.2 graphid.demo package

1.1.2.1 Submodules

1.1.2.1.1 graphid.demo.__main__ module

graphid.demo.__main__.**main**()

1.1.2.1.2 graphid.demo.demo_script module

graphid.demo.demo_script.**run_demo**()

CommandLine

```
python -m graphid.demo.demo_script run_demo --viz
python -m graphid.demo.demo_script run_demo
```

Example

```
>>> run_demo()
```

1.1.2.1.3 graphid.demo.dummy_algos module

class graphid.demo.dummy_algos.**DummyRanker**(*verif*)

Bases: object

Generates dummy rankings

predict_single_ranking(*u*, *K=10*)

simulates the ranking algorithm. Order is defined using the dummy vsone scores, but tests are only applied to randomly selected gt and gf pairs. So, you usually will get a gt result, but you might not if all the scores are bad.

predict_rankings(*nodes*, *K=10*)

Yields a list ranked edges connected to each node.

predict_candidate_edges(*nodes*, *K=10*)

CommandLine

```
python -m graphid.demo.dummy_algos DummyRanker.predict_candidate_edges
```

Example

```
>>> from graphid import demo
>>> kwargs = dict(num_pccs=40, size=2)
>>> infr = demo.demodata_infr(**kwargs)
>>> edges = list(infr.ranker.predict_candidate_edges(infr.aids, K=100))
>>> scores = np.array(infr.verifier.predict_edges(edges))
>>> assert len(edges) > 0
```

class graphid.demo.dummy_algos.**DummyVerif**(*infr*)

Bases: object

Generates dummy scores between pairs of annotations. (not necesarilly existing edges in the graph)

CommandLine

```
python -m graphid.demo DummyVerif:1
```

Example

```
>>> from graphid.demo import *  # NOQA
>>> from graphid import demo
>>> kwargs = dict(num_pccs=6, p_incon=.5, size_std=2)
>>> infr = demo.demodata_infr(**kwargs)
>>> infr.dummy_verif.predict_edges([(1, 2)])
>>> infr.dummy_verif.predict_edges([(1, 21)])
>>> assert len(infr.dummy_verif.infr.task_probs['match_state']) == 2
```

predict_proba_df(*edges*)

CommandLine

```
python -m graphid.demo DummyVerif.predict_edges
```

Example

```
>>> from graphid import demo
>>> kwargs = dict(num_pccs=40, size=2)
>>> infr = demo.demodata_infr(**kwargs)
>>> verif = infr.dummy_verif
>>> edges = list(infr.graph.edges())
>>> probs = verif.predict_proba_df(edges)
```

predict_edges(*edges*)

show_score_probs()

CommandLine

```
python -m graphid.demo.dummy_algos DummyVerif.show_score_probs --show
```

Example

```
>>> from graphid import core
>>> from graphid import demo
>>> infr = core.AnnotInference()
>>> verif = demo.DummyVerif(infr)
>>> verif.show_score_probs()
>>> util.show_if_requested()
```

_get_truth(*edge*)

1.1.2.1.4 graphid.demo.dummy_infr module

graphid.demo.dummy_infr.**demodata_infr**(***kwargs*)

Kwargs:

num_pccs (list): implicit number of individuals ccs (list): explicit list of connected components

p_incon (float): probability a PCC is inconsistent p_incomp (float): probability an edge is incomparable n_incon (int): target number of inconsistent components (default 3)

pcc_size_mean (int): average number of annots per PCC pcc_size_std (float): std dev of annots per PCC

pos_redun (int): desired level of positive redundancy

infer (bool): whether or not to run inference by default default True

ignore_pair (bool): if True ignores all pairwise dummy edge generation p_pair_neg (float): default = .4 p_pair_incmp (float): default = .2 p_pair_unrev (float): default = 0.0

CommandLine

```
python -m graphid.demo.dummy_infr demodata_infr:0 --show
python -m graphid.demo.dummy_infr demodata_infr:1 --show
python -m utool.util_inspect recursive_parse_kwargs:2 --mod graphid.demo.dummy_infr␣
↪--func demodata_infr
```

Example

```
>>> from graphid import demo
>>> import networkx as nx
>>> kwargs = dict(num_pccs=6, p_incon=.5, size_std=2)
>>> infr = demo.demodata_infr(**kwargs)
>>> pccs = list(infr.positive_components())
>>> assert len(pccs) == kwargs['num_pccs']
>>> nonfull_pccs = [cc for cc in pccs if len(cc) > 1 and nx.is_empty(nx.
↪complement(infr.pos_graph.subgraph(cc)))]
>>> expected_n_incon = len(nonfull_pccs) * kwargs['p_incon']
>>> n_incon = len(list(infr.inconsistent_components()))
>>> print('status = ' + ub.urepr(infr.status(extended=True)))
>>> # xdoctest: +REQUIRES(--show)
>>> infr.show(pickable=True, groupby='name_label')
>>> util.show_if_requested()
```

Doctest

```
>>> from graphid import demo
>>> import networkx as nx
>>> kwargs = dict(num_pccs=0)
>>> infr = demo.demodata_infr(**kwargs)
```

1.1.2.2 Module contents

Regenerate Input Command mkinit ~/code/graphid/graphid/demo

1.1.3 graphid.util package

1.1.3.1 Submodules

1.1.3.1.1 graphid.util.mpl_plottool module

Port of the less useful parts of plottool that util_graphviz still depends on

TODO: try and depricate or refactor these

graphid.util.mpl_plottool.**get_plotdat_dict**(*ax*)
sets internal property to a matplotlib axis

graphid.util.mpl_plottool.**set_plotdat**(*ax*, *key*, *val*)
sets internal property to a matplotlib axis

graphid.util.mpl_plottool.**make_bbox**(*bbox*, *theta=0*, *bbox_color=None*, *ax=None*, *lw=2*, *alpha=1.0*, *align='center'*, *fill=None*, ***kwargs*)

graphid.util.mpl_plottool.**get_axis_xy_width_height**(*ax=None*, *xaug=0*, *yaug=0*, *waug=0*, *haug=0*)
gets geometry of a subplot

graphid.util.mpl_plottool.**ax_absolute_text**(*x_*, *y_*, *txt*, *ax=None*, *roffset=None*, ***kwargs*)

Base function for text

Kwargs:
horizontalalignment in ['right', 'center', 'left'], verticalalignment in ['top'] color

graphid.util.mpl_plottool.**cartoon_stacked_rects**(*xy*, *width*, *height*, *num=4*, *shift=None*, ***kwargs*)

pt.figure() xy = (.5, .5) width = .2 height = .2 ax = pt.gca() ax.add_collection(col)

graphid.util.mpl_plottool.**parse_fontkw**(***kwargs*)

Kwargs:
fontsize, fontfamilty, fontproperties

Example

```
>>> # xdoctest: +REQUIRES(module:matplotlib)
>>> parse_fontkw()
```

1.1.3.1.2 graphid.util.mplutil module

graphid.util.mplutil.**multi_plot**(*xdata=None*, *ydata=[]*, ***kwargs*)

plots multiple lines, bars, etc...

This is the big function that implements almost all of the heavy lifting in this file. Any function not using this should probably find a way to use it. It is pretty general and relatively clean.

Parameters

- **xdata** (*ndarray*) – can also be a list of arrays
- **ydata** (*list or dict of ndarrays*) – can also be a single array
- ****kwargs** –

  Misc:
  fnum, pnum, use_legend, legend_loc

  Labels:
  xlabel, ylabel, title, figtitle ticksize, titlesize, legendsize, labelsize

  Grid:
  gridlinewidth, gridlinestyle

  Ticks:
  num_xticks, num_yticks, tickwidth, ticklength, ticksize

  Data:
  xmin, xmax, ymin, ymax, spread_list # can append _list to any of these # these can be dictionaries if ydata was also a dict

  plot_kw_keys = ['label', 'color', 'marker', 'markersize',
  'markeredgewidth', 'linewidth', 'linestyle']

  any plot_kw key can be a scalar (corresponding to all ydatas), a list if ydata was specified as a list, or a dict if ydata was specified as a dict.

  kind = ['bar', 'plot', ...]

if kind='plot':
spread

if kind='bar':
stacked, width

References

matplotlib.org/examples/api/barchart_demo.html

Example

```
>>> xdata = [1, 2, 3, 4, 5]
>>> ydata_list = [[1, 2, 3, 4, 5], [3, 3, 3, 3, 3], [5, 4, np.nan, 2, 1], [4, 3, np.
↪nan, 1, 0]]
>>> kwargs = {'label': ['spam', 'eggs', 'jamμ', 'pram'],  'linestyle': '-'}
>>> #fig = multi_plot(xdata, ydata_list, title='$\phi_1(\\vec{x})$', xlabel='\nfds',␣
↪**kwargs)
>>> fig = multi_plot(xdata, ydata_list, title='μμμ', xlabel='\nfdsμμμ', **kwargs)
>>> show_if_requested()
```

Example

```
>>> fig1 = multi_plot([1, 2, 3], [4, 5, 6])
>>> fig2 = multi_plot([1, 2, 3], [4, 5, 6], fnum=4)
>>> show_if_requested()
```

graphid.util.mplutil.**figure**(*fnum=None*, *pnum=(1, 1, 1)*, *title=None*, *figtitle=None*, *doclf=False*, *docla=False*, *projection=None*, ***kwargs*)

http://matplotlib.org/users/gridspec.html

Parameters

- **fnum** (*int*) – fignum = figure number
- **pnum** (*int, str, or tuple(int, int, int)*) – plotnum = plot tuple
- **title** (*str*) – (default = None)
- **figtitle** (*None*) – (default = None)
- **docla** (*bool*) – (default = False)
- **doclf** (*bool*) – (default = False)

Returns
fig

Return type
mpl.Figure

Example

```
>>> import matplotlib.pyplot as plt
>>> fnum = 1
>>> fig = figure(fnum, (2, 2, 1))
>>> plt.gca().text(0.5, 0.5, "ax1", va="center", ha="center")
>>> fig = figure(fnum, (2, 2, 2))
>>> plt.gca().text(0.5, 0.5, "ax2", va="center", ha="center")
>>> show_if_requested()
```

Example

```
>>> import matplotlib.pyplot as plt
>>> fnum = 1
>>> fig = figure(fnum, (2, 2, 1))
>>> plt.gca().text(0.5, 0.5, "ax1", va="center", ha="center")
>>> fig = figure(fnum, (2, 2, 2))
>>> plt.gca().text(0.5, 0.5, "ax2", va="center", ha="center")
>>> fig = figure(fnum, (2, 4, (1, slice(1, None))))
>>> plt.gca().text(0.5, 0.5, "ax3", va="center", ha="center")
>>> show_if_requested()
```

graphid.util.mplutil.**pandas_plot_matrix**(*df*, *rot=90*, *ax=None*, *grid=True*, *label=None*, *zerodiag=False*, *cmap='viridis'*, *showvals=False*, *logscale=True*)

graphid.util.mplutil.**axes_extent**(*axs*, *pad=0.0*)

Get the full extent of a group of axes, including axes labels, tick labels, and titles.

graphid.util.mplutil.**extract_axes_extents**(*fig*, *combine=False*, *pad=0.0*)

graphid.util.mplutil.**adjust_subplots**(*left=None*, *right=None*, *bottom=None*, *top=None*, *wspace=None*, *hspace=None*, *fig=None*)

Kwargs:

left (float): left side of the subplots of the figure right (float): right side of the subplots of the figure bottom (float): bottom of the subplots of the figure top (float): top of the subplots of the figure wspace (float): width reserved for blank space between subplots hspace (float): height reserved for blank space between subplots

graphid.util.mplutil.**dict_intersection**(*dict1*, *dict2*)

Key AND Value based dictionary intersection

Parameters

- **dict1** (*dict*)
- **dict2** (*dict*)

Returns

mergedict_

Return type

dict

Example

```
>>> dict1 = {'a': 1, 'b': 2, 'c': 3, 'd': 4}
>>> dict2 = {'b': 2, 'c': 3, 'd': 5, 'e': 21, 'f': 42}
>>> mergedict_ = dict_intersection(dict1, dict2)
>>> print(ub.urepr(mergedict_, nl=0, sort=1))
{'b': 2, 'c': 3}
```

graphid.util.mplutil.**_dark_background**(*ax=None*, *doubleit=False*, *force=False*)

Parameters

- **ax** (*None*) – (default = None)
- **doubleit** (*bool*) – (default = False)

CommandLine

```
python -m .draw_func2 --exec-_dark_background --show
```

Example

```
>>> # ENABLE_DOCTEST
>>> fig = figure()
>>> _dark_background()
>>> show_if_requested()
```

graphid.util.mplutil.**_get_axis_xy_width_height**(*ax=None*, *xaug=0*, *yaug=0*, *waug=0*, *haug=0*)

gets geometry of a subplot

graphid.util.mplutil.**set_figtitle**(*figtitle*, *subtitle=''*, *forcefignum=True*, *incanvas=True*, *size=None*, *fontfamily=None*, *fontweight=None*, *fig=None*)

Parameters

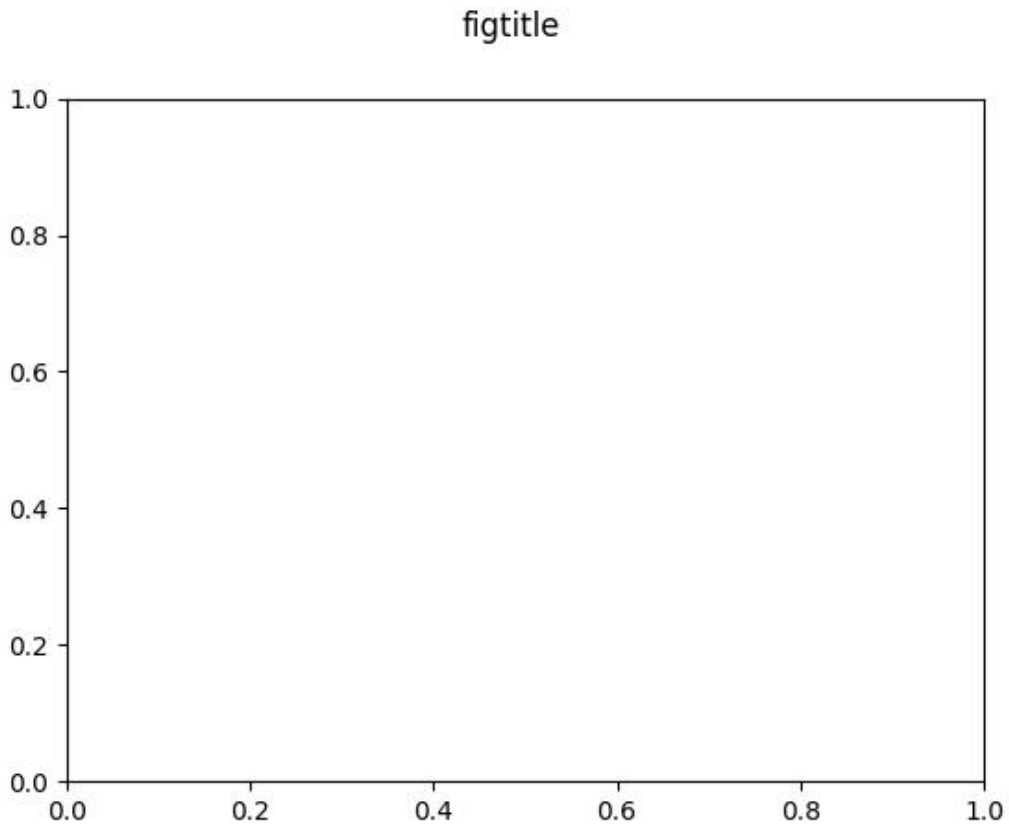
- **figtitle** (*?*)
- **subtitle** (*str*) – (default = '')
- **forcefignum** (*bool*) – (default = True)
- **incanvas** (*bool*) – (default = True)
- **fontfamily** (*None*) – (default = None)
- **fontweight** (*None*) – (default = None)
- **size** (*None*) – (default = None)
- **fig** (*None*) – (default = None)

CommandLine

```
python -m .custom_figure set_figtitle --show
```

Example

```
>>> # DISABLE_DOCTEST
>>> fig = figure(fnum=1, doclf=True)
>>> result = set_figtitle(figtitle='figtitle', fig=fig)
>>> # xdoc: +REQUIRES(--show)
>>> show_if_requested()
```



graphid.util.mplutil.**legend**(*loc='best'*, *fontproperties=None*, *size=None*, *fc='w'*, *alpha=1*, *ax=None*, *handles=None*)

Parameters

- **loc** (*str*) – (default = ‘best’)
- **fontproperties** (*None*) – (default = None)
- **size** (*None*) – (default = None)

graphid.util.mplutil.**distinct_colors**(*N*, *brightness=0.878*, *randomize=True*, *hue_range=(0.0, 1.0)*, *cmap_seed=None*)

Parameters

- **N** (*int*)
- **brightness** (*float*)

Returns
RGB_tuples

Return type
list

CommandLine

```
python -m color_funcs --test-distinct_colors --N 2 --show --hue-range=0.05,.95
python -m color_funcs --test-distinct_colors --N 3 --show --hue-range=0.05,.95
python -m color_funcs --test-distinct_colors --N 4 --show --hue-range=0.05,.95
python -m .color_funcs --test-distinct_colors --N 3 --show --no-randomize
python -m .color_funcs --test-distinct_colors --N 4 --show --no-randomize
python -m .color_funcs --test-distinct_colors --N 6 --show --no-randomize
python -m .color_funcs --test-distinct_colors --N 20 --show
```

References

http://blog.jianhuashao.com/2011/09/generate-n-distinct-colors.html

graphid.util.mplutil.**distinct_markers**(*num*, *style='astrisk'*, *total=None*, *offset=0*)

Parameters
num (*?*)

graphid.util.mplutil.**deterministic_shuffle**(*list_*, *rng=0*)

Parameters

- **list_** (*list*)
- **seed** (*int*)

Returns
list_

Return type
list

Example

```
>>> list_ = [1, 2, 3, 4, 5, 6]
>>> seed = 1
>>> list_ = deterministic_shuffle(list_, seed)
>>> result = str(list_)
>>> print(result)
[3, 2, 5, 1, 4, 6]
```

graphid.util.mplutil.**next_fnum**(*new_base=None*)

graphid.util.mplutil.**ensure_fnum**(*fnum*)

graphid.util.mplutil.**_save_requested**(*fpath_*, *save_parts*)

graphid.util.mplutil.**show_if_requested**(*N=1*)

Used at the end of tests. Handles command line arguments for saving figures

Referencse:
http://stackoverflow.com/questions/4325733/save-a-subplot-in-matplotlib

graphid.util.mplutil.**save_parts**(*fig*, *fpath*, *grouped_axes=None*, *dpi=None*)

FIXME: this works in mpl 2.0.0, but not 2.0.2

Parameters

- **fig** (*?*)
- **fpath** (*str*) – file path string
- **dpi** (*None*) – (default = None)

Returns
subpaths

Return type
list

CommandLine

```
python -m draw_func2 save_parts
```

graphid.util.mplutil.**qtensure**()

graphid.util.mplutil.**imshow**(*img*, *fnum=None*, *title=None*, *figtitle=None*, *pnum=None*, *interpolation='nearest'*, *cmap=None*, *heatmap=False*, *data_colorbar=False*, *xlabel=None*, *redraw_image=True*, *colorspace='bgr'*, *ax=None*, *alpha=None*, *norm=None*, ***kwargs*)

Parameters

- **img** (*ndarray*) – image data
- **fnum** (*int*) – figure number
- **colorspace** (*str*) – if the data is 3-4 channels, this indicates the colorspace 1 channel data is assumed grayscale. 4 channels assumes alpha.
- **title** (*str*)
- **figtitle** (*None*)
- **pnum** (*tuple*) – plot number
- **interpolation** (*str*) – other interpolations = nearest, bicubic, bilinear
- **cmap** (*None*)
- **heatmap** (*bool*)
- **data_colorbar** (*bool*)
- **darken** (*None*)

- **redraw_image** (*bool*) – used when calling imshow over and over. if false doesnt do the image part.

Returns
(fig, ax)

Return type
tuple

Kwargs:
docla, doclf, projection

Returns
(fig, ax)

Return type
tuple

graphid.util.mplutil.**colorbar**(*scalars*, *colors*, *custom=False*, *lbl=None*, *ticklabels=None*, *float_format='%.2f'*, ***kwargs*)

adds a color bar next to the axes based on specific scalars

Parameters

- **scalars** (*ndarray*)
- **colors** (*ndarray*)
- **custom** (*bool*) – use custom ticks

Kwargs:
See plt.colorbar

Returns
matplotlib colorbar object

Return type
cb

graphid.util.mplutil.**_get_plotdat**(*ax*, *key*, *default=None*)

returns internal property from a matplotlib axis

graphid.util.mplutil.**_set_plotdat**(*ax*, *key*, *val*)

sets internal property to a matplotlib axis

graphid.util.mplutil.**_del_plotdat**(*ax*, *key*)

sets internal property to a matplotlib axis

graphid.util.mplutil.**_get_plotdat_dict**(*ax*)

sets internal property to a matplotlib axis

graphid.util.mplutil.**_ensure_divider**(*ax*)

Returns previously constructed divider or creates one

graphid.util.mplutil.**scores_to_cmap**(*scores*, *colors=None*, *cmap_='hot'*)

graphid.util.mplutil.**scores_to_color**(*score_list*, *cmap_='hot'*, *logscale=False*, *reverse_cmap=False*, *custom=False*, *val2_customcolor=None*, *score_range=None*, *cmap_range=(0.1, 0.9)*)

Other good colormaps are ‘spectral’, ‘gist_rainbow’, ‘gist_ncar’, ‘Set1’, ‘Set2’, ‘Accent’ # TODO: plasma

Parameters

- **score_list** (*list*)
- **cmap_** (*str*) – defaults to hot
- **logscale** (*bool*)
- **cmap_range** (*tuple*) – restricts to only a portion of the cmap to avoid extremes

Returns

<class ‘_ast.ListComp’>

graphid.util.mplutil.**reverse_colormap**(*cmap*)

References

http://nbviewer.ipython.org/github/kwinkunks/notebooks/blob/master/Matteo_colourmaps.ipynb

class graphid.util.mplutil.**PlotNums**(*nRows=None*, *nCols=None*, *nSubplots=None*, *start=0*)

Bases: `object`

Convinience class for dealing with plot numberings (pnums)

Example

```
>>> pnum_ = PlotNums(nRows=2, nCols=2)
>>> # Indexable
>>> print(pnum_[0])
(2, 2, 1)
>>> # Iterable
>>> print(ub.urepr(list(pnum_), nl=0, nobr=1))
(2, 2, 1), (2, 2, 2), (2, 2, 3), (2, 2, 4)
>>> # Callable (iterates through a default iterator)
>>> print(pnum_())
(2, 2, 1)
>>> print(pnum_())
(2, 2, 2)
```

classmethod _get_num_rc(*nSubplots=None*, *nRows=None*, *nCols=None*)

Gets a constrained row column plot grid

Parameters

- **nSubplots** (*None*) – (default = None)
- **nRows** (*None*) – (default = None)
- **nCols** (*None*) – (default = None)

Returns

(nRows, nCols)

Return type
tuple

Example

```
>>> cases = [
>>>     dict(nRows=None, nCols=None, nSubplots=None),
>>>     dict(nRows=2, nCols=None, nSubplots=5),
>>>     dict(nRows=None, nCols=2, nSubplots=5),
>>>     dict(nRows=None, nCols=None, nSubplots=5),
>>> ]
>>> for kw in cases:
>>>     print('----')
>>>     size = PlotNums._get_num_rc(**kw)
>>>     if kw['nSubplots'] is not None:
>>>         assert size[0] * size[1] >= kw['nSubplots']
>>>     print('**kw = %s' % (ub.urepr(kw),))
>>>     print('size = %r' % (size,))
```

_get_square_row_cols(*max_cols=None*, *fix=False*, *inclusive=True*)

Parameters

- **nSubplots** (*int*)
- **max_cols** (*int*)

Returns
(int, int)

Return type
tuple

Example

```
>>> nSubplots = 9
>>> nSubplots_list = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]
>>> max_cols = None
>>> rc_list = [PlotNums._get_square_row_cols(nSubplots, fix=True) for␣
↪nSubplots in nSubplots_list]
>>> print(repr(np.array(rc_list).T))
array([[1, 1, 2, 2, 2, 2, 3, 3, 3, 3, 3],
       [1, 2, 2, 2, 3, 3, 3, 3, 3, 4, 4]])
```

graphid.util.mplutil.**draw_border**(*ax*, *color*, *lw=2*, *offset=None*, *adjust=True*)
draws rectangle border around a subplot

graphid.util.mplutil.**draw_boxes**(*boxes*, *box_format='xywh'*, *color='blue'*, *labels=None*, *textkw=None*, *ax=None*)

Parameters

- **boxes** (*list*) – list of coordindates in xywh, tlbr, or cxywh format

- **box_format** (*str*) – specify how boxes are formated xywh is the top left x and y pixel width and height cxywh is the center xy pixel width and height tlbr is the top left xy and the bottom right xy
- **color** (*str*) – edge color of the boxes
- **labels** (*list*) – if specified, plots a text annotation on each box

Example

```
>>> qtensure()  # xdoc: +SKIP
>>> bboxes = [[.1, .1, .6, .3], [.3, .5, .5, .6]]
>>> col = draw_boxes(bboxes)
```

graphid.util.mplutil.**draw_line_segments**(*pts1*, *pts2*, *ax=None*, ***kwargs*)

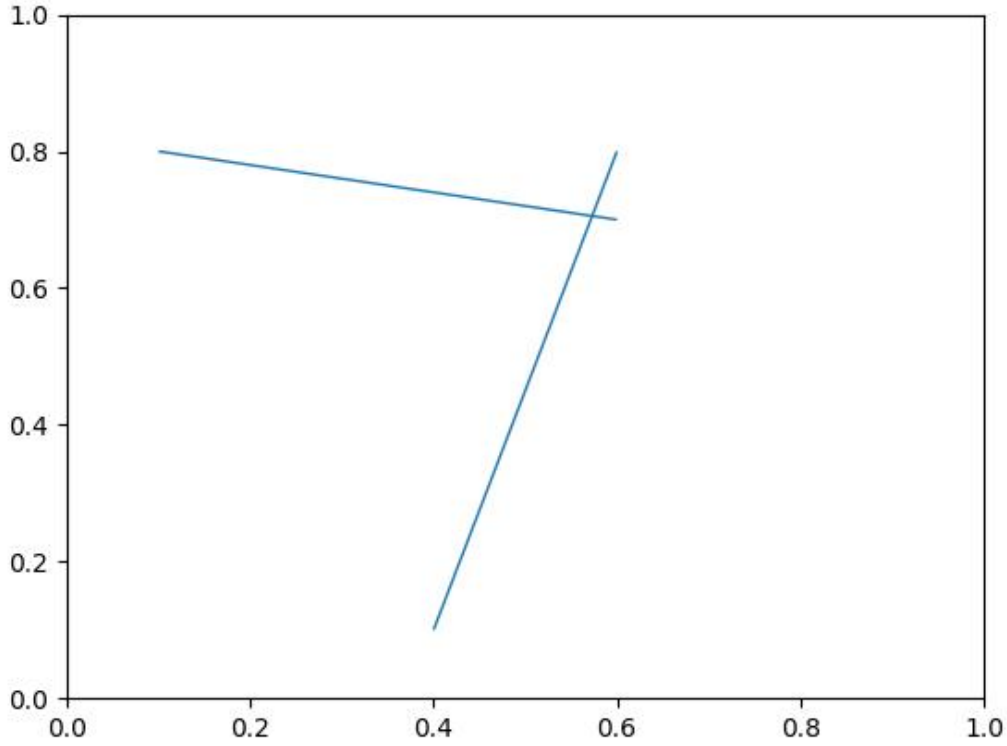
draws *N* line segments between *N* pairs of points

Parameters

- **pts1** (*ndarray*) – Nx2
- **pts2** (*ndarray*) – Nx2
- **ax** (*None*) – (default = None)
- ****kwargs** – lw, alpha, colors

Example

```
>>> pts1 = np.array([(.1, .8), (.6, .8)])
>>> pts2 = np.array([(.6, .7), (.4, .1)])
>>> figure(fnum=None)
>>> draw_line_segments(pts1, pts2)
>>> # xdoc: +REQUIRES(--show)
>>> import matplotlib.pyplot as plt
>>> ax = plt.gca()
>>> ax.set_xlim(0, 1)
>>> ax.set_ylim(0, 1)
>>> show_if_requested()
```



graphid.util.mplutil.**make_heatmask**(*probs*, *cmap='plasma'*, *with_alpha=True*)

Colorizes a single-channel intensity mask (with an alpha channel)

class graphid.util.mplutil.**Color**(*color*, *alpha=None*, *space=None*)

Bases: NiceRepr

move to colorutil?

Example

```
>>> print(Color('g'))
>>> print(Color('orangered'))
>>> print(Color('#AAAAAA').as255())
>>> print(Color([0, 255, 0]))
>>> print(Color([1, 1, 1.]))
>>> print(Color([1, 1, 1]))
>>> print(Color(Color([1, 1, 1])).as255())
>>> print(Color(Color([1., 0, 1, 0])).ashex())
>>> print(Color([1, 1, 1], alpha=255))
>>> print(Color([1, 1, 1], alpha=255, space='lab'))
```

ashex(*space=None*)

as255(*space=None*)

as01(*space=None*)

self = mplutil.Color('red') mplutil.Color('green').as01('rgba')

classmethod _is_base01()

check if a color is in base 01

classmethod _is_base255(*channels*)

there is a one corner case where all pixels are 1 or less

classmethod _hex_to_01(*hex_color*)

hex_color = '#6A5AFFAF'

_ensure_color01(*color*)

Infer what type color is and normalize to 01

classmethod _255_to_01(*color255*)

converts base 255 color to base 01 color

classmethod _string_to_01(*'green'*)

classmethod _string_to_01(*'red'*) → None

classmethod named_colors()

classmethod distinct(*num*, *space='bgr'*)

Make multiple distinct colors

adjust_hsv(*hue_adjust=0.0*, *sat_adjust=0.0*, *val_adjust=0.0*)

Performs adaptive changes to the HSV values of the color.

Parameters

- **hue_adjust** (*float*) – addative
- **sat_adjust** (*float*)
- **val_adjust** (*float*)

Returns

new_rgb

Return type

list

CommandLine

```
python -m graphid.util.mplutil Color.adjust_hsv
```

Example

```
>>> rgb_list = [Color(c).as01() for c in ['pink', 'yellow', 'green']]
>>> hue_adjust = -0.1
>>> sat_adjust = +0.5
>>> val_adjust = -0.1
>>> # execute function
>>> new_rgb_list = [Color(rgb).adjust_hsv(hue_adjust, sat_adjust, val_adjust)␣
```

(continues on next page)

(continued from previous page)

```
↪for rgb in rgb_list]
>>> print(ub.urepr(new_rgb_list, nl=1, sv=True))
[
    <Color(rgb: 0.90, 0.23, 0.75)>,
    <Color(rgb: 0.90, 0.36, 0.00)>,
    <Color(rgb: 0.24, 0.40, 0.00)>,
]
>>> # xdoc: +REQUIRES(--show)
>>> color_list = rgb_list + new_rgb_list
>>> testshow_colors(color_list)
```

convert(*space*)

Converts to a new colorspace

graphid.util.mplutil.**zoom_factory**(*ax=None*, *zoomable_list=[]*, *base_scale=1.1*)

References

https://gist.github.com/tacaswell/3144287 http://stackoverflow.com/questions/11551049/matplotlib-plot-zooming-with-scroll-wheel

graphid.util.mplutil.**pan_factory**(*ax=None*)

class graphid.util.mplutil.**PanEvents**(*ax=None*)

Bases: object

pan_on_press(*event*)

pan_on_release(*event*)

pan_on_motion(*event*)

graphid.util.mplutil.**relative_text**(*pos*, *text*, *ax=None*, *offset=None*, ***kwargs*)

Places text on axes in a relative position

Parameters

- **pos** (*tuple*) – relative xy position
- **text** (*str*) – text
- **ax** (*None*) – (default = None)
- **offset** (*None*) – (default = None)
- ****kwargs** – horizontalalignment, verticalalignment, roffset, ha, va, fontsize, fontproperties, fontproperties, clip_on

CommandLine

```
python -m graphid.util.mplutil relative_text --show
```

Example

```
>>> from graphid import util
>>> import matplotlib as mpl
>>> x = .5
>>> y = .5
>>> util.figure()
>>> txt = 'Hello World'
>>> family = 'monospace'
>>> family = 'CMU Typewriter Text'
>>> fontproperties = mpl.font_manager.FontProperties(family=family,
>>>                                                  size=42)
>>> relative_text((x, y), txt, halign='center',
>>>               fontproperties=fontproperties)
>>> util.show_if_requested()
```

graphid.util.mplutil.**get_axis_xy_width_height**(*ax=None*, *xaug=0*, *yaug=0*, *waug=0*, *haug=0*)

gets geometry of a subplot

1.1.3.1.3 graphid.util.name_rectifier module

graphid.util.name_rectifier.**demodata_oldnames**(*n_incon_groups=10*, *n_con_groups=2*, *n_per_con=5*, *n_per_incon=5*, *con_sep=4*, *n_empty_groups=0*)

graphid.util.name_rectifier.**simple_munkres**(*part_oldnames*)

Defines a munkres problem to solve name rectification.

Notes

We create a matrix where each rows represents a group of annotations in the same PCC and each column represents an original name. If there are more PCCs than original names the columns are padded with extra values. The matrix is first initialized to be negative infinity representing impossible assignments. Then for each column representing a padded name, we set we its value to 1 indicating that each new name could be assigned to a padded name for some small profit. Finally, let f_{rc} be the the number of annotations in row r with an original name of c. Each matrix value (r, c) is set to $f_{rc} + 1$ if $f_{rc} > 0$, to represent how much each name ``wants" to be labeled with a particular original name, and the extra one ensures that these original names are always preferred over padded names.

Example

```
>>> part_oldnames = [['a', 'b'], ['b', 'c'], ['c', 'a', 'a']]
>>> new_names = simple_munkres(part_oldnames)
>>> result = ub.urepr(new_names)
>>> print(new_names)
['b', 'c', 'a']
```

Example

```
>>> part_oldnames = [[], ['a', 'a'], [],
>>>                  ['a', 'a', 'a', 'a', 'a', 'a', 'a', 'b'], ['a']]
>>> new_names = simple_munkres(part_oldnames)
>>> result = ub.urepr(new_names)
>>> print(new_names)
[None, 'a', None, 'b', None]
```

Example

```
>>> part_oldnames = [[], ['b'], ['a', 'b', 'c'], ['b', 'c'], ['c', 'e', 'e']]
>>> new_names = find_consistent_labeling(part_oldnames)
>>> result = ub.urepr(new_names)
>>> print(new_names)
['_extra_name0', 'b', 'a', 'c', 'e']
```

Profit Matrix

b a c e _0

0 -10 -10 -10 -10 1 1 2 -10 -10 -10 1 2 2 2 2 -10 1 3 2 -10 2 -10 1 4 -10 -10 2 3 1

graphid.util.name_rectifier.**find_consistent_labeling**(*grouped_oldnames*, *extra_prefix='_extra_name'*, *verbose=False*)

Solves a a maximum bipirtite matching problem to find a consistent name assignment that minimizes the number of annotations with different names. For each new grouping of annotations we assign

For each group of annotations we must assign them all the same name, either from

To reduce the running time

Parameters

gropued_oldnames (*list*) – A group of old names where the grouping is based on new names. For instance:

Given:

aids = [1, 2, 3, 4, 5] old_names = [0, 1, 1, 1, 0] new_names = [0, 0, 1, 1, 0]

The grouping is

[[0, 1, 0], [1, 1]]

This lets us keep the old names in a split case and re-use exising names and make minimal changes to current annotation names while still being consistent with the new and improved grouping.

The output will be:
[0, 1]

Meaning that all annots in the first group are assigned the name 0 and all annots in the second group are assigned the name 1.

References

http://stackoverflow.com/questions/1398822/assignment-problem-numpy

Example

```
>>> grouped_oldnames = demodata_oldnames(25, 15,  5, n_per_incon=5)
>>> new_names = find_consistent_labeling(grouped_oldnames, verbose=1)
>>> grouped_oldnames = demodata_oldnames(0, 15,  5, n_per_incon=1)
>>> new_names = find_consistent_labeling(grouped_oldnames, verbose=1)
>>> grouped_oldnames = demodata_oldnames(0, 0, 0, n_per_incon=1)
>>> new_names = find_consistent_labeling(grouped_oldnames, verbose=1)
```

Example

```
>>> # xdoctest: +REQUIRES(module:timerit)
>>> import timerit
>>> ydata = []
>>> xdata = list(range(10, 150, 50))
>>> for x in xdata:
>>>     print('x = %r' % (x,))
>>>     grouped_oldnames = demodata_oldnames(x, 15,  5, n_per_incon=5)
>>>     t = timerit.Timerit(3, verbose=1)
>>>     for timer in t:
>>>         with timer:
>>>             new_names = find_consistent_labeling(grouped_oldnames)
>>>     ydata.append(t.min())
>>> # xdoc: +REQUIRES(--show)
>>> import plottool_ibeis as pt
>>> pt.qtensure()
>>> pt.multi_plot(xdata, [ydata])
>>> util.show_if_requested()
```

Example

```
>>> grouped_oldnames = [['a', 'b', 'c'], ['b', 'c'], ['c', 'e', 'e']]
>>> new_names = find_consistent_labeling(grouped_oldnames, verbose=1)
>>> result = ub.urepr(new_names)
>>> print(new_names)
['a', 'b', 'e']
```

Example

```
>>> grouped_oldnames = [['a', 'b'], ['a', 'a', 'b'], ['a']]
>>> new_names = find_consistent_labeling(grouped_oldnames)
>>> result = ub.urepr(new_names)
>>> print(new_names)
['b', 'a', '_extra_name0']
```

Example

```
>>> grouped_oldnames = [['a', 'b'], ['e'], ['a', 'a', 'b'], [], ['a'], ['d']]
>>> new_names = find_consistent_labeling(grouped_oldnames)
>>> result = ub.urepr(new_names)
>>> print(new_names)
['b', 'e', 'a', '_extra_name0', '_extra_name1', 'd']
```

Example

```
>>> grouped_oldnames = [[], ['a', 'a'], [],
>>>                     ['a', 'a', 'a', 'a', 'a', 'a', 'a', 'b'], ['a']]
>>> new_names = find_consistent_labeling(grouped_oldnames)
>>> result = ub.urepr(new_names)
>>> print(new_names)
['_extra_name0', 'a', '_extra_name1', 'b', '_extra_name2']
```

1.1.3.1.4 graphid.util.nx_dynamic_graph module

The main data structure for maintaining positive connected components and supporting dynamic addition and deletion of edges.

This uses a union-find-like algorithm (extended to support CC lookup) in the background, but could be implemented with another algorithm like Euler Tour Trees. UnionFind is good if you are mostly adding edges, but if you expect to remove edges a lot, then using a forest of ETTs may be better.

class graphid.util.nx_dynamic_graph.**GraphHelperMixin**

Bases: NiceRepr

Ensures that we always return edges in a consistent order

has_nodes(*nodes*)

has_edges(*edges*)

edges(*nbunch=None*, *data=False*, *default=None*)

class graphid.util.nx_dynamic_graph.**NiceGraph**(*incoming_graph_data=None*, ***attr*)

Bases: Graph, *GraphHelperMixin*

Initialize a graph with edges, name, or graph attributes.

Parameters

- **incoming_graph_data** (*input graph (optional, default: None)*) – Data to initialize graph. If None (default) an empty graph is created. The data can be an edge list, or any NetworkX graph object. If the corresponding optional Python packages are installed the data can also be a 2D NumPy array, a SciPy sparse array, or a PyGraphviz graph.
- **attr** (*keyword arguments, optional (default= no attributes)*) – Attributes to add to graph as key=value pairs.

See also:

`convert`

Examples

```
>>> G = nx.Graph()  # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G = nx.Graph(name="my graph")
>>> e = [(1, 2), (2, 3), (3, 4)]  # list of edges
>>> G = nx.Graph(e)
```

Arbitrary graph attribute pairs (key=value) may be assigned

```
>>> G = nx.Graph(e, day="Friday")
>>> G.graph
{'day': 'Friday'}
```

class graphid.util.nx_dynamic_graph.**nx_UnionFind**(*elements=None*)

Bases: `object`

Based off code in networkx

clear()

rebalance(*elements=None*)

to_sets()

union(**objects*)

Find the sets containing the objects and merge them all.

remove_entire_cc(*elements*)

add_element(*x*)

add_elements(*elements*)

class graphid.util.nx_dynamic_graph.**DynConnGraph**(**args*, ***kwargs*)

Bases: `Graph`, *GraphHelperMixin*

Dynamically connected graph.

Maintains a data structure parallel to a normal networkx graph that maintains dynamic connectivity for fast connected compoment queries.

Underlying Data Structures and limitations are

Data Structure | Insertion | Deletion | CC Find |

UnionFind | lg(n) | n | No UnionFind2 | n* | n | 1 EulerTourForest | lg^2(n) | lg^2(n) | lg(n) / lglg(n) - - Ammortized

- O(n) is worst case, but it seems to be very quick in practice. The average runtime should be close to lg(n), but I'm writing this doc 8 months after working on this algo, so I may not remember exactly.

References

https://courses.csail.mit.edu/6.851/spring14/lectures/L20.pdf https://courses.csail.mit.edu/6.851/spring14/lectures/L20.html http://cs.stackexchange.com/questions/33595/maintaining-connecte https://en.wikipedia.org/wiki/Dynamic_connectivity#Fully_dynamic_connectivity

Example

```
>>> self = DynConnGraph()
>>> self.add_edges_from([(1, 2), (2, 3), (4, 5), (6, 7), (7, 4)])
>>> self.add_edges_from([(10, 20), (20, 30), (40, 50), (60, 70), (70, 40)])
>>> self._ccs
>>> u, v = 20, 1
>>> assert self.node_label(u) != self.node_label(v)
>>> assert self.connected_to(u) != self.connected_to(v)
>>> self.add_edge(u, v)
>>> assert self.node_label(u) == self.node_label(v)
>>> assert self.connected_to(u) == self.connected_to(v)
>>> self.remove_edge(u, v)
>>> assert self.node_label(u) != self.node_label(v)
>>> assert self.connected_to(u) != self.connected_to(v)
>>> ccs = list(self.connected_components())
>>> # xdoctest: +REQUIRES(--show)
>>> import plottool_ibeis as pt
>>> pt.qtensure()
>>> pt.show_nx(self)
```

todo: check if nodes exist when adding

clear()

number_of_components()

component(*label*)

component_nodes(*label*)

connected_to(*node*)

node_label(*node*)

Example

```
>>> self = DynConnGraph()
>>> self.add_edges_from([(1, 2), (2, 3), (4, 5), (6, 7)])
>>> assert self.node_label(2) == self.node_label(1)
>>> assert self.node_label(2) != self.node_label(4)
```

node_labels(**nodes*)

are_nodes_connected(*u*, *v*)

connected_components()

Example

```
>>> self = DynConnGraph()
>>> self.add_edges_from([(1, 2), (2, 3), (4, 5), (6, 7)])
>>> ccs = list(self.connected_components())
>>> result = 'ccs = {}'.format(ub.urepr(ccs, nl=0))
>>> print(result)
ccs = [{1, 2, 3}, {4, 5}, {6, 7}]
```

component_labels()

_cut(*u*, *v*)

Decremental connectivity (slow)

_union(*u*, *v*)

Incremental connectivity (fast)

_add_node(*n*)

_remove_node(*n*)

add_edge(*u*, *v*, ***attr*)

Example

```
>>> self = DynConnGraph()
>>> self.add_edges_from([(1, 2), (2, 3), (4, 5), (6, 7), (7, 4)])
>>> assert self._ccs == {1: {1, 2, 3}, 4: {4, 5, 6, 7}}
>>> self.add_edge(1, 5)
>>> assert self._ccs == {1: {1, 2, 3, 4, 5, 6, 7}}
```

add_edges_from(*ebunch*, ***attr*)

add_node(*n*, ***attr*)

add_nodes_from(*nodes*, ***attr*)

remove_edge(*u*, *v*)

Example

```
>>> self = DynConnGraph()
>>> self.add_edges_from([(1, 2), (2, 3), (4, 5), (6, 7), (7, 4)])
>>> assert self._ccs == {1: {1, 2, 3}, 4: {4, 5, 6, 7}}
>>> self.add_edge(1, 5)
>>> assert self._ccs == {1: {1, 2, 3, 4, 5, 6, 7}}
>>> self.remove_edge(1, 5)
>>> assert self._ccs == {1: {1, 2, 3}, 4: {4, 5, 6, 7}}
```

remove_edges_from(*ebunch*)

remove_nodes_from(*nodes*)

remove_node(*n*)

Example

```
>>> self = DynConnGraph()
>>> self.add_edges_from([(1, 2), (2, 3), (4, 5), (5, 6), (6, 7), (7, 8), (8,␣
↪9)])
>>> assert self._ccs == {1: {1, 2, 3}, 4: {4, 5, 6, 7, 8, 9}}
>>> self.remove_node(2)
>>> assert self._ccs == {1: {1}, 3: {3}, 4: {4, 5, 6, 7, 8, 9}}
>>> self.remove_node(7)
>>> assert self._ccs == {1: {1}, 3: {3}, 4: {4, 5, 6}, 8: {8, 9}}
```

subgraph(*nbunch*, *dynamic=False*)

1.1.3.1.5 graphid.util.nx_utils module

graphid.util.nx_utils.**_dz**(*a*, *b*)

graphid.util.nx_utils.**nx_source_nodes**(*graph*)

graphid.util.nx_utils.**nx_sink_nodes**(*graph*)

graphid.util.nx_utils.**take_column**(*list_*, *colx*)

accepts a list of (indexables) and returns a list of indexables can also return a list of list of indexables if colx is a list

Parameters

- **list_** (*list*) – list of lists
- **colx** (*int or list*) – index or key in each sublist get item

Returns

list of selected items

Return type

list

Example0:

```
>>> list_ = [['a', 'b'], ['c', 'd']]
>>> colx = 0
>>> result = take_column(list_, colx)
>>> result = ub.urepr(result, nl=False)
>>> print(result)
['a', 'c']
```

Example1:

```
>>> list_ = [['a', 'b'], ['c', 'd']]
>>> colx = [1, 0]
>>> result = take_column(list_, colx)
>>> result = ub.urepr(result, nl=False)
>>> print(result)
[['b', 'a'], ['d', 'c']]
```

Example2:

```
>>> list_ = [{'spam': 'EGGS', 'ham': 'SPAM'}, {'spam': 'JAM', 'ham': 'PRAM'}]
>>> # colx can be a key or list of keys as well
>>> colx = ['spam']
>>> result = take_column(list_, colx)
>>> result = ub.urepr(result, nl=False)
>>> print(result)
[['EGGS'], ['JAM']]
```

graphid.util.nx_utils.**dict_take_column**(*list_of_dicts_*, *colkey*, *default=None*)

graphid.util.nx_utils.**itake_column**(*list_*, *colx*)

iterator version of get_list_column

graphid.util.nx_utils.**list_roll**(*list_*, *n*)

Like numpy.roll for python lists

Parameters

- **list_** (*list*)
- **n** (*int*)

Return type

list

References

http://stackoverflow.com/questions/9457832/python-list-rotation

Example

```
>>> list_ = [1, 2, 3, 4, 5]
>>> n = 2
>>> result = list_roll(list_, n)
>>> print(result)
[4, 5, 1, 2, 3]
```

graphid.util.nx_utils.**diag_product**(*s1*, *s2*)

Does product, but iterates over the diagonal first

graphid.util.nx_utils.**e_**(*u*, *v*)

graphid.util.nx_utils.**edges_inside**(*graph*, *nodes*)

Finds edges within a set of nodes Running time is O(len(nodes) ** 2)

Parameters

- **graph** (*nx.Graph*) – an undirected graph
- **nodes1** (*set*) – a set of nodes

graphid.util.nx_utils.**edges_outgoing**(*graph*, *nodes*)

Finds edges leaving a set of nodes. Average running time is O(len(nodes) * ave_degree(nodes)) Worst case running time is O(G.number_of_edges()).

Parameters

- **graph** (*nx.Graph*) – a graph
- **nodes** (*set*) – set of nodes

Example

```
>>> G = demodata_bridge()
>>> nodes = {1, 2, 3, 4}
>>> outgoing = edges_outgoing(G, nodes)
>>> assert outgoing == {(3, 5), (4, 8)}
```

graphid.util.nx_utils.**edges_cross**(*graph*, *nodes1*, *nodes2*)

Finds edges between two sets of disjoint nodes. Running time is O(len(nodes1) * len(nodes2))

Parameters

- **graph** (*nx.Graph*) – an undirected graph
- **nodes1** (*set*) – set of nodes disjoint from *nodes2*
- **nodes2** (*set*) – set of nodes disjoint from *nodes1*.

graphid.util.nx_utils.**edges_between**(*graph*, *nodes1*, *nodes2=None*, *assume_disjoint=False*, *assume_dense=True*)

Get edges between two components or within a single component

Parameters

- **graph** (*nx.Graph*) – the graph
- **nodes1** (*set*) – list of nodes

- **nodes2** (*set*) – if None it is equivlanet to nodes2=nodes1 (default=None)
- **assume_disjoint** (*bool*) – skips expensive check to ensure edges arnt returned twice (default=False)

Example

```
>>> edges = [
>>>     (1, 2), (2, 3), (3, 4), (4, 1), (4, 3),  # cc 1234
>>>     (1, 5), (7, 2), (5, 1),  # cc 567 / 5678
>>>     (7, 5), (5, 6), (8, 7),
>>> ]
>>> digraph = nx.DiGraph(edges)
>>> graph = nx.Graph(edges)
>>> nodes1 = [1, 2, 3, 4]
>>> nodes2 = [5, 6, 7]
>>> n2 = sorted(edges_between(graph, nodes1, nodes2))
>>> n4 = sorted(edges_between(graph, nodes1))
>>> n5 = sorted(edges_between(graph, nodes1, nodes1))
>>> n1 = sorted(edges_between(digraph, nodes1, nodes2))
>>> n3 = sorted(edges_between(digraph, nodes1))
>>> print('n2 == %r' % (n2,))
>>> print('n4 == %r' % (n4,))
>>> print('n5 == %r' % (n5,))
>>> print('n1 == %r' % (n1,))
>>> print('n3 == %r' % (n3,))
>>> assert n2 == ([(1, 5), (2, 7)]), '2'
>>> assert n4 == ([(1, 2), (1, 4), (2, 3), (3, 4)]), '4'
>>> assert n5 == ([(1, 2), (1, 4), (2, 3), (3, 4)]), '5'
>>> assert n1 == ([(1, 5), (5, 1), (7, 2)]), '1'
>>> assert n3 == ([(1, 2), (2, 3), (3, 4), (4, 1), (4, 3)]), '3'
>>> n6 = sorted(edges_between(digraph, nodes1 + [6], nodes2 + [1, 2], assume_
↪dense=False))
>>> print('n6 = %r' % (n6,))
>>> n6 = sorted(edges_between(digraph, nodes1 + [6], nodes2 + [1, 2], assume_
↪dense=True))
>>> print('n6 = %r' % (n6,))
>>> assert n6 == ([(1, 2), (1, 5), (2, 3), (4, 1), (5, 1), (5, 6), (7, 2)]), '6'
```

graphid.util.nx_utils.**_edges_between_dense**(*graph*, *nodes1*, *nodes2=None*, *assume_disjoint=False*)

The dense method is where we enumerate all possible edges and just take the ones that exist (faster for very dense graphs)

graphid.util.nx_utils.**_edges_inside_lower**(*graph*, *both_adj*)

finds lower triangular edges inside the nodes

graphid.util.nx_utils.**_edges_inside_upper**(*graph*, *both_adj*)

finds upper triangular edges inside the nodes

graphid.util.nx_utils.**_edges_between_disjoint**(*graph*, *only1_adj*, *only2*)

finds edges between disjoint nodes

graphid.util.nx_utils.**_edges_between_sparse**(*graph*, *nodes1*, *nodes2=None*, *assume_disjoint=False*)

In this version we check the intersection of existing edges and the edges in the second set (faster for sparse graphs)

graphid.util.nx_utils.**group_name_edges**(*g*, *node_to_label*)

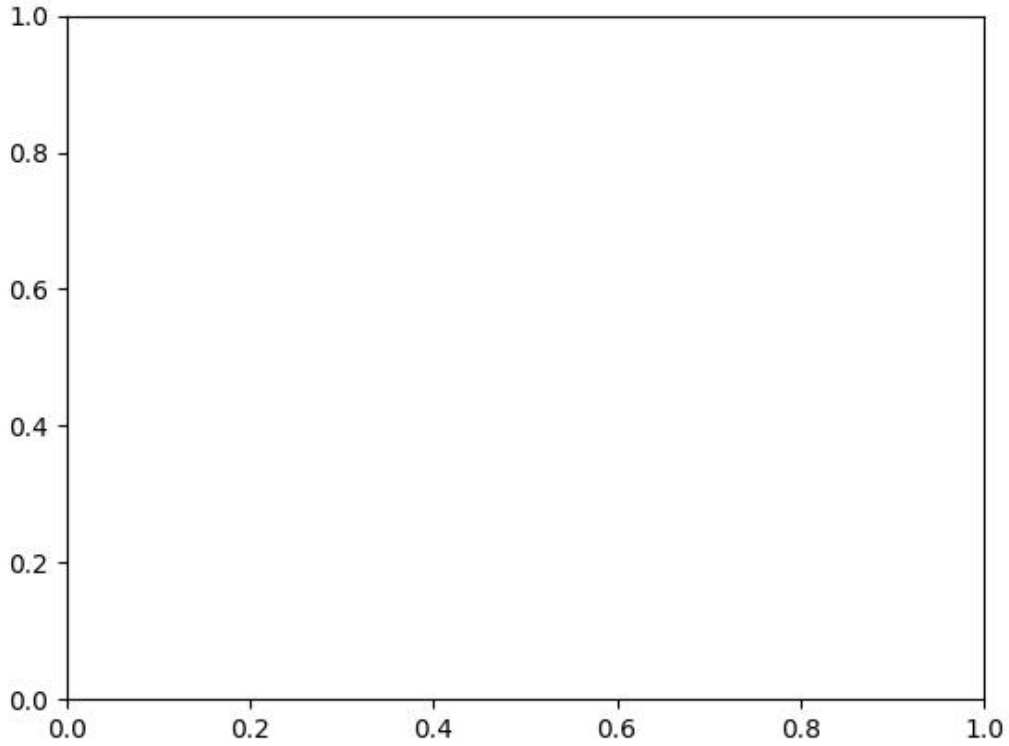
graphid.util.nx_utils.**ensure_multi_index**(*index*, *names*)

graphid.util.nx_utils.**demodata_bridge**()

graphid.util.nx_utils.**demodata_tarjan_bridge**()

Example

```
>>> from graphid import util
>>> G = demodata_tarjan_bridge()
>>> # xdoc: +REQUIRES(--show)
>>> util.show_nx(G)
>>> util.show_if_requested()
```



graphid.util.nx_utils.**is_k_edge_connected**(*G*, *k*)

graphid.util.nx_utils.**complement_edges**(*G*)

graphid.util.nx_utils.**k_edge_augmentation**(*G*, *k*, *avail=None*, *partial=False*)

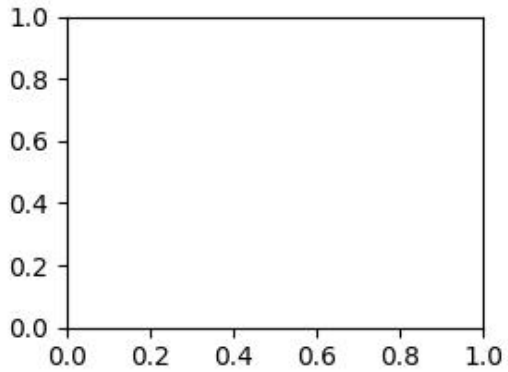
graphid.util.nx_utils.**is_complete**(*G*, *self_loops=False*)

graphid.util.nx_utils.**random_k_edge_connected_graph**(*size*, *k*, *p=0.1*, *rng=None*)

Super hacky way of getting a random k-connected graph

Example

```
>>> from graphid import util
>>> size, k, p = 25, 3, .1
>>> rng = util.ensure_rng(0)
>>> gs = []
>>> for x in range(4):
>>>     G = random_k_edge_connected_graph(size, k, p, rng)
>>>     gs.append(G)
>>> # xdoc: +REQUIRES(--show)
>>> pnum_ = util.PlotNums(nRows=2, nSubplots=len(gs))
>>> fnum = 1
>>> for g in gs:
>>>     util.show_nx(g, fnum=fnum, pnum=pnum_())
```



graphid.util.nx_utils.**edge_df**(*graph*, *edges*, *ignore=None*)

graphid.util.nx_utils.**nx_delete_node_attr**(*graph*, *name*, *nodes=None*)

Removes node attributes

Doctest

```
>>> G = nx.karate_club_graph()
>>> nx.set_node_attributes(G, name='foo', values='bar')
>>> datas = nx.get_node_attributes(G, 'club')
>>> assert len(nx.get_node_attributes(G, 'club')) == 34
>>> assert len(nx.get_node_attributes(G, 'foo')) == 34
>>> nx_delete_node_attr(G, ['club', 'foo'], nodes=[1, 2])
>>> assert len(nx.get_node_attributes(G, 'club')) == 32
>>> assert len(nx.get_node_attributes(G, 'foo')) == 32
>>> nx_delete_node_attr(G, ['club'])
>>> assert len(nx.get_node_attributes(G, 'club')) == 0
>>> assert len(nx.get_node_attributes(G, 'foo')) == 32
```

graphid.util.nx_utils.**nx_delete_edge_attr**(*graph*, *name*, *edges=None*)

Removes an attributes from specific edges in the graph

Doctest

```
>>> G = nx.karate_club_graph()
>>> nx.set_edge_attributes(G, name='spam', values='eggs')
>>> nx.set_edge_attributes(G, name='foo', values='bar')
>>> assert len(nx.get_edge_attributes(G, 'spam')) == 78
>>> assert len(nx.get_edge_attributes(G, 'foo')) == 78
>>> nx_delete_edge_attr(G, ['spam', 'foo'], edges=[(1, 2)])
>>> assert len(nx.get_edge_attributes(G, 'spam')) == 77
>>> assert len(nx.get_edge_attributes(G, 'foo')) == 77
>>> nx_delete_edge_attr(G, ['spam'])
>>> assert len(nx.get_edge_attributes(G, 'spam')) == 0
>>> assert len(nx.get_edge_attributes(G, 'foo')) == 77
```

Doctest

```
>>> G = nx.MultiGraph()
>>> G.add_edges_from([(1, 2), (2, 3), (3, 4), (4, 5), (4, 5), (1, 2)])
>>> nx.set_edge_attributes(G, name='spam', values='eggs')
>>> nx.set_edge_attributes(G, name='foo', values='bar')
>>> assert len(nx.get_edge_attributes(G, 'spam')) == 6
>>> assert len(nx.get_edge_attributes(G, 'foo')) == 6
>>> nx_delete_edge_attr(G, ['spam', 'foo'], edges=[(1, 2, 0)])
>>> assert len(nx.get_edge_attributes(G, 'spam')) == 5
>>> assert len(nx.get_edge_attributes(G, 'foo')) == 5
>>> nx_delete_edge_attr(G, ['spam'])
>>> assert len(nx.get_edge_attributes(G, 'spam')) == 0
>>> assert len(nx.get_edge_attributes(G, 'foo')) == 5
```

graphid.util.nx_utils.**nx_gen_node_values**(*G*, *key*, *nodes*, *default=NoParam*)

Generates attributes values of specific nodes

graphid.util.nx_utils.**nx_gen_node_attrs**(*G*, *key*, *nodes=None*, *default=NoParam*, *on_missing='error'*, *on_keyerr='default'*)

Improved generator version of nx.get_node_attributes

Parameters

- **on_missing** (*str*) – Strategy for handling nodes missing from G. Can be {'error', 'default', 'filter'}. defaults to 'error'.
- **on_keyerr** (*str*) – Strategy for handling keys missing from node dicts. Can be {'error', 'default', 'filter'}. defaults to 'default' if default is specified, otherwise defaults to 'error'.

Notes

strategies are:

error - raises an error if key or node does not exist
default - returns node, but uses value specified by default
filter - skips the node

Example

```
>>> # ENABLE_DOCTEST
>>> from graphid import util
>>> G = nx.Graph([(1, 2), (2, 3)])
>>> nx.set_node_attributes(G, name='part', values={1: 'bar', 3: 'baz'})
>>> nodes = [1, 2, 3, 4]
>>> #
>>> assert len(list(nx_gen_node_attrs(G, 'part', default=None, on_missing='error',␣
↪on_keyerr='default'))) == 3
>>> assert len(list(nx_gen_node_attrs(G, 'part', default=None, on_missing='error',␣
↪on_keyerr='filter'))) == 2
>>> assert_raises(KeyError, list, nx_gen_node_attrs(G, 'part', on_missing='error',␣
↪on_keyerr='error'))
>>> #
>>> assert len(list(nx_gen_node_attrs(G, 'part', nodes, default=None, on_missing=
↪'filter', on_keyerr='default'))) == 3
>>> assert len(list(nx_gen_node_attrs(G, 'part', nodes, default=None, on_missing=
↪'filter', on_keyerr='filter'))) == 2
>>> assert_raises(KeyError, list, nx_gen_node_attrs(G, 'part', nodes, on_missing=
↪'filter', on_keyerr='error'))
>>> #
>>> assert len(list(nx_gen_node_attrs(G, 'part', nodes, default=None, on_missing=
↪'default', on_keyerr='default'))) == 4
>>> assert len(list(nx_gen_node_attrs(G, 'part', nodes, default=None, on_missing=
↪'default', on_keyerr='filter'))) == 2
>>> assert_raises(KeyError, list, nx_gen_node_attrs(G, 'part', nodes, on_missing=
↪'default', on_keyerr='error'))
```

Example

```
>>> # DISABLE_DOCTEST
>>> # ALL CASES
>>> from graphid import util
>>> G = nx.Graph([(1, 2), (2, 3)])
>>> nx.set_node_attributes(G, name='full', values={1: 'A', 2: 'B', 3: 'C'})
>>> nx.set_node_attributes(G, name='part', values={1: 'bar', 3: 'baz'})
>>> nodes = [1, 2, 3, 4]
>>> attrs = dict(nx_gen_node_attrs(G, 'full'))
>>> input_grid = {
>>>     'nodes': [None, (1, 2, 3, 4)],
>>>     'key': ['part', 'full'],
>>>     'default': [ub.NoParam, None],
>>> }
>>> inputs = util.all_dict_combinations(input_grid)
>>> kw_grid = {
>>>     'on_missing': ['error', 'default', 'filter'],
>>>     'on_keyerr': ['error', 'default', 'filter'],
>>> }
>>> kws = util.all_dict_combinations(kw_grid)
>>> for in_ in inputs:
>>>     for kw in kws:
>>>         kw2 = ub.dict_union(kw, in_)
>>>         #print(kw2)
>>>         on_missing = kw['on_missing']
>>>         on_keyerr = kw['on_keyerr']
>>>         if on_keyerr == 'default' and in_['default'] is ub.NoParam:
>>>             on_keyerr = 'error'
>>>         will_miss = False
>>>         will_keyerr = False
>>>         if on_missing == 'error':
>>>             if in_['key'] == 'part' and in_['nodes'] is not None:
>>>                 will_miss = True
>>>             if in_['key'] == 'full' and in_['nodes'] is not None:
>>>                 will_miss = True
>>>         if on_keyerr == 'error':
>>>             if in_['key'] == 'part':
>>>                 will_keyerr = True
>>>             if on_missing == 'default':
>>>                 if in_['key'] == 'full' and in_['nodes'] is not None:
>>>                     will_keyerr = True
>>>         want_error = will_miss or will_keyerr
>>>         gen = nx_gen_node_attrs(G, **kw2)
>>>         try:
>>>             attrs = list(gen)
>>>         except KeyError:
>>>             if not want_error:
>>>                 raise AssertionError('should not have errored')
>>>         else:
>>>             if want_error:
>>>                 raise AssertionError('should have errored')
```

graphid.util.nx_utils.**graph_info**(*graph*, *ignore=None*, *stats=False*, *verbose=False*)

graphid.util.nx_utils.**assert_raises**(*ex_type*, *func*, **args*, ***kwargs*)

Checks that a function raises an error when given specific arguments.

Parameters

- **ex_type** (*Exception*) – exception type
- **func** (*callable*) – live python function

Example

```
>>> ex_type = AssertionError
>>> func = len
>>> assert_raises(ex_type, assert_raises, ex_type, func, [])
>>> assert_raises(ValueError, [].index, 0)
```

graphid.util.nx_utils.**bfs_conditional**(*G*, *source*, *reverse=False*, *keys=True*, *data=False*, *yield_nodes=True*, *yield_if=None*, *continue_if=None*, *visited_nodes=None*, *yield_source=False*)

Produce edges in a breadth-first-search starting at source, but only return nodes that satisfiy a condition, and only iterate past a node if it satisfies a different condition.

conditions are callables that take (G, child, edge) and return true or false

Example

```
>>> import networkx as nx
>>> G = nx.Graph()
>>> G.add_edges_from([(1, 2), (1, 3), (2, 3), (2, 4)])
>>> continue_if = lambda G, child, edge: True
>>> result = list(bfs_conditional(G, 1, yield_nodes=False))
>>> print(result)
[(1, 2), (1, 3), (2, 1), (2, 3), (2, 4), (3, 1), (3, 2), (4, 2)]
```

Example

```
>>> import networkx as nx
>>> G = nx.Graph()
>>> continue_if = lambda G, child, edge: (child % 2 == 0)
>>> yield_if = lambda G, child, edge: (child % 2 == 1)
>>> G.add_edges_from([(0, 1), (1, 3), (3, 5), (5, 10),
>>>                   (4, 3), (3, 6),
>>>                   (0, 2), (2, 4), (4, 6), (6, 10)])
>>> result = list(bfs_conditional(G, 0, continue_if=continue_if,
>>>                                  yield_if=yield_if))
>>> print(result)
[1, 3, 5]
```

graphid.util.nx_utils.**nx_gen_edge_attrs**(*G*, *key*, *edges=None*, *default=NoParam*, *on_missing='error'*, *on_keyerr='default'*)

Improved generator version of nx.get_edge_attributes

Parameters

- **on_missing** (*str*) – Strategy for handling nodes missing from G. Can be {'error', 'default', 'filter'}. defaults to 'error'. is on_missing is not error, then we allow any edge even if the endpoints are not in the graph.
- **on_keyerr** (*str*) – Strategy for handling keys missing from node dicts. Can be {'error', 'default', 'filter'}. defaults to 'default' if default is specified, otherwise defaults to 'error'.

CommandLine

```
python -m graphid.util.nx_utils nx_gen_edge_attrs
```

Example

```
>>> from graphid import util
>>> from functools import partial
>>> G = nx.Graph([(1, 2), (2, 3), (3, 4)])
>>> nx.set_edge_attributes(G, name='part', values={(1, 2): 'bar', (2, 3): 'baz'})
>>> edges = [(1, 2), (2, 3), (3, 4), (4, 5)]
>>> func = partial(nx_gen_edge_attrs, G, 'part', default=None)
>>> #
>>> assert len(list(func(on_missing='error', on_keyerr='default'))) == 3
>>> assert len(list(func(on_missing='error', on_keyerr='filter'))) == 2
>>> util.assert_raises(KeyError, list, func(on_missing='error', on_keyerr='error'))
>>> #
>>> assert len(list(func(edges, on_missing='filter', on_keyerr='default'))) == 3
>>> assert len(list(func(edges, on_missing='filter', on_keyerr='filter'))) == 2
>>> util.assert_raises(KeyError, list, func(edges, on_missing='filter', on_keyerr=
↪'error'))
>>> #
>>> assert len(list(func(edges, on_missing='default', on_keyerr='default'))) == 4
>>> assert len(list(func(edges, on_missing='default', on_keyerr='filter'))) == 2
>>> util.assert_raises(KeyError, list, func(edges, on_missing='default', on_keyerr=
↪'error'))
```

graphid.util.nx_utils.**nx_gen_edge_values**(*G*, *key*, *edges=None*, *default=NoParam*, *on_missing='error'*, *on_keyerr='default'*)

Generates attributes values of specific edges

Parameters

- **on_missing** (*str*) – Strategy for handling nodes missing from G. Can be {'error', 'default'}. defaults to 'error'.
- **on_keyerr** (*str*) – Strategy for handling keys missing from node dicts. Can be {'error', 'default'}. defaults to 'default' if default is specified, otherwise defaults to 'error'.

graphid.util.nx_utils.**nx_edges**(*graph*, *keys=False*, *data=False*)

graphid.util.nx_utils.**nx_delete_None_edge_attr**(*graph*, *edges=None*)

graphid.util.nx_utils.**nx_delete_None_node_attr**(*graph*, *nodes=None*)

graphid.util.nx_utils.**nx_node_dict**(*G*)

1.1.3.1.6 graphid.util.priority_queue module

graphid.util.priority_queue.**_heappush_max**(*heap*, *item*)

why is this not in heapq

class graphid.util.priority_queue.**PriorityQueue**(*items=None*, *ascending=True*)

Bases: NiceRepr

abstracted priority queue for our needs

Combines properties of dicts and heaps Uses a heap for fast minimum/maximum value search Uses a dict for fast read only operations

References

http://code.activestate.com/recipes/522995-priority-dict-a-priority-queue-with-updatable-prio/ https://stackoverflow.com/questions/33024215/built-in-max-heap-api-in-python

Example

```
>>> items = dict(a=42, b=29, c=40, d=95, e=10)
>>> self = PriorityQueue(items)
>>> print(self)
>>> assert len(self) == 5
>>> print(self.pop())
>>> assert len(self) == 4
>>> print(self.pop())
>>> assert len(self) == 3
>>> print(self.pop())
>>> print(self.pop())
>>> print(self.pop())
>>> assert len(self) == 0
```

Example

```
>>> items = dict(a=(1.0, (2, 3)), b=(1.0, (1, 2)), c=(.9, (3, 2)))
>>> self = PriorityQueue(items)
```

_rebuild()

get(*key*, *default=None*)

clear()

update(*items*)

delete_items(*key_list*)

peek()

Peek at the next item in the queue

peek_many(*n*)

Actually this can be quite inefficient

Example

```
>>> from graphid import util
>>> items = list(zip(range(256), range(256)))
>>> n = 32
>>> util.shuffle(items)
>>> self = PriorityQueue(items, ascending=False)
>>> self.peek_many(56)
```

pop_many(*n*)

pop(*key=NoParam*, *default=NoParam*)

Pop the next item off the queue

1.1.3.1.7 graphid.util.util_boxes module

graphid.util.util_boxes.**box_ious_py**(*boxes1*, *boxes2*, *bias=1*)

This is the fastest python implementation of bbox_ious I found

class graphid.util.util_boxes.**Boxes**(*data*, *format='xywh'*)

Bases: NiceRepr

Converts boxes between different formats as long as the last dimension contains 4 coordinates and the format is specified.

This is a convinience class, and should not not store the data for very long. The general idiom should be create class, convert data, and then get the raw data and let the class be garbage collected. This will help ensure that your code is portable and understandable if this class is not available.

Example

```
>>> # xdoctest: +IGNORE_WHITESPACE
>>> Boxes([25, 30, 15, 10], 'xywh')
<Boxes(xywh, array([25, 30, 15, 10]))>
>>> Boxes([25, 30, 15, 10], 'xywh').to_xywh()
<Boxes(xywh, array([25, 30, 15, 10]))>
>>> Boxes([25, 30, 15, 10], 'xywh').to_cxywh()
<Boxes(cxywh, array([32.5, 35. , 15. , 10. ]))>
>>> Boxes([25, 30, 15, 10], 'xywh').to_tlbr()
<Boxes(tlbr, array([25, 30, 40, 40]))>
>>> Boxes([25, 30, 15, 10], 'xywh').scale(2).to_tlbr()
<Boxes(tlbr, array([50., 60., 80., 80.]))>
```

Example

```
>>> datas = [
>>>     [1, 2, 3, 4],
>>>     [[1, 2, 3, 4], [4, 5, 6, 7]],
>>>     [[[1, 2, 3, 4], [4, 5, 6, 7]]],
>>> ]
>>> formats = ['xywh', 'cxywh', 'tlbr']
>>> for format1 in formats:
>>>     for data in datas:
>>>         self = box1 = Boxes(data, format1)
>>>         for format2 in formats:
>>>             box2 = box1.toformat(format2)
>>>             back = box2.toformat(format1)
>>>             assert box1 == back
```

classmethod random(*num=1*, *scale=1.0*, *format='xywh'*, *rng=None*)

Makes random boxes

Example

```
>>> # xdoctest: +IGNORE_WHITESPACE
>>> Boxes.random(3, rng=0, scale=100)
<Boxes(xywh,
    array([[27, 35, 30, 27],
           [21, 32, 21, 44],
           [48, 19, 39, 26]]))>
```

copy()

scale(*factor*)

works with tlbr, cxywh, xywh, xy, or wh formats

Example

```
>>> # xdoctest: +IGNORE_WHITESPACE
>>> Boxes(np.array([1, 1, 10, 10])).scale(2).data
array([ 2.,  2., 20., 20.])
>>> Boxes(np.array([[1, 1, 10, 10]])).scale((2, .5)).data
array([[ 2. ,  0.5, 20. ,  5. ]])
>>> Boxes(np.array([[10, 10]])).scale(.5).data
array([[5., 5.]])
```

shift(*amount*)

Example

```
>>> # xdoctest: +IGNORE_WHITESPACE
>>> Boxes([25, 30, 15, 10], 'xywh').shift(10)
<Boxes(xywh, array([35., 40., 15., 10.]))>
>>> Boxes([25, 30, 15, 10], 'xywh').shift((10, 0))
<Boxes(xywh, array([35., 30., 15., 10.]))>
>>> Boxes([25, 30, 15, 10], 'tlbr').shift((10, 5))
<Boxes(tlbr, array([35., 35., 25., 15.]))>
```

property center

property shape

property area

property components

classmethod _cat(*datas*)

toformat(*format*, *copy=True*)

to_extent(*copy=True*)

to_xywh(*copy=True*)

to_cxywh(*copy=True*)

to_tlbr(*copy=True*)

clip(*x_min*, *y_min*, *x_max*, *y_max*, *inplace=False*)

Clip boxes to image boundaries. If box is in tlbr format, inplace operation is an option.

Example

```
>>> # xdoctest: +IGNORE_WHITESPACE
>>> boxes = Boxes(np.array([[-10, -10, 120, 120], [1, -2, 30, 50]]), 'tlbr')
>>> clipped = boxes.clip(0, 0, 110, 100, inplace=False)
>>> assert np.any(boxes.data != clipped.data)
>>> clipped2 = boxes.clip(0, 0, 110, 100, inplace=True)
>>> assert clipped2.data is boxes.data
>>> assert np.all(clipped2.data == clipped.data)
>>> print(clipped)
<Boxes(tlbr,
    array([[  0,   0, 110, 100],
           [  1,   0,  30,  50]]))>
```

transpose()

compress(*flags*, *axis=0*, *inplace=False*)

Filters boxes based on a boolean criterion

Example

```
>>> self = Boxes([[25, 30, 15, 10]], 'tlbr')
>>> flags = [False]
```

1.1.3.1.8 graphid.util.util_grabdata module

Ported from kwimage.

graphid.util.util_grabdata.**grab_test_imgpath**(*key='astro.png'*, *allow_external=True*, *verbose=True*)

Gets paths to standard / fun test images. Downloads them if they dont exits

Parameters

- **key** (*str*) – one of the standard test images, e.g. astro.png, carl.jpg, ...
- **allow_external** (*bool*) – if True you can specify existing fpaths

Returns

testimg_fpath - filepath to the downloaded or cached test image.

Return type

str

Example

```
>>> testimg_fpath = grab_test_imgpath('carl.jpg')
>>> assert exists(testimg_fpath)
```

graphid.util.util_grabdata.**_update_hashes**()

for dev use to update hashes of the demo images

graphid.util.util_grabdata.**_grabdata_with_mirrors**(*url*, *mirror_urls*, *grabkw*)

graphid.util.util_grabdata.**grab_test_image_fpath**(*key='astro'*, *dsize=None*, *overviews=None*)

Ensures that the test image exists (this might use the network) and returns the cached filepath to the requested image.

Parameters

- **key** (*str*) – which test image to grab. Valid choices are: astro - an astronaught carl - Carl Sagan paraview - ParaView logo stars - picture of stars in the sky
- **dsize** (*None | Tuple[int, int]*) – if specified, we will return a variant of the data with the specific dsize
- **overviews** (*None | int*) – if specified, will return a variant of the data with overviews

Returns

path to the requested image

Return type

str

1.1.3.1.9 graphid.util.util_graphviz module

Helpers for graph plotting

References

http://www.graphviz.org/content/attrs http://www.graphviz.org/doc/info/attrs.html

graphid.util.util_graphviz.**dump_nx_ondisk**(*graph*, *fpath*)

graphid.util.util_graphviz.**ensure_nonhex_color**(*orig_color*)

graphid.util.util_graphviz.**show_nx**(*graph*, *with_labels=True*, *fnum=None*, *pnum=None*, *layout='agraph'*, *ax=None*, *pos=None*, *img_dict=None*, *title=None*, *layoutkw=None*, *verbose=None*, ***kwargs*)

Parameters

- **graph** (*networkx.Graph*)
- **with_labels** (*bool*) – (default = True)
- **fnum** (*int*) – figure number(default = None)
- **pnum** (*tuple*) – plot number(default = None)
- **layout** (*str*) – (default = 'agraph')
- **ax** (*None*) – (default = None)
- **pos** (*None*) – (default = None)
- **img_dict** (*dict*) – (default = None)
- **title** (*str*) – (default = None)
- **layoutkw** (*None*) – (default = None)
- **verbose** (*bool*) – verbosity flag(default = None)

Kwargs:
use_image, framewidth, modify_ax, as_directed, hacknoedge, hacknode, arrow_width, fontsize, fontweight, fontname, fontfamilty, fontproperties

CommandLine

```
python -m graphid.util.util_graphviz show_nx --show
```

Example

```
>>> # xdoctest: +REQUIRES(module:pygraphviz)
>>> from graphid.util.util_graphviz import *  # NOQA
>>> graph = nx.DiGraph()
>>> graph.add_nodes_from(['a', 'b', 'c', 'd'])
>>> graph.add_edges_from({'a': 'b', 'b': 'c', 'b': 'd', 'c': 'd'}.items())
>>> nx.set_node_attributes(graph, name='shape', values='rect')
```

(continues on next page)

(continued from previous page)

```
>>> nx.set_node_attributes(graph, name='image', values={'a': util.grab_test_imgpath(
↪'carl.jpg')})
>>> nx.set_node_attributes(graph, name='image', values={'d': util.grab_test_imgpath(
↪'astro.png')})
>>> #nx.set_node_attributes(graph, name='height', values=100)
>>> with_labels = True
>>> fnum = None
>>> pnum = None
>>> e = show_nx(graph, with_labels, fnum, pnum, layout='agraph')
>>> util.show_if_requested()
```

graphid.util.util_graphviz.**netx_draw_images_at_positions**(*img_list*, *pos_list*, *size_list*, *color_list*, *framewidth_list*)

Overlays images on a networkx graph

References

https://gist.github.com/shobhit/3236373 http://matplotlib.org/examples/pylab_examples/demo_annotation_box.html http://stackoverflow.com/questions/11487797/mpl-overlay-small-image http://matplotlib.org/api/text_api.html http://matplotlib.org/api/offsetbox_api.html

graphid.util.util_graphviz.**parse_html_graphviz_attrs**()

class graphid.util.util_graphviz.**GRAPHVIZ_KEYS**

Bases: object

N = {'URL', 'area', 'color', 'colorscheme', 'comment', 'distortion', 'fillcolor', 'fixedsize', 'fontcolor', 'fontname', 'fontsize', 'gradientangle', 'group', 'height', 'href', 'id', 'image', 'imagepos', 'imagescale', 'label', 'labelloc', 'layer', 'margin', 'nojustify', 'ordering', 'orientation', 'penwidth', 'peripheries', 'pin', 'pos', 'rects', 'regular', 'root', 'samplepoints', 'shape', 'shapefile', 'showboxes', 'sides', 'skew', 'sortv', 'style', 'target', 'tooltip', 'vertices', 'width', 'xlabel', 'xlp', 'z'}

E = {'URL', 'arrowhead', 'arrowsize', 'arrowtail', 'color', 'colorscheme', 'comment', 'constraint', 'decorate', 'dir', 'edgeURL', 'edgehref', 'edgetarget', 'edgetooltip', 'fillcolor', 'fontcolor', 'fontname', 'fontsize', 'headURL', 'head_lp', 'headclip', 'headhref', 'headlabel', 'headport', 'headtarget', 'headtooltip', 'href', 'id', 'label', 'labelURL', 'labelangle', 'labeldistance', 'labelfloat', 'labelfontcolor', 'labelfontname', 'labelfontsize', 'labelhref', 'labeltarget', 'labeltooltip', 'layer', 'len', 'lhead', 'lp', 'ltail', 'minlen', 'nojustify', 'penwidth', 'pos', 'samehead', 'sametail', 'showboxes', 'style', 'tailURL', 'tail_lp', 'tailclip', 'tailhref', 'taillabel', 'tailport', 'tailtarget', 'tailtooltip', 'target', 'tooltip', 'weight', 'xlabel', 'xlp'}

```
G = {'Damping', 'K', 'URL', '_background', 'bb', 'bgcolor', 'center', 'charset',
'clusterrank', 'colorscheme', 'comment', 'compound', 'concentrate', 'defaultdist',
'dim', 'dimen', 'diredgeconstraints', 'dpi', 'epsilon', 'esep', 'fontcolor',
'fontname', 'fontnames', 'fontpath', 'fontsize', 'forcelabels', 'gradientangle',
'href', 'id', 'imagepath', 'inputscale', 'label', 'label_scheme', 'labeljust',
'labelloc', 'landscape', 'layerlistsep', 'layers', 'layerselect', 'layersep',
'layout', 'levels', 'levelsgap', 'lheight', 'lp', 'lwidth', 'margin', 'maxiter',
'mclimit', 'mindist', 'mode', 'model', 'mosek', 'newrank', 'nodesep', 'nojustify',
'normalize', 'notranslate', 'nslimit\nnslimit1', 'ordering', 'orientation',
'outputorder', 'overlap', 'overlap_scaling', 'overlap_shrink', 'pack', 'packmode',
'pad', 'page', 'pagedir', 'quadtree', 'quantum', 'rankdir', 'ranksep', 'ratio',
'remincross', 'repulsiveforce', 'resolution', 'root', 'rotate', 'rotation', 'scale',
'searchsize', 'sep', 'showboxes', 'size', 'smoothing', 'sortv', 'splines', 'start',
'style', 'stylesheet', 'target', 'truecolor', 'viewport', 'voro_margin',
'xdotversion'}
```

graphid.util.util_graphviz.**get_explicit_graph**(*graph*)

Parameters
graph (*nx.Graph*)

graphid.util.util_graphviz.**get_nx_layout**(*graph*, *layout*, *layoutkw=None*, *verbose=None*)

graphid.util.util_graphviz.**apply_graph_layout_attrs**(*graph*, *layout_info*)

graphid.util.util_graphviz.**patch_pygraphviz**()

Hacks around a python3 problem in 1.3.1 of pygraphviz

graphid.util.util_graphviz.**make_agraph**(*graph_*)

graphid.util.util_graphviz.**_groupby_prelayout**(*graph_*, *layoutkw*, *groupby*)

sets *pin* attr of *graph_* inplace in order to nodes according to specified layout.

graphid.util.util_graphviz.**nx_agraph_layout**(*orig_graph*, *inplace=False*, *verbose=None*, *return_agraph=False*, *groupby=None*, ***layoutkw*)

Uses graphviz and custom code to determine position attributes of nodes and edges.

Parameters
groupby (*str*) – if not None then nodes will be grouped by this attributes and groups will be layed out separately and then stacked together in a grid

References

http://www.graphviz.org/content/attrs http://www.graphviz.org/doc/info/attrs.html

CommandLine

```
python -m graphid.util.util_graphviz nx_agraph_layout --show
```

Doctest

```
>>> # xdoctest: +REQUIRES(module:pygraphviz)
>>> from graphid.util.util_graphviz import *  # NOQA
>>> import networkx as nx
>>> import itertools as it
>>> from graphid import util
>>> n, s = 9, 4
>>> offsets = list(range(0, (1 + n) * s, s))
>>> node_groups = [list(map(str, range(*o))) for o in ub.iter_window(offsets, 2)]
>>> edge_groups = [it.combinations(nodes, 2) for nodes in node_groups]
>>> graph = nx.Graph()
>>> [graph.add_nodes_from(nodes) for nodes in node_groups]
>>> [graph.add_edges_from(edges) for edges in edge_groups]
>>> for count, nodes in enumerate(node_groups):
...     nx.set_node_attributes(graph, name='id', values=ub.dzip(nodes, [count]))
>>> layoutkw = dict(prog='neato')
>>> graph1, info1 = nx_agraph_layout(graph.copy(), inplace=True, groupby='id',␣
↪**layoutkw)
>>> graph2, info2 = nx_agraph_layout(graph.copy(), inplace=True, **layoutkw)
>>> graph3, _ = nx_agraph_layout(graph1.copy(), inplace=True, **layoutkw)
>>> nx.set_node_attributes(graph1, name='pin', values='true')
>>> graph4, _ = nx_agraph_layout(graph1.copy(), inplace=True, **layoutkw)
>>> # xdoc: +REQUIRES(--show)
>>> util.show_nx(graph1, layout='custom', pnum=(2, 2, 1), fnum=1)
>>> util.show_nx(graph2, layout='custom', pnum=(2, 2, 2), fnum=1)
>>> util.show_nx(graph3, layout='custom', pnum=(2, 2, 3), fnum=1)
>>> util.show_nx(graph4, layout='custom', pnum=(2, 2, 4), fnum=1)
>>> util.show_if_requested()
>>> g1pos = nx.get_node_attributes(graph1, 'pos')['1']
>>> g4pos = nx.get_node_attributes(graph4, 'pos')['1']
>>> g2pos = nx.get_node_attributes(graph2, 'pos')['1']
>>> g3pos = nx.get_node_attributes(graph3, 'pos')['1']
>>> print('g1pos = {!r}'.format(g1pos))
>>> print('g4pos = {!r}'.format(g4pos))
>>> print('g2pos = {!r}'.format(g2pos))
>>> print('g3pos = {!r}'.format(g3pos))
>>> assert np.all(g1pos == g4pos), 'points between 1 and 4 were pinned so they␣
↪should be equal'
>>> #assert np.all(g2pos != g3pos), 'points between 2 and 3 were not pinned, so they␣
↪should be different'
```

assert np.all(nx.get_node_attributes(graph1, ‘pos’)[‘1’] == nx.get_node_attributes(graph4, ‘pos’)[‘1’]) assert np.all(nx.get_node_attributes(graph2, ‘pos’)[‘1’] == nx.get_node_attributes(graph3, ‘pos’)[‘1’])

graphid.util.util_graphviz.**parse_point**(*ptstr*)

graphid.util.util_graphviz.**parse_anode_layout_attrs**(*anode*)

graphid.util.util_graphviz.**parse_aedge_layout_attrs**(*aedge*, *translation=None*)

parse grpahviz splineType

graphid.util.util_graphviz.**_get_node_size**(*graph*, *node*, *node_size*)

graphid.util.util_graphviz.**draw_network2**(*graph*, *layout_info*, *ax*, *as_directed=None*, *hacknoedge=False*, *hacknode=False*, *verbose=None*, ***kwargs*)

Kwargs:
use_image, arrow_width, fontsize, fontweight, fontname, fontfamilty, fontproperties

fancy way to draw networkx graphs without directly using networkx

graphid.util.util_graphviz.**stack_graphs**(*graph_list*, *vert=False*, *pad=None*)

graphid.util.util_graphviz.**translate_graph**(*graph*, *t_xy*)

graphid.util.util_graphviz.**translate_graph_to_origin**(*graph*)

graphid.util.util_graphviz.**get_graph_bounding_box**(*graph*)

Example

```
>>> # xdoctest: +REQUIRES(module:pygraphviz)
>>> graph = nx.path_graph([1, 2, 3, 4])
>>> nx_agraph_layout(graph, inplace=True)
>>> bbox = get_graph_bounding_box(graph)
>>> print(ub.urepr(bbox, nl=0))
[0.0, 0.0, 54.0, 252.0]
```

graphid.util.util_graphviz.**nx_ensure_agraph_color**(*graph*)

changes colors to hex strings on graph attrs

graphid.util.util_graphviz.**bbox_from_extent**(*extent*)

Parameters
extent (*ndarray*) – tl_x, br_x, tl_y, br_y

Returns
tl_x, tl_y, w, h

Return type
bbox (ndarray)

Example

```
>>> extent = [0, 10, 0, 10]
>>> bbox = bbox_from_extent(extent)
>>> print('bbox = {}'.format(ub.urepr(list(bbox), nl=0)))
bbox = [0, 0, 10, 10]
```

graphid.util.util_graphviz.**get_pointset_extents**(*pts*)

1.1.3.1.10 graphid.util.util_group module

graphid.util.util_group.**sortedby**(*item_list*, *key_list*, *reverse=False*)

sorts `item_list` using key_list

Parameters

- **list_** (*list*) – list to sort
- **key_list** (*list*) – list to sort by
- **reverse** (*bool*) – sort order is descending (largest first) if reverse is True else acscending (smallest first)

Returns

`list_` sorted by the values of another `list`. defaults to ascending order

Return type

list

SeeAlso:

sortedby2

Examples

```
>>> list_    = [1, 2, 3, 4, 5]
>>> key_list = [2, 5, 3, 1, 5]
>>> result = sortedby(list_, key_list, reverse=True)
>>> print(result)
[5, 2, 3, 1, 4]
```

graphid.util.util_group.**grouping_delta**(*old*, *new*, *pure=True*)

Finds what happened to the old groups to form the new groups.

Parameters

- **old** (*set of frozensets*) – old grouping
- **new** (*set of frozensets*) – new grouping
- **pure** (*bool*) – hybrids are separated from pure merges and splits if pure is True, otherwise hybrid cases are grouped in merges and splits.

Returns

delta: dictionary of changes containing the merges, splits,
unchanged, and hybrid cases. Except for unchanged, case a subdict with new and old keys. For splits / merges, one of these contains nested sequences to indicate what the split / merge is. Also reports elements added and removed between old and new if the flattened sets are not the same.

Return type

dict

Notes

merges - which old groups were merged into a single new group. splits - which old groups were split into multiple new groups. hybrid - which old groups had split/merge actions applied. unchanged - which old groups are the same as new groups.

Example

```
>>> # xdoc: +IGNORE_WHITESPACE
>>> old = [
>>>     [20, 21, 22, 23], [1, 2], [12], [13, 14], [3, 4], [5, 6,11],
>>>     [7], [8, 9], [10], [31, 32], [33, 34, 35], [41, 42, 43, 44, 45]
>>> ]
>>> new = [
>>>   [20, 21], [22, 23], [1, 2], [12, 13, 14], [4], [5, 6, 3], [7, 8],
>>>   [9, 10, 11], [31, 32, 33, 34, 35],   [41, 42, 43, 44], [45],
>>> ]
>>> delta = grouping_delta(old, new)
>>> assert set(old[0]) in delta['splits']['old']
>>> assert set(new[3]) in delta['merges']['new']
>>> assert set(old[1]) in delta['unchanged']
>>> result = ub.urepr(delta, nl=2, sort=True, nobr=1, sk=True)
>>> print(result)
hybrid: {
    merges: [{{10}, {11}, {9}}, {{3}, {5, 6}}, {{4}}, {{7}, {8}}],
    new: {{3, 5, 6}, {4}, {7, 8}, {9, 10, 11}},
    old: {{10}, {3, 4}, {5, 6, 11}, {7}, {8, 9}},
    splits: [{{10}}, {{11}, {5, 6}}, {{3}, {4}}, {{7}}, {{8}, {9}}],
},
items: {
    added: {},
    removed: {},
},
merges: {
    new: [{12, 13, 14}, {31, 32, 33, 34, 35}],
    old: [{{12}, {13, 14}}, {{31, 32}, {33, 34, 35}}],
},
splits: {
    new: [{{20, 21}, {22, 23}}, {{41, 42, 43, 44}, {45}}],
    old: [{20, 21, 22, 23}, {41, 42, 43, 44, 45}],
},
unchanged: {
    {1, 2},
},
```

Example

```
>>> old = [
>>>     [1, 2, 3], [4], [5, 6, 7, 8, 9], [10, 11, 12]
>>> ]
>>> new = [
>>>     [1], [2], [3, 4], [5, 6, 7], [8, 9, 10, 11, 12]
>>> ]
>>> # every case here is hybrid
>>> pure_delta = grouping_delta(old, new, pure=True)
>>> assert len(list(ub.flatten(pure_delta['merges'].values()))) == 0
>>> assert len(list(ub.flatten(pure_delta['splits'].values()))) == 0
>>> delta = grouping_delta(old, new, pure=False)
>>> delta = order_dict_by(delta, ['unchanged', 'splits', 'merges'])
>>> result = ub.urepr(delta, nl=2, sort=True, sk=True)
>>> print(result)
{
    items: {
        added: {},
        removed: {},
    },
    merges: [
        [{3}, {4}],
        [{10, 11, 12}, {8, 9}],
    ],
    splits: [
        [{1}, {2}, {3}],
        [{5, 6, 7}, {8, 9}],
    ],
    unchanged: {},
}
```

Example

```
>>> delta = grouping_delta([[1, 2, 3]], [])
>>> assert len(delta['items']['removed']) == 3
>>> delta = grouping_delta([], [[1, 2, 3]])
>>> assert len(delta['items']['added']) == 3
>>> delta = grouping_delta([[1]], [[1, 2, 3]])
>>> assert len(delta['items']['added']) == 2
>>> assert len(delta['unchanged']) == 1
```

graphid.util.util_group.**order_dict_by**(*dict_*, *key_order*)

Reorders items in a dictionary according to a custom key order

Parameters

- **dict_** (*dict_*) – a dictionary
- **key_order** (*list*) – custom key order

Returns

sorted_dict

Return type
OrderedDict

Example

```
>>> dict_ = {1: 1, 2: 2, 3: 3, 4: 4}
>>> key_order = [4, 2, 3, 1]
>>> sorted_dict = order_dict_by(dict_, key_order)
>>> result = ('sorted_dict = %s' % (ub.urepr(sorted_dict, nl=False),))
>>> print(result)
>>> assert result == 'sorted_dict = {4: 4, 2: 2, 3: 3, 1: 1}'
```

graphid.util.util_group.**group_pairs**(*pair_list*)

Groups a list of items using the first element in each pair as the item and the second element as the groupid.

Parameters
pair_list (*list*) – list of 2-tuples (item, groupid)

Returns
groupid_to_items: maps a groupid to a list of items

Return type
dict

graphid.util.util_group.**sort_dict**(*dict_*, *part='keys'*, *key=None*, *reverse=False*)

sorts a dictionary by its values or its keys

Parameters

- **dict_** (*dict_*) – a dictionary
- **part** (*str*) – specifies to sort by keys or values
- **key** (*Optional[func]*) – a function that takes specified part and returns a sortable value
- **reverse** (*bool*) – (Defaults to False) - True for descinding order. False for ascending order.

Returns
sorted dictionary

Return type
OrderedDict

Example

```
>>> dict_ = {'a': 3, 'c': 2, 'b': 1}
>>> results = []
>>> results.append(sort_dict(dict_, 'keys'))
>>> results.append(sort_dict(dict_, 'vals'))
>>> results.append(sort_dict(dict_, 'vals', lambda x: -x))
>>> result = ub.urepr(results)
>>> print(result)
[
    {'a': 3, 'b': 1, 'c': 2},
    {'b': 1, 'c': 2, 'a': 3},
    {'a': 3, 'c': 2, 'b': 1},
]
```

1.1.3.1.11 graphid.util.util_image module

graphid.util.util_image.**ensure_float01**(*img*, *dtype=<class 'numpy.float32'>*, *copy=True*)

Ensure that an image is encoded using a float properly

graphid.util.util_image.**get_num_channels**(*img*)

Returns the number of color channels

graphid.util.util_image.**convert_colorspace**(*img*, *dst_space*, *src_space='BGR'*, *copy=False*, *dst=None*)

Converts colorspace of img. Convinience function around cv2.cvtColor

Parameters

- **img** (*ndarray[uint8_t, ndim=2]*) – image data
- **colorspace** (*str*) – RGB, LAB, etc
- **dst_space** (*unicode*) – (default = u'BGR')

Returns

img - image data

Return type

ndarray[uint8_t, ndim=2]

Example

```
>>> convert_colorspace(np.array([[[0, 0, 1]]], dtype=np.float32), 'LAB', src_space=
↪'RGB')
>>> convert_colorspace(np.array([[[0, 1, 0]]], dtype=np.float32), 'LAB', src_space=
↪'RGB')
>>> convert_colorspace(np.array([[[1, 0, 0]]], dtype=np.float32), 'LAB', src_space=
↪'RGB')
>>> convert_colorspace(np.array([[[1, 1, 1]]], dtype=np.float32), 'LAB', src_space=
↪'RGB')
>>> convert_colorspace(np.array([[[0, 0, 1]]], dtype=np.float32), 'HSV', src_space=
↪'RGB')
```

graphid.util.util_image.**_lookup_colorspace_code**(*dst_space*, *src_space='BGR'*)

graphid.util.util_image.**imread**(*fpath*, ***kw*)

reads image data in BGR format

Example

```
>>> # xdoctest: +SKIP("use kwimage.imread")
>>> import ubelt as ub
>>> import tempfile
>>> from os.path import splitext  # NOQA
>>> fpath = ub.grabdata('https://i.imgur.com/oHGsmvF.png', fname='carl.png')
>>> #fpath = ub.grabdata('http://www.topcoder.com/contest/problem/UrbanMapper3D/JAX_
↪Tile_043_DTM.tif')
>>> ext = splitext(fpath)[1]
>>> img1 = imread(fpath)
```

(continues on next page)

(continued from previous page)

```
>>> # Check that write + read preserves data
>>> tmp = tempfile.NamedTemporaryFile(suffix=ext)
>>> imwrite(tmp.name, img1)
>>> img2 = imread(tmp.name)
>>> assert np.all(img2 == img1)
```

Example

```
>>> # xdoctest: +SKIP("use kwimage.imread")
>>> import tempfile
>>> import ubelt as ub
>>> #img1 = (np.arange(0, 12 * 12 * 3).reshape(12, 12, 3) % 255).astype(np.uint8)
>>> img1 = imread(ub.grabdata('http://i.imgur.com/iXNf4Me.png', fname='ada.png'))
>>> tmp_tif = tempfile.NamedTemporaryFile(suffix='.tif')
>>> tmp_png = tempfile.NamedTemporaryFile(suffix='.png')
>>> imwrite(tmp_tif.name, img1)
>>> imwrite(tmp_png.name, img1)
>>> tif_im = imread(tmp_tif.name)
>>> png_im = imread(tmp_png.name)
>>> assert np.all(tif_im == png_im)
```

Example

```
>>> # xdoctest: +SKIP("use kwimage.imread")
>>> from graphid.util.util_image import *
>>> import tempfile
>>> import ubelt as ub
>>> #img1 = (np.arange(0, 12 * 12 * 3).reshape(12, 12, 3) % 255).astype(np.uint8)
>>> tif_fpath = ub.grabdata('https://ghostscript.com/doc/tiff/test/images/rgb-3c-
↪16b.tiff')
>>> img1 = imread(tif_fpath)
>>> tmp_tif = tempfile.NamedTemporaryFile(suffix='.tif')
>>> tmp_png = tempfile.NamedTemporaryFile(suffix='.png')
>>> imwrite(tmp_tif.name, img1)
>>> imwrite(tmp_png.name, img1)
>>> tif_im = imread(tmp_tif.name)
>>> png_im = imread(tmp_png.name)
>>> assert np.all(tif_im == png_im)
```

graphid.util.util_image.**imwrite**(*fpath*, *image*, ***kw*)

writes image data in BGR format

1.1.3.1.12 graphid.util.util_kw module

class graphid.util.util_kw.**KWSpec**(*spec*)

Bases: object

Safer keyword arguments with keyword specifications.

1.1.3.1.13 graphid.util.util_misc module

graphid.util.util_misc.**randn**(*mean=0*, *std=1*, *shape=[]*, *a_max=None*, *a_min=None*, *rng=None*)

graphid.util.util_misc.**aslist**(*sequence*)

Ensures that the sequence object is a Python list. Handles, numpy arrays, and python sequences (e.g. tuples, and iterables).

Parameters
sequence (*sequence*) – a list-like object

Returns
list_ - *sequence* as a Python list

Return type
list

Example

```
>>> s1 = [1, 2, 3]
>>> s2 = (1, 2, 3)
>>> assert aslist(s1) is s1
>>> assert aslist(s2) is not s2
>>> aslist(np.array([[1, 2], [3, 4], [5, 6]]))
[[1, 2], [3, 4], [5, 6]]
>>> aslist(range(3))
[0, 1, 2]
```

class graphid.util.util_misc.**classproperty**(*fget=None*, *fset=None*, *fdel=None*, *doc=None*)

Bases: property

Decorates a method turning it into a classattribute

References

https://stackoverflow.com/questions/1697501/python-staticmethod-with-property

graphid.util.util_misc.**estarmap**(*func*, *iter_*, ***kwargs*)

Eager version of it.starmap from itertools

Note this is inefficient and should only be used when prototyping and debugging.

graphid.util.util_misc.**delete_dict_keys**(*dict_*, *key_list*)

Removes items from a dictionary inplace. Keys that do not exist are ignored.

Parameters

- **dict_** (*dict*) – dict like object with a __del__ attribute

- **key_list** (*list*) – list of keys that specify the items to remove

Example

```
>>> dict_ = {'bread': 1, 'churches': 1, 'cider': 2, 'very small rocks': 2}
>>> key_list = ['duck', 'bread', 'cider']
>>> delete_dict_keys(dict_, key_list)
>>> result = ub.urepr(dict_, nl=False)
>>> print(result)
{'churches': 1, 'very small rocks': 2}
```

graphid.util.util_misc.**flag_None_items**(*list_*)

graphid.util.util_misc.**where**(*flag_list*)

takes flags returns indexes of True values

graphid.util.util_misc.**delete_items_by_index**(*list_*, *index_list*, *copy=False*)

Remove items from `list_` at positions specified in `index_list` The original `list_` is preserved if `copy` is True

Parameters

- **list_** (*list*)
- **index_list** (*list*)
- **copy** (*bool*) – preserves original list if True

Example

```
>>> list_ = [8, 1, 8, 1, 6, 6, 3, 4, 4, 5, 6]
>>> index_list = [2, -1]
>>> result = delete_items_by_index(list_, index_list)
>>> print(result)
[8, 1, 1, 6, 6, 3, 4, 4, 5]
```

graphid.util.util_misc.**make_index_lookup**(*list_*, *dict_factory=<class 'dict'>*)

Parameters

list_ (*list*) – assumed to have unique items

Returns

mapping from item to index

Return type

dict

Example

```
>>> list_ = [5, 3, 8, 2]
>>> idx2_item = make_index_lookup(list_)
>>> result = ub.urepr(idx2_item, nl=False, sort=1)
>>> assert list(ub.take(idx2_item, list_)) == list(range(len(list_)))
>>> print(result)
{2: 3, 3: 1, 5: 0, 8: 2}
```

graphid.util.util_misc.**cprint**(*text*, *color=None*)

provides some color to terminal output

Parameters

- **text** (*str*)
- **color** (*str*)

Example0:

```
>>> import pygments.console
>>> msg_list = list(pygments.console.codes.keys())
>>> color_list = list(pygments.console.codes.keys())
>>> [cprint(text, color) for text, color in zip(msg_list, color_list)]
```

Example1:

```
>>> import pygments.console
>>> print('line1')
>>> cprint('line2', 'red')
>>> cprint('line3', 'blue')
>>> cprint('line4', 'magenta')
>>> cprint('line5', 'reset')
>>> cprint('line5', 'magenta')
>>> print('line6')
```

graphid.util.util_misc.**ensure_iterable**(*obj*)

Parameters
obj (*scalar or iterable*)

Returns
obj if it was iterable otherwise [obj]

Return type
it3erable

Timeit:
%timeit util.ensure_iterable([1]) %timeit util.ensure_iterable(1) %timeit util.ensure_iterable(np.array(1)) %timeit util.ensure_iterable([1]) %timeit [1]

Example

```
>>> obj_list = [3, [3], '3', (3,), [3,4,5]]
>>> result = [ensure_iterable(obj) for obj in obj_list]
>>> result = str(result)
>>> print(result)
[[3], [3], ['3'], (3,), [3, 4, 5]]
```

graphid.util.util_misc.**highlight_regex**(*str_*, *pat*, *reflags=0*, *color='red'*)

FIXME Use pygments instead

graphid.util.util_misc.**regex_word**(*w*)

graphid.util.util_misc.**setdiff**(*list1*, *list2*)

returns list1 elements that are not in list2. preserves order of list1

Parameters

- **list1** (*list*)
- **list2** (*list*)

Returns

new_list

Return type

list

Example

```
>>> list1 = ['featweight_rowid', 'feature_rowid', 'config_rowid', 'featweight_
↪forground_weight']
>>> list2 = [u'featweight_rowid']
>>> new_list = setdiff(list1, list2)
>>> result = ub.urepr(new_list, nl=False)
>>> print(result)
['feature_rowid', 'config_rowid', 'featweight_forground_weight']
```

graphid.util.util_misc.**all_dict_combinations**(*varied_dict*)

Parameters

varied_dict (*dict*) – a dict with lists of possible parameter settings

Returns

dict_list a list of dicts correpsonding to all combinations of params settings

Return type

list

Example

```
>>> varied_dict = {'logdist_weight': [0.0, 1.0], 'pipeline_root': ['vsmany'], 'sv_on
↪': [True, False, None]}
>>> dict_list = all_dict_combinations(varied_dict)
>>> result = str(ub.urepr(dict_list))
>>> print(result)
[
    {'logdist_weight': 0.0, 'pipeline_root': 'vsmany', 'sv_on': True},
    {'logdist_weight': 0.0, 'pipeline_root': 'vsmany', 'sv_on': False},
    {'logdist_weight': 0.0, 'pipeline_root': 'vsmany', 'sv_on': None},
    {'logdist_weight': 1.0, 'pipeline_root': 'vsmany', 'sv_on': True},
    {'logdist_weight': 1.0, 'pipeline_root': 'vsmany', 'sv_on': False},
    {'logdist_weight': 1.0, 'pipeline_root': 'vsmany', 'sv_on': None},
]
```

graphid.util.util_misc.**iteritems_sorted**(*dict_*)

change to iteritems ordered

graphid.util.util_misc.**partial_order**(*list_*, *part*)

graphid.util.util_misc.**replace_nones**(*list_*, *repl=-1*)

Recursively removes Nones in all lists and sublists and replaces them with the repl variable

Parameters

- **list_** (*list*)
- **repl** (*obj*) – replacement value

Returns

list

Example

```
>>> list_ = [None, 0, 1, 2]
>>> repl = -1
>>> repl_list = replace_nones(list_, repl)
>>> result = str(repl_list)
>>> print(result)
[-1, 0, 1, 2]
```

graphid.util.util_misc.**take_percentile_parts**(*arr*, *front=None*, *mid=None*, *back=None*)

Take parts from front, back, or middle of a list

Example

```
>>> arr = list(range(20))
>>> front = 3
>>> mid = 3
>>> back = 3
>>> result = take_percentile_parts(arr, front, mid, back)
>>> print(result)
[0, 1, 2, 9, 10, 11, 17, 18, 19]
```

graphid.util.util_misc.**snapped_slice**(*size*, *frac*, *n*)

Creates a slice spanning *n* items in a list of length *size* at position *frac*.

Parameters

- **size** (*int*) – length of the list
- **frac** (*float*) – position in the range [0, 1]
- **n** (*int*) – number of items in the slice

Returns

slice object that best fits the criteria

Return type

slice

SeeAlso:

take_percentile_parts

Example:

Example

```
>>> # DISABLE_DOCTEST
>>> print(snapped_slice(0, 0, 10))
>>> print(snapped_slice(1, 0, 10))
>>> print(snapped_slice(100, 0, 10))
>>> print(snapped_slice(9, 0, 10))
>>> print(snapped_slice(100, 1, 10))
pass
```

graphid.util.util_misc.**get_timestamp**(*format_='iso'*, *use_second=False*, *delta_seconds=None*, *isutc=False*, *timezone=False*)

Parameters

- **format_** (*str*) – (tag, printable, filename, other)
- **use_second** (*bool*)
- **delta_seconds** (*None*)

Returns

stamp

Return type

str

Example

```
>>> format_ = 'printable'
>>> use_second = False
>>> delta_seconds = None
>>> stamp = get_timestamp(format_, use_second, delta_seconds)
>>> print(stamp)
>>> assert len(stamp) == len('15:43:04 2015/02/24')
```

graphid.util.util_misc.**isect**(*list1*, *list2*)

returns list1 elements that are also in list2. preserves order of list1

intersect_ordered

Parameters

- **list1** (*list*)
- **list2** (*list*)

Returns

new_list

Return type

list

Example

```
>>> list1 = ['featweight_rowid', 'feature_rowid', 'config_rowid', 'featweight_
↪forground_weight']
>>> list2 = [u'featweight_rowid']
>>> result = isect(list1, list2)
>>> print(result)
['featweight_rowid']
```

graphid.util.util_misc.**safe_extreme**(*arr*, *op*, *fill=nan*, *finite=False*, *nans=True*)

Applies an exterme operation to an 1d array (typically max/min) but ensures a value is always returned even in operations without identities. The default identity must be specified using the *fill* argument.

Parameters

- **arr** (*ndarray*) – 1d array to take extreme of
- **op** (*func*) – vectorized operation like np.max to apply to array
- **fill** (*float*) – return type if arr has no elements (default = nan)
- **finite** (*bool*) – if True ignores non-finite values (default = False)
- **nans** (*bool*) – if False ignores nans (default = True)

graphid.util.util_misc.**safe_argmax**(*arr*, *fill=nan*, *finite=False*, *nans=True*)

Doctest

```
>>> assert safe_argmax([np.nan, np.nan], nans=False) == 0
>>> assert safe_argmax([-100, np.nan], nans=False) == 0
>>> assert safe_argmax([np.nan, -100], nans=False) == 1
>>> assert safe_argmax([-100, 0], nans=False) == 1
>>> assert np.isnan(safe_argmax([]))
```

graphid.util.util_misc.**safe_max**(*arr*, *fill=nan*, *finite=False*, *nans=True*)

Parameters

- **arr** (*ndarray*) – 1d array to take max of
- **fill** (*float*) – return type if arr has no elements (default = nan)
- **finite** (*bool*) – if True ignores non-finite values (default = False)
- **nans** (*bool*) – if False ignores nans (default = True)

Example

```
>>> arrs = [[], [np.nan], [-np.inf, np.nan, np.inf], [np.inf], [np.inf, 1], [0, 1]]
>>> arrs = [np.array(arr) for arr in arrs]
>>> fill = np.nan
>>> results1 = [safe_max(arr, fill, finite=False, nans=True) for arr in arrs]
>>> results2 = [safe_max(arr, fill, finite=True, nans=True) for arr in arrs]
>>> results3 = [safe_max(arr, fill, finite=True, nans=False) for arr in arrs]
>>> results4 = [safe_max(arr, fill, finite=False, nans=False) for arr in arrs]
>>> results = [results1, results2, results3, results4]
>>> result = ('results = %s' % (ub.urepr(results, nl=1, sv=1),))
>>> print(result)
results = [
    [nan, nan, nan, inf, inf, 1],
    [nan, nan, nan, nan, 1.0, 1],
    [nan, nan, nan, nan, 1.0, 1],
    [nan, nan, inf, inf, inf, 1],
]
```

graphid.util.util_misc.**safe_min**(*arr*, *fill=nan*, *finite=False*, *nans=True*)

Example

```
>>> arrs = [[], [np.nan], [-np.inf, np.nan, np.inf], [np.inf], [np.inf, 1], [0, 1]]
>>> arrs = [np.array(arr) for arr in arrs]
>>> fill = np.nan
>>> results1 = [safe_min(arr, fill, finite=False, nans=True) for arr in arrs]
>>> results2 = [safe_min(arr, fill, finite=True, nans=True) for arr in arrs]
>>> results3 = [safe_min(arr, fill, finite=True, nans=False) for arr in arrs]
>>> results4 = [safe_min(arr, fill, finite=False, nans=False) for arr in arrs]
>>> results = [results1, results2, results3, results4]
>>> result = ('results = %s' % (ub.urepr(results, nl=1, sv=1),))
>>> print(result)
```

(continues on next page)

(continued from previous page)

```
results = [
    [nan, nan, nan, inf, 1.0, 0],
    [nan, nan, nan, nan, 1.0, 0],
    [nan, nan, nan, nan, 1.0, 0],
    [nan, nan, -inf, inf, 1.0, 0],
]
```

graphid.util.util_misc.**stats_dict**(*list_*, *axis=None*, *use_nan=False*, *use_sum=False*, *use_median=False*, *size=False*)

Parameters

- **list_** (*listlike*) – values to get statistics of
- **axis** (*int*) – if *list_* is ndarray then this specifies the axis

Returns

stats: dictionary of common numpy statistics
(min, max, mean, std, nMin, nMax, shape)

Return type
OrderedDict

Examples0:

```
>>> # xdoctest: +IGNORE_WHITESPACE
>>> import numpy as np
>>> axis = 0
>>> np.random.seed(0)
>>> list_ = np.random.rand(10, 2).astype(np.float32)
>>> stats = stats_dict(list_, axis, use_nan=False)
>>> result = str(ub.urepr(stats, nl=1, precision=4, with_dtype=True))
>>> print(result)
{
    'mean': np.array([0.5206, 0.6425], dtype=np.float32),
    'std': np.array([0.2854, 0.2517], dtype=np.float32),
    'max': np.array([0.9637, 0.9256], dtype=np.float32),
    'min': np.array([0.0202, 0.0871], dtype=np.float32),
    'nMin': np.array([1, 1], dtype=np.int32),
    'nMax': np.array([1, 1], dtype=np.int32),
    'shape': (10, 2),
}
```

Examples1:

```
>>> import numpy as np
>>> axis = 0
>>> rng = np.random.RandomState(0)
>>> list_ = rng.randint(0, 42, size=100).astype(np.float32)
>>> list_[4] = np.nan
>>> stats = stats_dict(list_, axis, use_nan=True)
>>> result = str(ub.urepr(stats, precision=1, sk=True))
>>> print(result)
```

(continues on next page)

(continued from previous page)

```
{mean: 20.0, std: 13.2, max: 41.0, min: 0.0, nMin: 7, nMax: 3, shape: (100,),␣
↪num_nan: 1,}
```

1.1.3.1.14 graphid.util.util_numpy module

graphid.util.util_numpy.**iter_reduce_ufunc**(*ufunc*, *arr_iter*, *out=None*)

constant memory iteration and reduction

applys ufunc from left to right over the input arrays

Example

```
>>> arr_list = [
...     np.array([0, 1, 2, 3, 8, 9]),
...     np.array([4, 1, 2, 3, 4, 5]),
...     np.array([0, 5, 2, 3, 4, 5]),
...     np.array([1, 1, 6, 3, 4, 5]),
...     np.array([0, 1, 2, 7, 4, 5])
... ]
>>> memory = np.array([9, 9, 9, 9, 9, 9])
>>> gen_memory = memory.copy()
>>> def arr_gen(arr_list, gen_memory):
...     for arr in arr_list:
...         gen_memory[:] = arr
...         yield gen_memory
>>> print('memory = %r' % (memory,))
>>> print('gen_memory = %r' % (gen_memory,))
>>> ufunc = np.maximum
>>> res1 = iter_reduce_ufunc(ufunc, iter(arr_list), out=None)
>>> res2 = iter_reduce_ufunc(ufunc, iter(arr_list), out=memory)
>>> res3 = iter_reduce_ufunc(ufunc, arr_gen(arr_list, gen_memory), out=memory)
>>> print('res1       = %r' % (res1,))
>>> print('res2       = %r' % (res2,))
>>> print('res3       = %r' % (res3,))
>>> print('memory     = %r' % (memory,))
>>> print('gen_memory = %r' % (gen_memory,))
>>> assert np.all(res1 == res2)
>>> assert np.all(res2 == res3)
```

graphid.util.util_numpy.**isect_flags**(*arr*, *other*)

Example

```
>>> arr = np.array([
>>>     [1, 2, 3, 4],
>>>     [5, 6, 3, 4],
>>>     [1, 1, 3, 4],
>>> ])
>>> other = np.array([1, 4, 6])
>>> mask = isect_flags(arr, other)
>>> print(mask)
[[ True False False  True]
 [False  True False  True]
 [ True  True False  True]]
```

graphid.util.util_numpy.**atleast_nd**(*arr*, *n*, *front=False*)

View inputs as arrays with at least n dimensions. TODO: Submit as a PR to numpy

Parameters

- **arr** (*array_like*) – One array-like object. Non-array inputs are converted to arrays. Arrays that already have n or more dimensions are preserved.
- **n** (*int*) – number of dimensions to ensure
- **tofront** (*bool*) – if True new dimensions are added to the front of the array. otherwise they are added to the back.

Returns

An array with `a.ndim >= n`. Copies are avoided where possible, and views with three or more dimensions are returned. For example, a 1-D array of shape `(N,)` becomes a view of shape `(1, N, 1)`, and a 2-D array of shape `(M, N)` becomes a view of shape `(M, N, 1)`.

Return type

ndarray

Example

```
>>> n = 2
>>> arr = np.array([1, 1, 1])
>>> arr_ = atleast_nd(arr, n)
>>> result = ub.urepr(arr_.tolist(), nl=0)
>>> print(result)
[[1], [1], [1]]
```

Example

```
>>> n = 4
>>> arr1 = [1, 1, 1]
>>> arr2 = np.array(0)
>>> arr3 = np.array([[[[[1]]]]])
>>> arr1_ = atleast_nd(arr1, n)
>>> arr2_ = atleast_nd(arr2, n)
>>> arr3_ = atleast_nd(arr3, n)
```

(continues on next page)

(continued from previous page)

```
>>> result1 = ub.urepr(arr1_.tolist(), nl=0)
>>> result2 = ub.urepr(arr2_.tolist(), nl=0)
>>> result3 = ub.urepr(arr3_.tolist(), nl=0)
>>> result = '\n'.join([result1, result2, result3])
>>> print(result)
[[[[1]]], [[[1]]], [[[1]]]]
[[[[0]]]]
[[[[[1]]]]]
```

Benchmark

import ubelt N = 100

t1 = ubelt.Timerit(N, label='mine') for timer in t1:

arr = np.empty((10, 10)) with timer:

atleast_nd(arr, 3)

t2 = ubelt.Timerit(N, label='baseline') for timer in t2:

arr = np.empty((10, 10)) with timer:

np.atleast_3d(arr)

graphid.util.util_numpy.**apply_grouping**(*items*, *groupxs*, *axis=0*)

applies grouping from group_indicies apply_grouping

Parameters

- **items** (*ndarray*)
- **groupxs** (*list of ndarrays*)

Returns

grouped items

Return type

list of ndarrays

SeeAlso:

group_indices invert_apply_grouping

Example

```
>>> # xdoctest: +IGNORE_WHITESPACE
>>> idx2_groupid = np.array([2, 1, 2, 1, 2, 1, 2, 3, 3, 3, 3])
>>> items        = np.array([1, 8, 5, 5, 8, 6, 7, 5, 3, 0, 9])
>>> (keys, groupxs) = group_indices(idx2_groupid)
>>> grouped_items = apply_grouping(items, groupxs)
>>> result = str(grouped_items)
>>> print(result)
[array([8, 5, 6]), array([1, 5, 8, 7]), array([5, 3, 0, 9])]
```

graphid.util.util_numpy.**group_indices**(*idx2_groupid*, *assume_sorted=False*)

Parameters
idx2_groupid (*ndarray*) – numpy array of group ids (must be numeric)

Returns
(keys, groupxs)

Return type
tuple (ndarray, list of ndarrays)

Example0:

```
>>> # xdoctest: +IGNORE_WHITESPACE
>>> idx2_groupid = np.array([2, 1, 2, 1, 2, 1, 2, 3, 3, 3, 3])
>>> (keys, groupxs) = group_indices(idx2_groupid)
>>> result = ub.urepr((keys, groupxs), nobr=True, with_dtype=True)
>>> print(result)
```

np.array([1, 2, 3], dtype=np.int64), [

np.array([1, 3, 5], dtype=np.int64), np.array([0, 2, 4, 6], dtype=np.int64), np.array([7, 8, 9, 10], dtype=np.int64)...

Example1:

```
>>> # xdoctest: +IGNORE_WHITESPACE
>>> idx2_groupid = np.array([[  24], [ 129], [ 659], [ 659], [ 24],
...       [659], [ 659], [ 822], [ 659], [ 659], [24]])
>>> # 2d arrays must be flattened before coming into this function so
>>> # information is on the last axis
>>> (keys, groupxs) = group_indices(idx2_groupid.T[0])
>>> result = ub.urepr((keys, groupxs), nobr=True, with_dtype=True)
>>> print(result)
```

np.array([24, 129, 659, 822], dtype=np.int64), [

np.array([0, 4, 10], dtype=np.int64), np.array([1], dtype=np.int64), np.array([2, 3, 5, 6, 8, 9], dtype=np.int64), np.array([7], dtype=np.int64)...

Example2:

```
>>> # xdoctest: +IGNORE_WHITESPACE
>>> idx2_groupid = np.array([True, True, False, True, False, False, True])
>>> (keys, groupxs) = group_indices(idx2_groupid)
>>> result = ub.urepr((keys, groupxs), nobr=True, with_dtype=True)
>>> print(result)
```

np.array([False, True], dtype=bool), [

np.array([2, 4, 5], dtype=np.int64), np.array([0, 1, 3, 6], dtype=np.int64)...

Timeit:
import numba group_indices_numba = numba.jit(group_indices) group_indices_numba(idx2_groupid)

SeeAlso:
apply_grouping

References

http://stackoverflow.com/questions/4651683/ numpy-grouping-using-itertools-groupby-performance

---

Todo: Look into np.split http://stackoverflow.com/questions/21888406/ getting-the-indexes-to-the-duplicate-columns-of-a-numpy-array

---

graphid.util.util_numpy.**group_items**(*item_list*, *groupid_list*, *assume_sorted=False*, *axis=None*)

1.1.3.1.15 graphid.util.util_random module

graphid.util.util_random.**shuffle**(*items*, *rng=None*)

Shuffles a list inplace and then returns it for convinience

Parameters

- **items** (*list or ndarray*) – list to shuffl
- **rng** (*RandomState or int*) – seed or random number gen

Returns

this is the input, but returned for convinience

Return type

list

Example

```
>>> list1 = [1, 2, 3, 4, 5, 6]
>>> list2 = shuffle(list(list1), rng=1)
>>> assert list1 != list2
>>> result = str(list2)
>>> print(result)
[3, 2, 5, 1, 4, 6]
```

graphid.util.util_random.**random_combinations**(*items*, *size*, *num=None*, *rng=None*)

Yields *num* combinations of length *size* from items in random order

Parameters

- **items** (*List*) – pool of items to choose from
- **size** (*int*) – number of items in each combination
- **num** (*None, default=None*) – number of combinations to generate
- **rng** (*int | RandomState, default=None*) – seed or random number generator

Yields

tuple – combo

Example

```
>>> import ubelt as ub  # NOQA
>>> items = list(range(10))
>>> size = 3
>>> num = 5
>>> rng = 0
>>> combos = list(random_combinations(items, size, num, rng))
>>> result = ('combos = %s' % (ub.urepr(combos),))
>>> print(result)
```

Example

```
>>> import ubelt as ub  # NOQA
>>> items = list(zip(range(10), range(10)))
>>> size = 3
>>> num = 5
>>> rng = 0
>>> combos = list(random_combinations(items, size, num, rng))
>>> result = ('combos = %s' % (ub.urepr(combos),))
>>> print(result)
```

graphid.util.util_random.**random_product**(*items*, *num=None*, *rng=None*)

Yields *num* items from the cartesian product of items in a random order.

Parameters

items (*list of sequences*) – items to get caresian product of packed in a list or tuple. (note this deviates from api of it.product)

Example

```
>>> items = [(1, 2, 3), (4, 5, 6, 7)]
>>> rng = 0
>>> list(random_product(items, rng=0))
>>> list(random_product(items, num=3, rng=0))
```

graphid.util.util_random.**_npstate_to_pystate**(*npstate*)

Convert state of a NumPy RandomState object to a state that can be used by Python's Random.

References

https://stackoverflow.com/questions/44313620/converting-randomstate

Example

```
>>> py_rng = random.Random(0)
>>> np_rng = np.random.RandomState(seed=0)
>>> npstate = np_rng.get_state()
>>> pystate = _npstate_to_pystate(npstate)
>>> py_rng.setstate(pystate)
>>> assert np_rng.rand() == py_rng.random()
```

graphid.util.util_random.**_pystate_to_npstate**(*pystate*)

Convert state of a Python Random object to state usable by NumPy RandomState.

References

https://stackoverflow.com/questions/44313620/converting-randomstate

Example

```
>>> py_rng = random.Random(0)
>>> np_rng = np.random.RandomState(seed=0)
>>> pystate = py_rng.getstate()
>>> npstate = _pystate_to_npstate(pystate)
>>> np_rng.set_state(npstate)
>>> assert np_rng.rand() == py_rng.random()
```

graphid.util.util_random.**ensure_rng**(*rng*, *api='numpy'*)

Returns a random number generator

Parameters

seed – if None, then the rng is unseeded. Otherwise the seed can be an integer or a RandomState class

Example

```
>>> rng = ensure_rng(None)
>>> ensure_rng(0).randint(0, 1000)
684
>>> ensure_rng(np.random.RandomState(1)).randint(0, 1000)
37
```

Example

```
>>> num = 4
>>> print('--- Python as PYTHON ---')
>>> py_rng = random.Random(0)
>>> pp_nums = [py_rng.random() for _ in range(num)]
>>> print(pp_nums)
>>> print('--- Numpy as PYTHON ---')
>>> np_rng = ensure_rng(random.Random(0), api='numpy')
```

(continues on next page)

(continued from previous page)

```
>>> np_nums = [np_rng.rand() for _ in range(num)]
>>> print(np_nums)
>>> print('--- Numpy as NUMPY---')
>>> np_rng = np.random.RandomState(seed=0)
>>> nn_nums = [np_rng.rand() for _ in range(num)]
>>> print(nn_nums)
>>> print('--- Python as NUMPY---')
>>> py_rng = ensure_rng(np.random.RandomState(seed=0), api='python')
>>> pn_nums = [py_rng.random() for _ in range(num)]
>>> print(pn_nums)
>>> assert np_nums == pp_nums
>>> assert pn_nums == nn_nums
```

1.1.3.1.16 graphid.util.util_tags module

graphid.util.util_tags.**tag_hist**(*tags_list*)

graphid.util.util_tags.**build_alias_map**(*regex_map*, *tag_vocab*)

Constructs explicit mapping. Order of items in regex map matters. Items at top are given preference.

graphid.util.util_tags.**alias_tags**(*tags_list*, *alias_map*)

update tags to new values

Parameters

- **tags_list** (*list*)
- **alias_map** (*list*) – list of 2-tuples with regex, value

Returns

updated tags

Return type

list

graphid.util.util_tags.**filterflags_general_tags**(*tags_list*, *has_any=None*, *has_all=None*, *has_none=None*, *min_num=None*, *max_num=None*, *any_startswith=None*, *any_endswith=None*, *in_any=None*, *any_match=None*, *none_match=None*, *logic='and'*, *ignore_case=True*)

Parameters

- **tags_list** (*list*)
- **has_any** (*None*) – (default = None)
- **has_all** (*None*) – (default = None)
- **min_num** (*None*) – (default = None)
- **max_num** (*None*) – (default = None)

Notes

in_any should probably be ni_any

Example1:

```
>>> # ENABLE_DOCTEST
>>> tags_list = [['v'], [], ['P'], ['P'], ['n', 'o'], [], ['n', 'N'], ['e', 'i
↪', 'p', 'b', 'n'], ['n'], ['n'], ['N']]
>>> has_all = 'n'
>>> min_num = 1
>>> flags = filterflags_general_tags(tags_list, has_all=has_all, min_num=min_
↪num)
>>> result = list(ub.compress(tags_list, flags))
>>> print('result = %r' % (result,))
```

Example2:

```
>>> tags_list = [['vn'], ['vn', 'no'], ['P'], ['P'], ['n', 'o'], [], ['n', 'N
↪'], ['e', 'i', 'p', 'b', 'n'], ['n'], ['n', 'nP'], ['NP']]
>>> kwargs = {
>>>     'any_endswith': 'n',
>>>     'any_match': None,
>>>     'any_startswith': 'n',
>>>     'has_all': None,
>>>     'has_any': None,
>>>     'has_none': None,
>>>     'max_num': 3,
>>>     'min_num': 1,
>>>     'none_match': ['P'],
>>> }
>>> flags = filterflags_general_tags(tags_list, **kwargs)
>>> filtered = list(ub.compress(tags_list, flags))
>>> result = ('result = %s' % (ub.urepr(filtered, nl=0),))
>>> print(result)
result = [['vn', 'no'], ['n', 'o'], ['n', 'N'], ['n'], ['n', 'nP']]
```

1.1.3.2 Module contents

mkinit graphid.util

class graphid.util.**Color**(*color*, *alpha=None*, *space=None*)

Bases: NiceRepr

move to colorutil?

Example

```
>>> print(Color('g'))
>>> print(Color('orangered'))
>>> print(Color('#AAAAAA').as255())
>>> print(Color([0, 255, 0]))
>>> print(Color([1, 1, 1.]))
>>> print(Color([1, 1, 1]))
>>> print(Color(Color([1, 1, 1])).as255())
>>> print(Color(Color([1., 0, 1, 0])).ashex())
>>> print(Color([1, 1, 1], alpha=255))
>>> print(Color([1, 1, 1], alpha=255, space='lab'))
```

ashex(*space=None*)

as255(*space=None*)

as01(*space=None*)

self = mplutil.Color('red') mplutil.Color('green').as01('rgba')

classmethod _is_base01()

check if a color is in base 01

classmethod _is_base255(*channels*)

there is a one corner case where all pixels are 1 or less

classmethod _hex_to_01(*hex_color*)

hex_color = '#6A5AFFAF'

_ensure_color01(*color*)

Infer what type color is and normalize to 01

classmethod _255_to_01(*color255*)

converts base 255 color to base 01 color

classmethod _string_to_01(*'green'*)

classmethod _string_to_01(*'red'*) → None

classmethod named_colors()

classmethod distinct(*num*, *space='bgr'*)

Make multiple distinct colors

adjust_hsv(*hue_adjust=0.0*, *sat_adjust=0.0*, *val_adjust=0.0*)

Performs adaptive changes to the HSV values of the color.

Parameters

- **hue_adjust** (*float*) – addative
- **sat_adjust** (*float*)
- **val_adjust** (*float*)

Returns

new_rgb

Return type

list

CommandLine

```
python -m graphid.util.mplutil Color.adjust_hsv
```

Example

```
>>> rgb_list = [Color(c).as01() for c in ['pink', 'yellow', 'green']]
>>> hue_adjust = -0.1
>>> sat_adjust = +0.5
>>> val_adjust = -0.1
>>> # execute function
>>> new_rgb_list = [Color(rgb).adjust_hsv(hue_adjust, sat_adjust, val_adjust)␣
↪for rgb in rgb_list]
>>> print(ub.urepr(new_rgb_list, nl=1, sv=True))
[
    <Color(rgb: 0.90, 0.23, 0.75)>,
    <Color(rgb: 0.90, 0.36, 0.00)>,
    <Color(rgb: 0.24, 0.40, 0.00)>,
]
>>> # xdoc: +REQUIRES(--show)
>>> color_list = rgb_list + new_rgb_list
>>> testshow_colors(color_list)
```

convert(*space*)

Converts to a new colorspace

class graphid.util.**PanEvents**(*ax=None*)

Bases: object

pan_on_press(*event*)

pan_on_release(*event*)

pan_on_motion(*event*)

class graphid.util.**PlotNums**(*nRows=None*, *nCols=None*, *nSubplots=None*, *start=0*)

Bases: object

Convinience class for dealing with plot numberings (pnums)

Example

```
>>> pnum_ = PlotNums(nRows=2, nCols=2)
>>> # Indexable
>>> print(pnum_[0])
(2, 2, 1)
>>> # Iterable
>>> print(ub.urepr(list(pnum_), nl=0, nobr=1))
(2, 2, 1), (2, 2, 2), (2, 2, 3), (2, 2, 4)
>>> # Callable (iterates through a default iterator)
>>> print(pnum_())
(2, 2, 1)
```

(continues on next page)

(continued from previous page)

```
>>> print(pnum_())
(2, 2, 2)
```

classmethod _get_num_rc(*nSubplots=None*, *nRows=None*, *nCols=None*)

Gets a constrained row column plot grid

Parameters

- **nSubplots** (*None*) – (default = None)
- **nRows** (*None*) – (default = None)
- **nCols** (*None*) – (default = None)

Returns

(nRows, nCols)

Return type

tuple

Example

```
>>> cases = [
>>>     dict(nRows=None, nCols=None, nSubplots=None),
>>>     dict(nRows=2, nCols=None, nSubplots=5),
>>>     dict(nRows=None, nCols=2, nSubplots=5),
>>>     dict(nRows=None, nCols=None, nSubplots=5),
>>> ]
>>> for kw in cases:
>>>     print('----')
>>>     size = PlotNums._get_num_rc(**kw)
>>>     if kw['nSubplots'] is not None:
>>>         assert size[0] * size[1] >= kw['nSubplots']
>>>     print('**kw = %s' % (ub.urepr(kw),))
>>>     print('size = %r' % (size,))
```

_get_square_row_cols(*max_cols=None*, *fix=False*, *inclusive=True*)

Parameters

- **nSubplots** (*int*)
- **max_cols** (*int*)

Returns

(int, int)

Return type

tuple

Example

```
>>> nSubplots = 9
>>> nSubplots_list = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]
>>> max_cols = None
>>> rc_list = [PlotNums._get_square_row_cols(nSubplots, fix=True) for␣
↪nSubplots in nSubplots_list]
>>> print(repr(np.array(rc_list).T))
array([[1, 1, 2, 2, 2, 2, 3, 3, 3, 3, 3],
       [1, 2, 2, 2, 3, 3, 3, 3, 3, 4, 4]])
```

graphid.util.**adjust_subplots**(*left=None*, *right=None*, *bottom=None*, *top=None*, *wspace=None*, *hspace=None*, *fig=None*)

Kwargs:
left (float): left side of the subplots of the figure right (float): right side of the subplots of the figure bottom (float): bottom of the subplots of the figure top (float): top of the subplots of the figure wspace (float): width reserved for blank space between subplots hspace (float): height reserved for blank space between subplots

graphid.util.**axes_extent**(*axs*, *pad=0.0*)

Get the full extent of a group of axes, including axes labels, tick labels, and titles.

graphid.util.**colorbar**(*scalars*, *colors*, *custom=False*, *lbl=None*, *ticklabels=None*, *float_format='%.2f'*, ***kwargs*)

adds a color bar next to the axes based on specific scalars

Parameters

- **scalars** (*ndarray*)
- **colors** (*ndarray*)
- **custom** (*bool*) – use custom ticks

Kwargs:
See plt.colorbar

Returns
matplotlib colorbar object

Return type
cb

graphid.util.**deterministic_shuffle**(*list_*, *rng=0*)

Parameters

- **list_** (*list*)
- **seed** (*int*)

Returns
list_

Return type
list

Example

```
>>> list_ = [1, 2, 3, 4, 5, 6]
>>> seed = 1
>>> list_ = deterministic_shuffle(list_, seed)
>>> result = str(list_)
>>> print(result)
[3, 2, 5, 1, 4, 6]
```

graphid.util.**dict_intersection**(*dict1*, *dict2*)

Key AND Value based dictionary intersection

Parameters

- **dict1** (*dict*)
- **dict2** (*dict*)

Returns

mergedict_

Return type

dict

Example

```
>>> dict1 = {'a': 1, 'b': 2, 'c': 3, 'd': 4}
>>> dict2 = {'b': 2, 'c': 3, 'd': 5, 'e': 21, 'f': 42}
>>> mergedict_ = dict_intersection(dict1, dict2)
>>> print(ub.urepr(mergedict_, nl=0, sort=1))
{'b': 2, 'c': 3}
```

graphid.util.**distinct_colors**(*N*, *brightness=0.878*, *randomize=True*, *hue_range=(0.0, 1.0)*, *cmap_seed=None*)

Parameters

- **N** (*int*)
- **brightness** (*float*)

Returns

RGB_tuples

Return type

list

CommandLine

```
python -m color_funcs --test-distinct_colors --N 2 --show --hue-range=0.05,.95
python -m color_funcs --test-distinct_colors --N 3 --show --hue-range=0.05,.95
python -m color_funcs --test-distinct_colors --N 4 --show --hue-range=0.05,.95
python -m .color_funcs --test-distinct_colors --N 3 --show --no-randomize
python -m .color_funcs --test-distinct_colors --N 4 --show --no-randomize
python -m .color_funcs --test-distinct_colors --N 6 --show --no-randomize
python -m .color_funcs --test-distinct_colors --N 20 --show
```

References

http://blog.jianhuashao.com/2011/09/generate-n-distinct-colors.html

graphid.util.**distinct_markers**(*num*, *style='astrisk'*, *total=None*, *offset=0*)

Parameters
num (*?*)

graphid.util.**draw_border**(*ax*, *color*, *lw=2*, *offset=None*, *adjust=True*)

draws rectangle border around a subplot

graphid.util.**draw_boxes**(*boxes*, *box_format='xywh'*, *color='blue'*, *labels=None*, *textkw=None*, *ax=None*)

Parameters

- **boxes** (*list*) – list of coordindates in xywh, tlbr, or cxywh format
- **box_format** (*str*) – specify how boxes are formated xywh is the top left x and y pixel width and height cxywh is the center xy pixel width and height tlbr is the top left xy and the bottom right xy
- **color** (*str*) – edge color of the boxes
- **labels** (*list*) – if specified, plots a text annotation on each box

Example

```
>>> qtensure()  # xdoc: +SKIP
>>> bboxes = [[.1, .1, .6, .3], [.3, .5, .5, .6]]
>>> col = draw_boxes(bboxes)
```

graphid.util.**draw_line_segments**(*pts1*, *pts2*, *ax=None*, ***kwargs*)

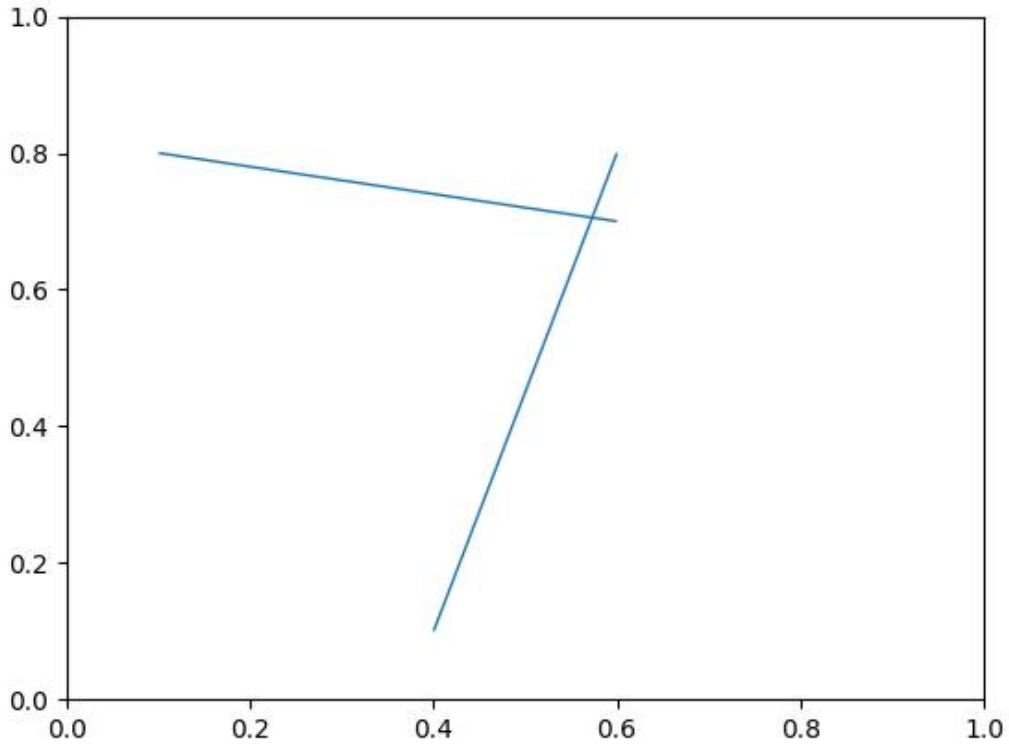
draws *N* line segments between *N* pairs of points

Parameters

- **pts1** (*ndarray*) – Nx2
- **pts2** (*ndarray*) – Nx2
- **ax** (*None*) – (default = None)
- ****kwargs** – lw, alpha, colors

Example

```
>>> pts1 = np.array([(.1, .8), (.6, .8)])
>>> pts2 = np.array([(.6, .7), (.4, .1)])
>>> figure(fnum=None)
>>> draw_line_segments(pts1, pts2)
>>> # xdoc: +REQUIRES(--show)
>>> import matplotlib.pyplot as plt
>>> ax = plt.gca()
>>> ax.set_xlim(0, 1)
>>> ax.set_ylim(0, 1)
>>> show_if_requested()
```



graphid.util.**ensure_fnum**(*fnum*)

graphid.util.**extract_axes_extents**(*fig*, *combine=False*, *pad=0.0*)

graphid.util.**figure**(*fnum=None*, *pnum=(1, 1, 1)*, *title=None*, *figtitle=None*, *doclf=False*, *docla=False*, *projection=None*, ***kwargs*)

http://matplotlib.org/users/gridspec.html

Parameters

- **fnum** (*int*) – fignum = figure number
- **pnum** (*int, str, or tuple(int, int, int)*) – plotnum = plot tuple
- **title** (*str*) – (default = None)

- **figtitle** (*None*) – (default = None)
- **docla** (*bool*) – (default = False)
- **doclf** (*bool*) – (default = False)

Returns
fig

Return type
mpl.Figure

Example

```
>>> import matplotlib.pyplot as plt
>>> fnum = 1
>>> fig = figure(fnum, (2, 2, 1))
>>> plt.gca().text(0.5, 0.5, "ax1", va="center", ha="center")
>>> fig = figure(fnum, (2, 2, 2))
>>> plt.gca().text(0.5, 0.5, "ax2", va="center", ha="center")
>>> show_if_requested()
```

Example

```
>>> import matplotlib.pyplot as plt
>>> fnum = 1
>>> fig = figure(fnum, (2, 2, 1))
>>> plt.gca().text(0.5, 0.5, "ax1", va="center", ha="center")
>>> fig = figure(fnum, (2, 2, 2))
>>> plt.gca().text(0.5, 0.5, "ax2", va="center", ha="center")
>>> fig = figure(fnum, (2, 4, (1, slice(1, None))))
>>> plt.gca().text(0.5, 0.5, "ax3", va="center", ha="center")
>>> show_if_requested()
```

graphid.util.**get_axis_xy_width_height**(*ax=None*, *xaug=0*, *yaug=0*, *waug=0*, *haug=0*)

gets geometry of a subplot

graphid.util.**imshow**(*img*, *fnum=None*, *title=None*, *figtitle=None*, *pnum=None*, *interpolation='nearest'*, *cmap=None*, *heatmap=False*, *data_colorbar=False*, *xlabel=None*, *redraw_image=True*, *colorspace='bgr'*, *ax=None*, *alpha=None*, *norm=None*, ***kwargs*)

Parameters

- **img** (*ndarray*) – image data
- **fnum** (*int*) – figure number
- **colorspace** (*str*) – if the data is 3-4 channels, this indicates the colorspace 1 channel data is assumed grayscale. 4 channels assumes alpha.
- **title** (*str*)
- **figtitle** (*None*)
- **pnum** (*tuple*) – plot number
- **interpolation** (*str*) – other interpolations = nearest, bicubic, bilinear

- **cmap** (*None*)
- **heatmap** (*bool*)
- **data_colorbar** (*bool*)
- **darken** (*None*)
- **redraw_image** (*bool*) – used when calling imshow over and over. if false doesnt do the image part.

Returns
(fig, ax)

Return type
tuple

Kwargs:
docla, doclf, projection

Returns
(fig, ax)

Return type
tuple

graphid.util.**legend**(*loc='best'*, *fontproperties=None*, *size=None*, *fc='w'*, *alpha=1*, *ax=None*, *handles=None*)

Parameters

- **loc** (*str*) – (default = 'best')
- **fontproperties** (*None*) – (default = None)
- **size** (*None*) – (default = None)

graphid.util.**make_heatmask**(*probs*, *cmap='plasma'*, *with_alpha=True*)

Colorizes a single-channel intensity mask (with an alpha channel)

graphid.util.**multi_plot**(*xdata=None*, *ydata=[]*, ***kwargs*)

plots multiple lines, bars, etc...

This is the big function that implements almost all of the heavy lifting in this file. Any function not using this should probably find a way to use it. It is pretty general and relatively clean.

Parameters

- **xdata** (*ndarray*) – can also be a list of arrays
- **ydata** (*list or dict of ndarrays*) – can also be a single array
- ****kwargs** –

  Misc:
  fnum, pnum, use_legend, legend_loc

  Labels:
  xlabel, ylabel, title, figtitle ticksize, titlesize, legendsize, labelsize

  Grid:
  gridlinewidth, gridlinestyle

  Ticks:
  num_xticks, num_yticks, tickwidth, ticklength, ticksize

Data:
xmin, xmax, ymin, ymax, spread_list # can append _list to any of these # these can be dictionaries if ydata was also a dict

plot_kw_keys = ['label', 'color', 'marker', 'markersize',
'markeredgewidth', 'linewidth', 'linestyle']

any plot_kw key can be a scalar (corresponding to all ydatas), a list if ydata was specified as a list, or a dict if ydata was specified as a dict.

kind = ['bar', 'plot', ...]

if kind='plot':
spread

if kind='bar':
stacked, width

References

matplotlib.org/examples/api/barchart_demo.html

Example

```
>>> xdata = [1, 2, 3, 4, 5]
>>> ydata_list = [[1, 2, 3, 4, 5], [3, 3, 3, 3, 3], [5, 4, np.nan, 2, 1], [4, 3, np.
↪nan, 1, 0]]
>>> kwargs = {'label': ['spam', 'eggs', 'jamμ', 'pram'],  'linestyle': '-'}
>>> #fig = multi_plot(xdata, ydata_list, title='$\phi_1(\\vec{x})$', xlabel='\nfds',␣
↪**kwargs)
>>> fig = multi_plot(xdata, ydata_list, title='μμμ', xlabel='\nfdsμμμ', **kwargs)
>>> show_if_requested()
```

Example

```
>>> fig1 = multi_plot([1, 2, 3], [4, 5, 6])
>>> fig2 = multi_plot([1, 2, 3], [4, 5, 6], fnum=4)
>>> show_if_requested()
```

graphid.util.**next_fnum**(*new_base=None*)

graphid.util.**pan_factory**(*ax=None*)

graphid.util.**pandas_plot_matrix**(*df*, *rot=90*, *ax=None*, *grid=True*, *label=None*, *zerodiag=False*, *cmap='viridis'*, *showvals=False*, *logscale=True*)

graphid.util.**qtensure**()

graphid.util.**relative_text**(*pos*, *text*, *ax=None*, *offset=None*, ***kwargs*)

Places text on axes in a relative position

Parameters

- **pos** (*tuple*) – relative xy position
- **text** (*str*) – text

- **ax** (*None*) – (default = None)
- **offset** (*None*) – (default = None)
- ****kwargs** – horizontalalignment, verticalalignment, roffset, ha, va, fontsize, fontproperties, fontproperties, clip_on

CommandLine

```
python -m graphid.util.mplutil relative_text --show
```

Example

```
>>> from graphid import util
>>> import matplotlib as mpl
>>> x = .5
>>> y = .5
>>> util.figure()
>>> txt = 'Hello World'
>>> family = 'monospace'
>>> family = 'CMU Typewriter Text'
>>> fontproperties = mpl.font_manager.FontProperties(family=family,
>>>                                                  size=42)
>>> relative_text((x, y), txt, halign='center',
>>>               fontproperties=fontproperties)
>>> util.show_if_requested()
```

graphid.util.**reverse_colormap**(*cmap*)

References

http://nbviewer.ipython.org/github/kwinkunks/notebooks/blob/master/Matteo_colourmaps.ipynb

graphid.util.**save_parts**(*fig*, *fpath*, *grouped_axes=None*, *dpi=None*)

FIXME: this works in mpl 2.0.0, but not 2.0.2

Parameters

- **fig** (*?*)
- **fpath** (*str*) – file path string
- **dpi** (*None*) – (default = None)

Returns

subpaths

Return type

list

CommandLine

```
python -m draw_func2 save_parts
```

graphid.util.**scores_to_cmap**(*scores*, *colors=None*, *cmap_='hot'*)

graphid.util.**scores_to_color**(*score_list*, *cmap_='hot'*, *logscale=False*, *reverse_cmap=False*, *custom=False*, *val2_customcolor=None*, *score_range=None*, *cmap_range=(0.1, 0.9)*)

Other good colormaps are 'spectral', 'gist_rainbow', 'gist_ncar', 'Set1', 'Set2', 'Accent' # TODO: plasma

Parameters

- **score_list** (*list*)
- **cmap_** (*str*) – defaults to hot
- **logscale** (*bool*)
- **cmap_range** (*tuple*) – restricts to only a portion of the cmap to avoid extremes

Returns

<class '_ast.ListComp'>

graphid.util.**set_figtitle**(*figtitle*, *subtitle=''*, *forcefignum=True*, *incanvas=True*, *size=None*, *fontfamily=None*, *fontweight=None*, *fig=None*)

Parameters

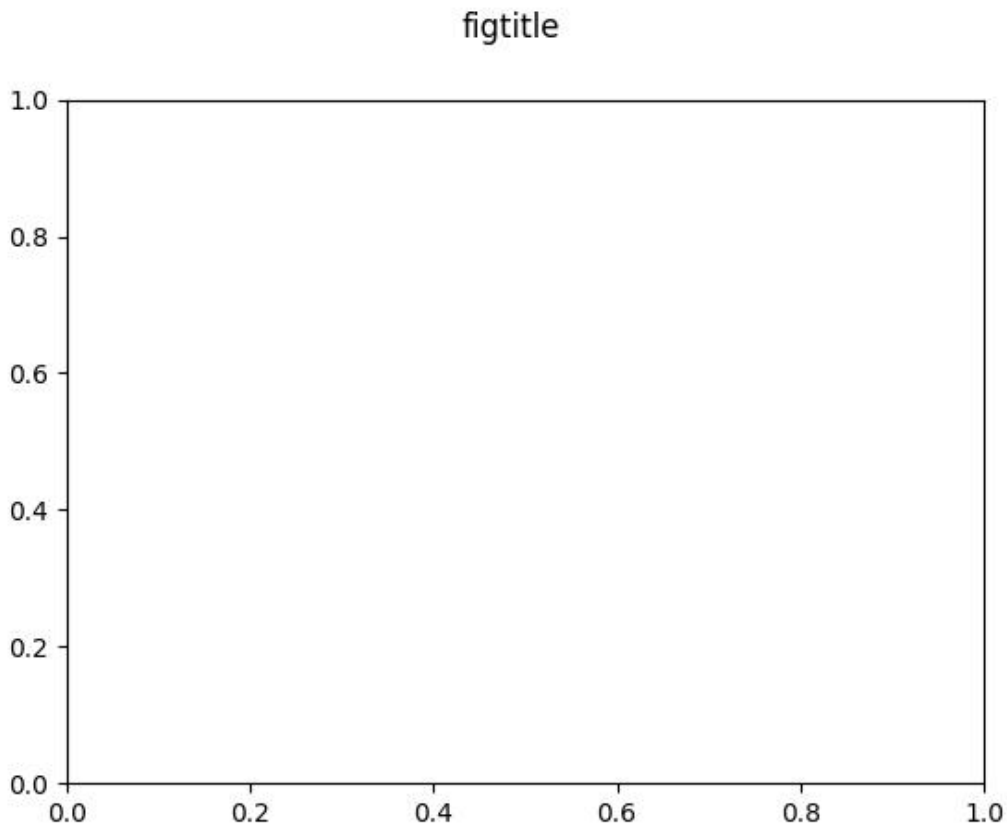
- **figtitle** (*?*)
- **subtitle** (*str*) – (default = '')
- **forcefignum** (*bool*) – (default = True)
- **incanvas** (*bool*) – (default = True)
- **fontfamily** (*None*) – (default = None)
- **fontweight** (*None*) – (default = None)
- **size** (*None*) – (default = None)
- **fig** (*None*) – (default = None)

CommandLine

```
python -m .custom_figure set_figtitle --show
```

Example

```
>>> # DISABLE_DOCTEST
>>> fig = figure(fnum=1, doclf=True)
>>> result = set_figtitle(figtitle='figtitle', fig=fig)
>>> # xdoc: +REQUIRES(--show)
>>> show_if_requested()
```



graphid.util.**show_if_requested**(*N=1*)

Used at the end of tests. Handles command line arguments for saving figures

Referencse:
http://stackoverflow.com/questions/4325733/save-a-subplot-in-matplotlib

graphid.util.**zoom_factory**(*ax=None*, *zoomable_list=[]*, *base_scale=1.1*)

References

https://gist.github.com/tacaswell/3144287 http://stackoverflow.com/questions/11551049/matplotlib-plot-zooming-with-scroll-wheel

graphid.util.**demodata_oldnames**(*n_incon_groups=10*, *n_con_groups=2*, *n_per_con=5*, *n_per_incon=5*, *con_sep=4*, *n_empty_groups=0*)

graphid.util.**find_consistent_labeling**(*grouped_oldnames*, *extra_prefix='_extra_name'*, *verbose=False*)

Solves a a maximum bipirtite matching problem to find a consistent name assignment that minimizes the number of annotations with different names. For each new grouping of annotations we assign

For each group of annotations we must assign them all the same name, either from

To reduce the running time

Parameters
gropued_oldnames (*list*) – A group of old names where the grouping is based on new names. For instance:

Given:
aids = [1, 2, 3, 4, 5] old_names = [0, 1, 1, 1, 0] new_names = [0, 0, 1, 1, 0]

The grouping is
[[0, 1, 0], [1, 1]]

This lets us keep the old names in a split case and re-use exising names and make minimal changes to current annotation names while still being consistent with the new and improved grouping.

The output will be:
[0, 1]

Meaning that all annots in the first group are assigned the name 0 and all annots in the second group are assigned the name 1.

References

http://stackoverflow.com/questions/1398822/assignment-problem-numpy

Example

```
>>> grouped_oldnames = demodata_oldnames(25, 15,  5, n_per_incon=5)
>>> new_names = find_consistent_labeling(grouped_oldnames, verbose=1)
>>> grouped_oldnames = demodata_oldnames(0, 15,  5, n_per_incon=1)
>>> new_names = find_consistent_labeling(grouped_oldnames, verbose=1)
>>> grouped_oldnames = demodata_oldnames(0, 0, 0, n_per_incon=1)
>>> new_names = find_consistent_labeling(grouped_oldnames, verbose=1)
```

Example

```
>>> # xdoctest: +REQUIRES(module:timerit)
>>> import timerit
>>> ydata = []
>>> xdata = list(range(10, 150, 50))
>>> for x in xdata:
>>>     print('x = %r' % (x,))
>>>     grouped_oldnames = demodata_oldnames(x, 15,  5, n_per_incon=5)
>>>     t = timerit.Timerit(3, verbose=1)
>>>     for timer in t:
>>>         with timer:
>>>             new_names = find_consistent_labeling(grouped_oldnames)
>>>     ydata.append(t.min())
>>> # xdoc: +REQUIRES(--show)
>>> import plottool_ibeis as pt
>>> pt.qtensure()
>>> pt.multi_plot(xdata, [ydata])
>>> util.show_if_requested()
```

Example

```
>>> grouped_oldnames = [['a', 'b', 'c'], ['b', 'c'], ['c', 'e', 'e']]
>>> new_names = find_consistent_labeling(grouped_oldnames, verbose=1)
>>> result = ub.urepr(new_names)
>>> print(new_names)
['a', 'b', 'e']
```

Example

```
>>> grouped_oldnames = [['a', 'b'], ['a', 'a', 'b'], ['a']]
>>> new_names = find_consistent_labeling(grouped_oldnames)
>>> result = ub.urepr(new_names)
>>> print(new_names)
['b', 'a', '_extra_name0']
```

Example

```
>>> grouped_oldnames = [['a', 'b'], ['e'], ['a', 'a', 'b'], [], ['a'], ['d']]
>>> new_names = find_consistent_labeling(grouped_oldnames)
>>> result = ub.urepr(new_names)
>>> print(new_names)
['b', 'e', 'a', '_extra_name0', '_extra_name1', 'd']
```

Example

```
>>> grouped_oldnames = [[], ['a', 'a'], [],
>>                      ['a', 'a', 'a', 'a', 'a', 'a', 'a', 'b'], ['a']]
>>> new_names = find_consistent_labeling(grouped_oldnames)
>>> result = ub.urepr(new_names)
>>> print(new_names)
['_extra_name0', 'a', '_extra_name1', 'b', '_extra_name2']
```

graphid.util.**simple_munkres**(*part_oldnames*)

Defines a munkres problem to solve name rectification.

Notes

We create a matrix where each rows represents a group of annotations in the same PCC and each column represents an original name. If there are more PCCs than original names the columns are padded with extra values. The matrix is first initialized to be negative infinity representing impossible assignments. Then for each column representing a padded name, we set we its value to 1 indicating that each new name could be assigned to a padded name for some small profit. Finally, let f_{rc} be the the number of annotations in row r with an original name of c. Each matrix value (r, c) is set to $f_{rc} + 1$ if $f_{rc} > 0$, to represent how much each name ``wants" to be labeled with a particular original name, and the extra one ensures that these original names are always preferred over padded names.

Example

```
>>> part_oldnames = [['a', 'b'], ['b', 'c'], ['c', 'a', 'a']]
>>> new_names = simple_munkres(part_oldnames)
>>> result = ub.urepr(new_names)
>>> print(new_names)
['b', 'c', 'a']
```

Example

```
>>> part_oldnames = [[], ['a', 'a'], [],
>>>                  ['a', 'a', 'a', 'a', 'a', 'a', 'a', 'b'], ['a']]
>>> new_names = simple_munkres(part_oldnames)
>>> result = ub.urepr(new_names)
>>> print(new_names)
[None, 'a', None, 'b', None]
```

Example

```
>>> part_oldnames = [[], ['b'], ['a', 'b', 'c'], ['b', 'c'], ['c', 'e', 'e']]
>>> new_names = find_consistent_labeling(part_oldnames)
>>> result = ub.urepr(new_names)
>>> print(new_names)
['_extra_name0', 'b', 'a', 'c', 'e']
```

Profit Matrix
b a c e _0

0 -10 -10 -10 -10 1 1 2 -10 -10 -10 1 2 2 2 2 -10 1 3 2 -10 2 -10 1 4 -10 -10 2 3 1

class graphid.util.**DynConnGraph**(**args*, ***kwargs*)

Bases: Graph, *GraphHelperMixin*

Dynamically connected graph.

Maintains a data structure parallel to a normal networkx graph that maintains dynamic connectivity for fast connected compoment queries.

Underlying Data Structures and limitations are

1.1.3.2.1 Data Structure | Insertion | Deletion | CC Find |

UnionFind | lg(n) | n | No UnionFind2 | n* | n | 1 EulerTourForest | lg^2(n) | lg^2(n) | lg(n) / lglg(n) - - Ammortized

- O(n) is worst case, but it seems to be very quick in practice. The average runtime should be close to lg(n), but I'm writing this doc 8 months after working on this algo, so I may not remember exactly.

References

https://courses.csail.mit.edu/6.851/spring14/lectures/L20.pdf https://courses.csail.mit.edu/6.851/spring14/lectures/L20.html http://cs.stackexchange.com/questions/33595/maintaining-connecte https://en.wikipedia.org/wiki/Dynamic_connectivity#Fully_dynamic_connectivity

Example

```
>>> self = DynConnGraph()
>>> self.add_edges_from([(1, 2), (2, 3), (4, 5), (6, 7), (7, 4)])
>>> self.add_edges_from([(10, 20), (20, 30), (40, 50), (60, 70), (70, 40)])
>>> self._ccs
>>> u, v = 20, 1
>>> assert self.node_label(u) != self.node_label(v)
>>> assert self.connected_to(u) != self.connected_to(v)
>>> self.add_edge(u, v)
>>> assert self.node_label(u) == self.node_label(v)
>>> assert self.connected_to(u) == self.connected_to(v)
>>> self.remove_edge(u, v)
>>> assert self.node_label(u) != self.node_label(v)
>>> assert self.connected_to(u) != self.connected_to(v)
>>> ccs = list(self.connected_components())
>>> # xdoctest: +REQUIRES(--show)
>>> import plottool_ibeis as pt
>>> pt.qtensure()
>>> pt.show_nx(self)
```

todo: check if nodes exist when adding

clear()

number_of_components()

component(*label*)

component_nodes(*label*)

connected_to(*node*)

node_label(*node*)

Example

```
>>> self = DynConnGraph()
>>> self.add_edges_from([(1, 2), (2, 3), (4, 5), (6, 7)])
>>> assert self.node_label(2) == self.node_label(1)
>>> assert self.node_label(2) != self.node_label(4)
```

node_labels(**nodes*)

are_nodes_connected(*u*, *v*)

connected_components()

Example

```
>>> self = DynConnGraph()
>>> self.add_edges_from([(1, 2), (2, 3), (4, 5), (6, 7)])
>>> ccs = list(self.connected_components())
>>> result = 'ccs = {}'.format(ub.urepr(ccs, nl=0))
>>> print(result)
ccs = [{1, 2, 3}, {4, 5}, {6, 7}]
```

component_labels()

_cut(*u*, *v*)

Decremental connectivity (slow)

_union(*u*, *v*)

Incremental connectivity (fast)

_add_node(*n*)

_remove_node(*n*)

add_edge(*u*, *v*, ***attr*)

Example

```
>>> self = DynConnGraph()
>>> self.add_edges_from([(1, 2), (2, 3), (4, 5), (6, 7), (7, 4)])
>>> assert self._ccs == {1: {1, 2, 3}, 4: {4, 5, 6, 7}}
>>> self.add_edge(1, 5)
>>> assert self._ccs == {1: {1, 2, 3, 4, 5, 6, 7}}
```

add_edges_from(*ebunch*, ***attr*)

add_node(*n*, ***attr*)

add_nodes_from(*nodes*, ***attr*)

remove_edge(*u*, *v*)

Example

```
>>> self = DynConnGraph()
>>> self.add_edges_from([(1, 2), (2, 3), (4, 5), (6, 7), (7, 4)])
>>> assert self._ccs == {1: {1, 2, 3}, 4: {4, 5, 6, 7}}
>>> self.add_edge(1, 5)
>>> assert self._ccs == {1: {1, 2, 3, 4, 5, 6, 7}}
>>> self.remove_edge(1, 5)
>>> assert self._ccs == {1: {1, 2, 3}, 4: {4, 5, 6, 7}}
```

remove_edges_from(*ebunch*)

remove_nodes_from(*nodes*)

remove_node(*n*)

Example

```
>>> self = DynConnGraph()
>>> self.add_edges_from([(1, 2), (2, 3), (4, 5), (5, 6), (6, 7), (7, 8), (8,␣
↪9)])
>>> assert self._ccs == {1: {1, 2, 3}, 4: {4, 5, 6, 7, 8, 9}}
>>> self.remove_node(2)
>>> assert self._ccs == {1: {1}, 3: {3}, 4: {4, 5, 6, 7, 8, 9}}
>>> self.remove_node(7)
>>> assert self._ccs == {1: {1}, 3: {3}, 4: {4, 5, 6}, 8: {8, 9}}
```

subgraph(*nbunch*, *dynamic=False*)

class graphid.util.**GraphHelperMixin**

Bases: NiceRepr

Ensures that we always return edges in a consistent order

has_nodes(*nodes*)

has_edges(*edges*)

edges(*nbunch=None*, *data=False*, *default=None*)

class graphid.util.**NiceGraph**(*incoming_graph_data=None*, ***attr*)

Bases: Graph, *GraphHelperMixin*

Initialize a graph with edges, name, or graph attributes.

Parameters

- **incoming_graph_data** (*input graph (optional, default: None)*) – Data to initialize graph. If None (default) an empty graph is created. The data can be an edge list, or any NetworkX graph object. If the corresponding optional Python packages are installed the data can also be a 2D NumPy array, a SciPy sparse array, or a PyGraphviz graph.
- **attr** (*keyword arguments, optional (default= no attributes)*) – Attributes to add to graph as key=value pairs.

See also:

convert

Examples

```
>>> G = nx.Graph()  # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G = nx.Graph(name="my graph")
>>> e = [(1, 2), (2, 3), (3, 4)]  # list of edges
>>> G = nx.Graph(e)
```

Arbitrary graph attribute pairs (key=value) may be assigned

```
>>> G = nx.Graph(e, day="Friday")
>>> G.graph
{'day': 'Friday'}
```

class graphid.util.**nx_UnionFind**(*elements=None*)

Bases: object

Based off code in networkx

clear()

rebalance(*elements=None*)

to_sets()

union(**objects*)

Find the sets containing the objects and merge them all.

remove_entire_cc(*elements*)

add_element(*x*)

add_elements(*elements*)

graphid.util.**assert_raises**(*ex_type*, *func*, **args*, ***kwargs*)

Checks that a function raises an error when given specific arguments.

Parameters

- **ex_type** (*Exception*) – exception type
- **func** (*callable*) – live python function

Example

```
>>> ex_type = AssertionError
>>> func = len
>>> assert_raises(ex_type, assert_raises, ex_type, func, [])
>>> assert_raises(ValueError, [].index, 0)
```

graphid.util.**bfs_conditional**(*G*, *source*, *reverse=False*, *keys=True*, *data=False*, *yield_nodes=True*, *yield_if=None*, *continue_if=None*, *visited_nodes=None*, *yield_source=False*)

Produce edges in a breadth-first-search starting at source, but only return nodes that satisfiy a condition, and only iterate past a node if it satisfies a different condition.

conditions are callables that take (G, child, edge) and return true or false

Example

```
>>> import networkx as nx
>>> G = nx.Graph()
>>> G.add_edges_from([(1, 2), (1, 3), (2, 3), (2, 4)])
>>> continue_if = lambda G, child, edge: True
>>> result = list(bfs_conditional(G, 1, yield_nodes=False))
>>> print(result)
[(1, 2), (1, 3), (2, 1), (2, 3), (2, 4), (3, 1), (3, 2), (4, 2)]
```

Example

```
>>> import networkx as nx
>>> G = nx.Graph()
>>> continue_if = lambda G, child, edge: (child % 2 == 0)
>>> yield_if = lambda G, child, edge: (child % 2 == 1)
>>> G.add_edges_from([(0, 1), (1, 3), (3, 5), (5, 10),
>>>                   (4, 3), (3, 6),
>>>                   (0, 2), (2, 4), (4, 6), (6, 10)])
>>> result = list(bfs_conditional(G, 0, continue_if=continue_if,
>>>                                  yield_if=yield_if))
>>> print(result)
[1, 3, 5]
```

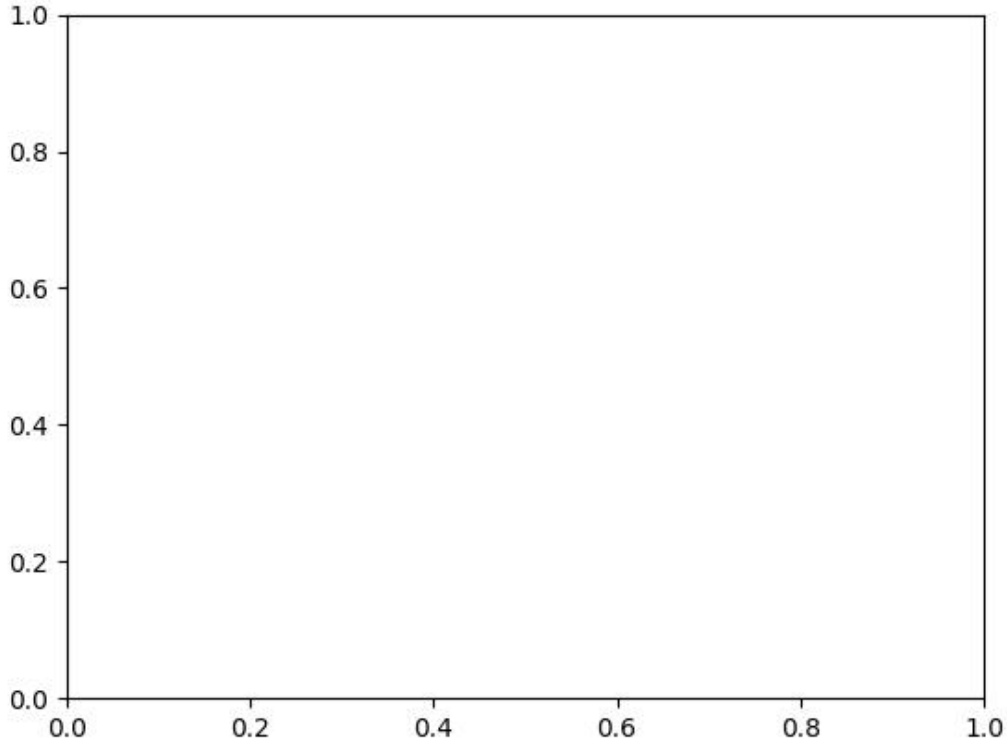
graphid.util.**complement_edges**(*G*)

graphid.util.**demodata_bridge**()

graphid.util.**demodata_tarjan_bridge**()

Example

```
>>> from graphid import util
>>> G = demodata_tarjan_bridge()
>>> # xdoc: +REQUIRES(--show)
>>> util.show_nx(G)
>>> util.show_if_requested()
```



graphid.util.**diag_product**(*s1*, *s2*)

Does product, but iterates over the diagonal first

graphid.util.**dict_take_column**(*list_of_dicts_*, *colkey*, *default=None*)

graphid.util.**e_**(*u*, *v*)

graphid.util.**edge_df**(*graph*, *edges*, *ignore=None*)

graphid.util.**edges_between**(*graph*, *nodes1*, *nodes2=None*, *assume_disjoint=False*, *assume_dense=True*)

Get edges between two components or within a single component

Parameters

- **graph** (*nx.Graph*) – the graph
- **nodes1** (*set*) – list of nodes
- **nodes2** (*set*) – if None it is equivlanet to nodes2=nodes1 (default=None)
- **assume_disjoint** (*bool*) – skips expensive check to ensure edges arnt returned twice (default=False)

Example

```
>>> edges = [
>>>     (1, 2), (2, 3), (3, 4), (4, 1), (4, 3),  # cc 1234
>>>     (1, 5), (7, 2), (5, 1),  # cc 567 / 5678
>>>     (7, 5), (5, 6), (8, 7),
>>> ]
>>> digraph = nx.DiGraph(edges)
>>> graph = nx.Graph(edges)
>>> nodes1 = [1, 2, 3, 4]
>>> nodes2 = [5, 6, 7]
>>> n2 = sorted(edges_between(graph, nodes1, nodes2))
>>> n4 = sorted(edges_between(graph, nodes1))
>>> n5 = sorted(edges_between(graph, nodes1, nodes1))
>>> n1 = sorted(edges_between(digraph, nodes1, nodes2))
>>> n3 = sorted(edges_between(digraph, nodes1))
>>> print('n2 == %r' % (n2,))
>>> print('n4 == %r' % (n4,))
>>> print('n5 == %r' % (n5,))
>>> print('n1 == %r' % (n1,))
>>> print('n3 == %r' % (n3,))
>>> assert n2 == ([(1, 5), (2, 7)]), '2'
>>> assert n4 == ([(1, 2), (1, 4), (2, 3), (3, 4)]), '4'
>>> assert n5 == ([(1, 2), (1, 4), (2, 3), (3, 4)]), '5'
>>> assert n1 == ([(1, 5), (5, 1), (7, 2)]), '1'
>>> assert n3 == ([(1, 2), (2, 3), (3, 4), (4, 1), (4, 3)]), '3'
>>> n6 = sorted(edges_between(digraph, nodes1 + [6], nodes2 + [1, 2], assume_
↪dense=False))
>>> print('n6 = %r' % (n6,))
>>> n6 = sorted(edges_between(digraph, nodes1 + [6], nodes2 + [1, 2], assume_
↪dense=True))
>>> print('n6 = %r' % (n6,))
>>> assert n6 == ([(1, 2), (1, 5), (2, 3), (4, 1), (5, 1), (5, 6), (7, 2)]), '6'
```

graphid.util.**edges_cross**(*graph*, *nodes1*, *nodes2*)

Finds edges between two sets of disjoint nodes. Running time is O(len(nodes1) * len(nodes2))

Parameters

- **graph** (*nx.Graph*) – an undirected graph
- **nodes1** (*set*) – set of nodes disjoint from *nodes2*
- **nodes2** (*set*) – set of nodes disjoint from *nodes1*.

graphid.util.**edges_inside**(*graph*, *nodes*)

Finds edges within a set of nodes Running time is O(len(nodes) ** 2)

Parameters

- **graph** (*nx.Graph*) – an undirected graph
- **nodes1** (*set*) – a set of nodes

graphid.util.**edges_outgoing**(*graph*, *nodes*)

Finds edges leaving a set of nodes. Average running time is O(len(nodes) * ave_degree(nodes)) Worst case running time is O(G.number_of_edges()).

Parameters

- **graph** (*nx.Graph*) – a graph
- **nodes** (*set*) – set of nodes

Example

```
>>> G = demodata_bridge()
>>> nodes = {1, 2, 3, 4}
>>> outgoing = edges_outgoing(G, nodes)
>>> assert outgoing == {(3, 5), (4, 8)}
```

graphid.util.**ensure_multi_index**(*index*, *names*)

graphid.util.**graph_info**(*graph*, *ignore=None*, *stats=False*, *verbose=False*)

graphid.util.**group_name_edges**(*g*, *node_to_label*)

graphid.util.**is_complete**(*G*, *self_loops=False*)

graphid.util.**is_k_edge_connected**(*G*, *k*)

graphid.util.**itake_column**(*list_*, *colx*)

iterator version of get_list_column

graphid.util.**k_edge_augmentation**(*G*, *k*, *avail=None*, *partial=False*)

graphid.util.**list_roll**(*list_*, *n*)

Like numpy.roll for python lists

Parameters

- **list_** (*list*)
- **n** (*int*)

Return type

list

References

http://stackoverflow.com/questions/9457832/python-list-rotation

Example

```
>>> list_ = [1, 2, 3, 4, 5]
>>> n = 2
>>> result = list_roll(list_, n)
>>> print(result)
[4, 5, 1, 2, 3]
```

graphid.util.**nx_delete_None_edge_attr**(*graph*, *edges=None*)

graphid.util.**nx_delete_None_node_attr**(*graph*, *nodes=None*)

graphid.util.**nx_delete_edge_attr**(*graph*, *name*, *edges=None*)

Removes an attributes from specific edges in the graph

Doctest

```
>>> G = nx.karate_club_graph()
>>> nx.set_edge_attributes(G, name='spam', values='eggs')
>>> nx.set_edge_attributes(G, name='foo', values='bar')
>>> assert len(nx.get_edge_attributes(G, 'spam')) == 78
>>> assert len(nx.get_edge_attributes(G, 'foo')) == 78
>>> nx_delete_edge_attr(G, ['spam', 'foo'], edges=[(1, 2)])
>>> assert len(nx.get_edge_attributes(G, 'spam')) == 77
>>> assert len(nx.get_edge_attributes(G, 'foo')) == 77
>>> nx_delete_edge_attr(G, ['spam'])
>>> assert len(nx.get_edge_attributes(G, 'spam')) == 0
>>> assert len(nx.get_edge_attributes(G, 'foo')) == 77
```

Doctest

```
>>> G = nx.MultiGraph()
>>> G.add_edges_from([(1, 2), (2, 3), (3, 4), (4, 5), (4, 5), (1, 2)])
>>> nx.set_edge_attributes(G, name='spam', values='eggs')
>>> nx.set_edge_attributes(G, name='foo', values='bar')
>>> assert len(nx.get_edge_attributes(G, 'spam')) == 6
>>> assert len(nx.get_edge_attributes(G, 'foo')) == 6
>>> nx_delete_edge_attr(G, ['spam', 'foo'], edges=[(1, 2, 0)])
>>> assert len(nx.get_edge_attributes(G, 'spam')) == 5
>>> assert len(nx.get_edge_attributes(G, 'foo')) == 5
>>> nx_delete_edge_attr(G, ['spam'])
>>> assert len(nx.get_edge_attributes(G, 'spam')) == 0
>>> assert len(nx.get_edge_attributes(G, 'foo')) == 5
```

graphid.util.**nx_delete_node_attr**(*graph*, *name*, *nodes=None*)

Removes node attributes

Doctest

```
>>> G = nx.karate_club_graph()
>>> nx.set_node_attributes(G, name='foo', values='bar')
>>> datas = nx.get_node_attributes(G, 'club')
>>> assert len(nx.get_node_attributes(G, 'club')) == 34
>>> assert len(nx.get_node_attributes(G, 'foo')) == 34
>>> nx_delete_node_attr(G, ['club', 'foo'], nodes=[1, 2])
>>> assert len(nx.get_node_attributes(G, 'club')) == 32
>>> assert len(nx.get_node_attributes(G, 'foo')) == 32
>>> nx_delete_node_attr(G, ['club'])
>>> assert len(nx.get_node_attributes(G, 'club')) == 0
>>> assert len(nx.get_node_attributes(G, 'foo')) == 32
```

graphid.util.**nx_edges**(*graph*, *keys=False*, *data=False*)

graphid.util.**nx_gen_edge_attrs**(*G*, *key*, *edges=None*, *default=NoParam*, *on_missing='error'*, *on_keyerr='default'*)

Improved generator version of nx.get_edge_attributes

Parameters

- **on_missing** (*str*) – Strategy for handling nodes missing from G. Can be {'error', 'default', 'filter'}. defaults to 'error'. is on_missing is not error, then we allow any edge even if the endpoints are not in the graph.
- **on_keyerr** (*str*) – Strategy for handling keys missing from node dicts. Can be {'error', 'default', 'filter'}. defaults to 'default' if default is specified, otherwise defaults to 'error'.

CommandLine

```
python -m graphid.util.nx_utils nx_gen_edge_attrs
```

Example

```
>>> from graphid import util
>>> from functools import partial
>>> G = nx.Graph([(1, 2), (2, 3), (3, 4)])
>>> nx.set_edge_attributes(G, name='part', values={(1, 2): 'bar', (2, 3): 'baz'})
>>> edges = [(1, 2), (2, 3), (3, 4), (4, 5)]
>>> func = partial(nx_gen_edge_attrs, G, 'part', default=None)
>>> #
>>> assert len(list(func(on_missing='error', on_keyerr='default'))) == 3
>>> assert len(list(func(on_missing='error', on_keyerr='filter'))) == 2
>>> util.assert_raises(KeyError, list, func(on_missing='error', on_keyerr='error'))
>>> #
>>> assert len(list(func(edges, on_missing='filter', on_keyerr='default'))) == 3
>>> assert len(list(func(edges, on_missing='filter', on_keyerr='filter'))) == 2
>>> util.assert_raises(KeyError, list, func(edges, on_missing='filter', on_keyerr=
↪'error'))
>>> #
>>> assert len(list(func(edges, on_missing='default', on_keyerr='default'))) == 4
>>> assert len(list(func(edges, on_missing='default', on_keyerr='filter'))) == 2
>>> util.assert_raises(KeyError, list, func(edges, on_missing='default', on_keyerr=
↪'error'))
```

graphid.util.**nx_gen_edge_values**(*G*, *key*, *edges=None*, *default=NoParam*, *on_missing='error'*, *on_keyerr='default'*)

Generates attributes values of specific edges

Parameters

- **on_missing** (*str*) – Strategy for handling nodes missing from G. Can be {'error', 'default'}. defaults to 'error'.
- **on_keyerr** (*str*) – Strategy for handling keys missing from node dicts. Can be {'error', 'default'}. defaults to 'default' if default is specified, otherwise defaults to 'error'.

graphid.util.**nx_gen_node_attrs**(*G*, *key*, *nodes=None*, *default=NoParam*, *on_missing='error'*, *on_keyerr='default'*)

Improved generator version of nx.get_node_attributes

Parameters

- **on_missing** (*str*) – Strategy for handling nodes missing from G. Can be {'error', 'default', 'filter'}. defaults to 'error'.
- **on_keyerr** (*str*) – Strategy for handling keys missing from node dicts. Can be {'error', 'default', 'filter'}. defaults to 'default' if default is specified, otherwise defaults to 'error'.

Notes

strategies are:

error - raises an error if key or node does not exist
default - returns node, but uses value specified by default
filter - skips the node

Example

```
>>> # ENABLE_DOCTEST
>>> from graphid import util
>>> G = nx.Graph([(1, 2), (2, 3)])
>>> nx.set_node_attributes(G, name='part', values={1: 'bar', 3: 'baz'})
>>> nodes = [1, 2, 3, 4]
>>> #
>>> assert len(list(nx_gen_node_attrs(G, 'part', default=None, on_missing='error',␣
↪on_keyerr='default'))) == 3
>>> assert len(list(nx_gen_node_attrs(G, 'part', default=None, on_missing='error',␣
↪on_keyerr='filter'))) == 2
>>> assert_raises(KeyError, list, nx_gen_node_attrs(G, 'part', on_missing='error',␣
↪on_keyerr='error'))
>>> #
>>> assert len(list(nx_gen_node_attrs(G, 'part', nodes, default=None, on_missing=
↪'filter', on_keyerr='default'))) == 3
>>> assert len(list(nx_gen_node_attrs(G, 'part', nodes, default=None, on_missing=
↪'filter', on_keyerr='filter'))) == 2
>>> assert_raises(KeyError, list, nx_gen_node_attrs(G, 'part', nodes, on_missing=
↪'filter', on_keyerr='error'))
>>> #
>>> assert len(list(nx_gen_node_attrs(G, 'part', nodes, default=None, on_missing=
↪'default', on_keyerr='default'))) == 4
>>> assert len(list(nx_gen_node_attrs(G, 'part', nodes, default=None, on_missing=
↪'default', on_keyerr='filter'))) == 2
>>> assert_raises(KeyError, list, nx_gen_node_attrs(G, 'part', nodes, on_missing=
↪'default', on_keyerr='error'))
```

Example

```
>>> # DISABLE_DOCTEST
>>> # ALL CASES
>>> from graphid import util
>>> G = nx.Graph([(1, 2), (2, 3)])
>>> nx.set_node_attributes(G, name='full', values={1: 'A', 2: 'B', 3: 'C'})
>>> nx.set_node_attributes(G, name='part', values={1: 'bar', 3: 'baz'})
>>> nodes = [1, 2, 3, 4]
>>> attrs = dict(nx_gen_node_attrs(G, 'full'))
>>> input_grid = {
>>>     'nodes': [None, (1, 2, 3, 4)],
>>>     'key': ['part', 'full'],
>>>     'default': [ub.NoParam, None],
>>> }
>>> inputs = util.all_dict_combinations(input_grid)
>>> kw_grid = {
>>>     'on_missing': ['error', 'default', 'filter'],
>>>     'on_keyerr': ['error', 'default', 'filter'],
>>> }
>>> kws = util.all_dict_combinations(kw_grid)
>>> for in_ in inputs:
>>>     for kw in kws:
>>>         kw2 = ub.dict_union(kw, in_)
>>>         #print(kw2)
>>>         on_missing = kw['on_missing']
>>>         on_keyerr = kw['on_keyerr']
>>>         if on_keyerr == 'default' and in_['default'] is ub.NoParam:
>>>             on_keyerr = 'error'
>>>         will_miss = False
>>>         will_keyerr = False
>>>         if on_missing == 'error':
>>>             if in_['key'] == 'part' and in_['nodes'] is not None:
>>>                 will_miss = True
>>>             if in_['key'] == 'full' and in_['nodes'] is not None:
>>>                 will_miss = True
>>>         if on_keyerr == 'error':
>>>             if in_['key'] == 'part':
>>>                 will_keyerr = True
>>>             if on_missing == 'default':
>>>                 if in_['key'] == 'full' and in_['nodes'] is not None:
>>>                     will_keyerr = True
>>>         want_error = will_miss or will_keyerr
>>>         gen = nx_gen_node_attrs(G, **kw2)
>>>         try:
>>>             attrs = list(gen)
>>>         except KeyError:
>>>             if not want_error:
>>>                 raise AssertionError('should not have errored')
>>>         else:
>>>             if want_error:
>>>                 raise AssertionError('should have errored')
```

graphid.util.**nx_gen_node_values**(*G*, *key*, *nodes*, *default=NoParam*)

Generates attributes values of specific nodes

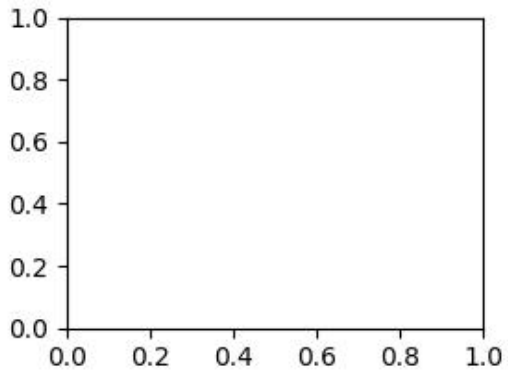
graphid.util.**nx_node_dict**(*G*)

graphid.util.**random_k_edge_connected_graph**(*size*, *k*, *p=0.1*, *rng=None*)

Super hacky way of getting a random k-connected graph

Example

```
>>> from graphid import util
>>> size, k, p = 25, 3, .1
>>> rng = util.ensure_rng(0)
>>> gs = []
>>> for x in range(4):
>>>     G = random_k_edge_connected_graph(size, k, p, rng)
>>>     gs.append(G)
>>> # xdoc: +REQUIRES(--show)
>>> pnum_ = util.PlotNums(nRows=2, nSubplots=len(gs))
>>> fnum = 1
>>> for g in gs:
>>>     util.show_nx(g, fnum=fnum, pnum=pnum_())
```



graphid.util.**take_column**(*list_*, *colx*)

accepts a list of (indexables) and returns a list of indexables can also return a list of list of indexables if colx is a

list

Parameters

- **list_** (*list*) – list of lists
- **colx** (*int or list*) – index or key in each sublist get item

Returns
list of selected items

Return type
list

Example0:

```
>>> list_ = [['a', 'b'], ['c', 'd']]
>>> colx = 0
>>> result = take_column(list_, colx)
>>> result = ub.urepr(result, nl=False)
>>> print(result)
['a', 'c']
```

Example1:

```
>>> list_ = [['a', 'b'], ['c', 'd']]
>>> colx = [1, 0]
>>> result = take_column(list_, colx)
>>> result = ub.urepr(result, nl=False)
>>> print(result)
[['b', 'a'], ['d', 'c']]
```

Example2:

```
>>> list_ = [{'spam': 'EGGS', 'ham': 'SPAM'}, {'spam': 'JAM', 'ham': 'PRAM'}]
>>> # colx can be a key or list of keys as well
>>> colx = ['spam']
>>> result = take_column(list_, colx)
>>> result = ub.urepr(result, nl=False)
>>> print(result)
[['EGGS'], ['JAM']]
```

class graphid.util.**PriorityQueue**(*items=None*, *ascending=True*)

Bases: NiceRepr

abstracted priority queue for our needs

Combines properties of dicts and heaps Uses a heap for fast minimum/maximum value search Uses a dict for fast read only operations

References

http://code.activestate.com/recipes/522995-priority-dict-a-priority-queue-with-updatable-prio/ https://stackoverflow.com/questions/33024215/built-in-max-heap-api-in-python

Example

```
>>> items = dict(a=42, b=29, c=40, d=95, e=10)
>>> self = PriorityQueue(items)
>>> print(self)
>>> assert len(self) == 5
>>> print(self.pop())
>>> assert len(self) == 4
>>> print(self.pop())
>>> assert len(self) == 3
>>> print(self.pop())
>>> print(self.pop())
>>> print(self.pop())
>>> assert len(self) == 0
```

Example

```
>>> items = dict(a=(1.0, (2, 3)), b=(1.0, (1, 2)), c=(.9, (3, 2)))
>>> self = PriorityQueue(items)
```

_rebuild()

get(*key*, *default=None*)

clear()

update(*items*)

delete_items(*key_list*)

peek()

Peek at the next item in the queue

peek_many(*n*)

Actually this can be quite inefficient

Example

```
>>> from graphid import util
>>> items = list(zip(range(256), range(256)))
>>> n = 32
>>> util.shuffle(items)
>>> self = PriorityQueue(items, ascending=False)
>>> self.peek_many(56)
```

pop_many(*n*)

pop(*key=NoParam*, *default=NoParam*)

Pop the next item off the queue

class graphid.util.**Boxes**(*data*, *format='xywh'*)

Bases: NiceRepr

Converts boxes between different formats as long as the last dimension contains 4 coordinates and the format is specified.

This is a convinience class, and should not not store the data for very long. The general idiom should be create class, convert data, and then get the raw data and let the class be garbage collected. This will help ensure that your code is portable and understandable if this class is not available.

Example

```
>>> # xdoctest: +IGNORE_WHITESPACE
>>> Boxes([25, 30, 15, 10], 'xywh')
<Boxes(xywh, array([25, 30, 15, 10]))>
>>> Boxes([25, 30, 15, 10], 'xywh').to_xywh()
<Boxes(xywh, array([25, 30, 15, 10]))>
>>> Boxes([25, 30, 15, 10], 'xywh').to_cxywh()
<Boxes(cxywh, array([32.5, 35. , 15. , 10. ]))>
>>> Boxes([25, 30, 15, 10], 'xywh').to_tlbr()
<Boxes(tlbr, array([25, 30, 40, 40]))>
>>> Boxes([25, 30, 15, 10], 'xywh').scale(2).to_tlbr()
<Boxes(tlbr, array([50., 60., 80., 80.]))>
```

Example

```
>>> datas = [
>>>     [1, 2, 3, 4],
>>>     [[1, 2, 3, 4], [4, 5, 6, 7]],
>>>     [[[1, 2, 3, 4], [4, 5, 6, 7]]],
>>> ]
>>> formats = ['xywh', 'cxywh', 'tlbr']
>>> for format1 in formats:
>>>     for data in datas:
>>>         self = box1 = Boxes(data, format1)
>>>         for format2 in formats:
>>>             box2 = box1.toformat(format2)
>>>             back = box2.toformat(format1)
>>>             assert box1 == back
```

classmethod random(*num=1*, *scale=1.0*, *format='xywh'*, *rng=None*)

Makes random boxes

Example

```
>>> # xdoctest: +IGNORE_WHITESPACE
>>> Boxes.random(3, rng=0, scale=100)
<Boxes(xywh,
    array([[27, 35, 30, 27],
           [21, 32, 21, 44],
           [48, 19, 39, 26]]))>
```

copy()

scale(*factor*)

works with tlbr, cxywh, xywh, xy, or wh formats

Example

```
>>> # xdoctest: +IGNORE_WHITESPACE
>>> Boxes(np.array([1, 1, 10, 10])).scale(2).data
array([ 2.,  2., 20., 20.])
>>> Boxes(np.array([[1, 1, 10, 10]])).scale((2, .5)).data
array([[ 2. ,  0.5, 20. ,  5. ]])
>>> Boxes(np.array([[10, 10]])).scale(.5).data
array([[5., 5.]])
```

shift(*amount*)

Example

```
>>> # xdoctest: +IGNORE_WHITESPACE
>>> Boxes([25, 30, 15, 10], 'xywh').shift(10)
<Boxes(xywh, array([35., 40., 15., 10.]))>
>>> Boxes([25, 30, 15, 10], 'xywh').shift((10, 0))
<Boxes(xywh, array([35., 30., 15., 10.]))>
>>> Boxes([25, 30, 15, 10], 'tlbr').shift((10, 5))
<Boxes(tlbr, array([35., 35., 25., 15.]))>
```

property center

property shape

property area

property components

classmethod _cat(*datas*)

toformat(*format*, *copy=True*)

to_extent(*copy=True*)

to_xywh(*copy=True*)

to_cxywh(*copy=True*)

to_tlbr(*copy=True*)

clip(*x_min*, *y_min*, *x_max*, *y_max*, *inplace=False*)

Clip boxes to image boundaries. If box is in tlbr format, inplace operation is an option.

Example

```
>>> # xdoctest: +IGNORE_WHITESPACE
>>> boxes = Boxes(np.array([[-10, -10, 120, 120], [1, -2, 30, 50]]), 'tlbr')
>>> clipped = boxes.clip(0, 0, 110, 100, inplace=False)
>>> assert np.any(boxes.data != clipped.data)
>>> clipped2 = boxes.clip(0, 0, 110, 100, inplace=True)
>>> assert clipped2.data is boxes.data
>>> assert np.all(clipped2.data == clipped.data)
>>> print(clipped)
<Boxes(tlbr,
    array([[  0,   0, 110, 100],
           [  1,   0,  30,  50]]))>
```

transpose()

compress(*flags*, *axis=0*, *inplace=False*)

Filters boxes based on a boolean criterion

Example

```
>>> self = Boxes([[25, 30, 15, 10]], 'tlbr')
>>> flags = [False]
```

graphid.util.**box_ious_py**(*boxes1*, *boxes2*, *bias=1*)

This is the fastest python implementation of bbox_ious I found

graphid.util.**grab_test_imgpath**(*key='astro.png'*, *allow_external=True*, *verbose=True*)

Gets paths to standard / fun test images. Downloads them if they dont exits

Parameters

- **key** (*str*) – one of the standard test images, e.g. astro.png, carl.jpg, ...
- **allow_external** (*bool*) – if True you can specify existing fpaths

Returns

testimg_fpath - filepath to the downloaded or cached test image.

Return type

str

Example

```
>>> testimg_fpath = grab_test_imgpath('carl.jpg')
>>> assert exists(testimg_fpath)
```

class graphid.util.**GRAPHVIZ_KEYS**

Bases: object

N = {'URL', 'area', 'color', 'colorscheme', 'comment', 'distortion', 'fillcolor', 'fixedsize', 'fontcolor', 'fontname', 'fontsize', 'gradientangle', 'group', 'height', 'href', 'id', 'image', 'imagepos', 'imagescale', 'label', 'labelloc', 'layer', 'margin', 'nojustify', 'ordering', 'orientation', 'penwidth', 'peripheries', 'pin', 'pos', 'rects', 'regular', 'root', 'samplepoints', 'shape', 'shapefile', 'showboxes', 'sides', 'skew', 'sortv', 'style', 'target', 'tooltip', 'vertices', 'width', 'xlabel', 'xlp', 'z'}

E = {'URL', 'arrowhead', 'arrowsize', 'arrowtail', 'color', 'colorscheme', 'comment', 'constraint', 'decorate', 'dir', 'edgeURL', 'edgehref', 'edgetarget', 'edgetooltip', 'fillcolor', 'fontcolor', 'fontname', 'fontsize', 'headURL', 'head_lp', 'headclip', 'headhref', 'headlabel', 'headport', 'headtarget', 'headtooltip', 'href', 'id', 'label', 'labelURL', 'labelangle', 'labeldistance', 'labelfloat', 'labelfontcolor', 'labelfontname', 'labelfontsize', 'labelhref', 'labeltarget', 'labeltooltip', 'layer', 'len', 'lhead', 'lp', 'ltail', 'minlen', 'nojustify', 'penwidth', 'pos', 'samehead', 'sametail', 'showboxes', 'style', 'tailURL', 'tail_lp', 'tailclip', 'tailhref', 'taillabel', 'tailport', 'tailtarget', 'tailtooltip', 'target', 'tooltip', 'weight', 'xlabel', 'xlp'}

G = {'Damping', 'K', 'URL', '_background', 'bb', 'bgcolor', 'center', 'charset', 'clusterrank', 'colorscheme', 'comment', 'compound', 'concentrate', 'defaultdist', 'dim', 'dimen', 'diredgeconstraints', 'dpi', 'epsilon', 'esep', 'fontcolor', 'fontname', 'fontnames', 'fontpath', 'fontsize', 'forcelabels', 'gradientangle', 'href', 'id', 'imagepath', 'inputscale', 'label', 'label_scheme', 'labeljust', 'labelloc', 'landscape', 'layerlistsep', 'layers', 'layerselect', 'layersep', 'layout', 'levels', 'levelsgap', 'lheight', 'lp', 'lwidth', 'margin', 'maxiter', 'mclimit', 'mindist', 'mode', 'model', 'mosek', 'newrank', 'nodesep', 'nojustify', 'normalize', 'notranslate', 'nslimit\nnslimit1', 'ordering', 'orientation', 'outputorder', 'overlap', 'overlap_scaling', 'overlap_shrink', 'pack', 'packmode', 'pad', 'page', 'pagedir', 'quadtree', 'quantum', 'rankdir', 'ranksep', 'ratio', 'remincross', 'repulsiveforce', 'resolution', 'root', 'rotate', 'rotation', 'scale', 'searchsize', 'sep', 'showboxes', 'size', 'smoothing', 'sortv', 'splines', 'start', 'style', 'stylesheet', 'target', 'truecolor', 'viewport', 'voro_margin', 'xdotversion'}

graphid.util.**apply_graph_layout_attrs**(*graph*, *layout_info*)

graphid.util.**bbox_from_extent**(*extent*)

Parameters
extent (*ndarray*) – tl_x, br_x, tl_y, br_y

Returns
tl_x, tl_y, w, h

Return type
bbox (ndarray)

Example

```
>>> extent = [0, 10, 0, 10]
>>> bbox = bbox_from_extent(extent)
>>> print('bbox = {}'.format(ub.urepr(list(bbox), nl=0)))
bbox = [0, 0, 10, 10]
```

graphid.util.**draw_network2**(*graph*, *layout_info*, *ax*, *as_directed=None*, *hacknoedge=False*, *hacknode=False*, *verbose=None*, ***kwargs*)

Kwargs:
use_image, arrow_width, fontsize, fontweight, fontname, fontfamilty, fontproperties

fancy way to draw networkx graphs without directly using networkx

graphid.util.**dump_nx_ondisk**(*graph*, *fpath*)

graphid.util.**ensure_nonhex_color**(*orig_color*)

graphid.util.**get_explicit_graph**(*graph*)

Parameters
graph (*nx.Graph*)

graphid.util.**get_graph_bounding_box**(*graph*)

Example

```
>>> # xdoctest: +REQUIRES(module:pygraphviz)
>>> graph = nx.path_graph([1, 2, 3, 4])
>>> nx_agraph_layout(graph, inplace=True)
>>> bbox = get_graph_bounding_box(graph)
>>> print(ub.urepr(bbox, nl=0))
[0.0, 0.0, 54.0, 252.0]
```

graphid.util.**get_nx_layout**(*graph*, *layout*, *layoutkw=None*, *verbose=None*)

graphid.util.**get_pointset_extents**(*pts*)

graphid.util.**make_agraph**(*graph_*)

graphid.util.**netx_draw_images_at_positions**(*img_list*, *pos_list*, *size_list*, *color_list*, *framewidth_list*)
Overlays images on a networkx graph

References

https://gist.github.com/shobhit/3236373 http://matplotlib.org/examples/pylab_examples/demo_annotation_box.html http://stackoverflow.com/questions/11487797/mpl-overlay-small-image http://matplotlib.org/api/text_api.html http://matplotlib.org/api/offsetbox_api.html

graphid.util.**nx_agraph_layout**(*orig_graph*, *inplace=False*, *verbose=None*, *return_agraph=False*, *groupby=None*, ***layoutkw*)

Uses graphviz and custom code to determine position attributes of nodes and edges.

Parameters

groupby (*str*) – if not None then nodes will be grouped by this attributes and groups will be layed out separately and then stacked together in a grid

References

http://www.graphviz.org/content/attrs http://www.graphviz.org/doc/info/attrs.html

CommandLine

```
python -m graphid.util.util_graphviz nx_agraph_layout --show
```

Doctest

```
>>> # xdoctest: +REQUIRES(module:pygraphviz)
>>> from graphid.util.util_graphviz import *  # NOQA
>>> import networkx as nx
>>> import itertools as it
>>> from graphid import util
>>> n, s = 9, 4
>>> offsets = list(range(0, (1 + n) * s, s))
>>> node_groups = [list(map(str, range(*o))) for o in ub.iter_window(offsets, 2)]
>>> edge_groups = [it.combinations(nodes, 2) for nodes in node_groups]
>>> graph = nx.Graph()
>>> [graph.add_nodes_from(nodes) for nodes in node_groups]
>>> [graph.add_edges_from(edges) for edges in edge_groups]
>>> for count, nodes in enumerate(node_groups):
...     nx.set_node_attributes(graph, name='id', values=ub.dzip(nodes, [count]))
>>> layoutkw = dict(prog='neato')
>>> graph1, info1 = nx_agraph_layout(graph.copy(), inplace=True, groupby='id',␣
↪**layoutkw)
>>> graph2, info2 = nx_agraph_layout(graph.copy(), inplace=True, **layoutkw)
>>> graph3, _ = nx_agraph_layout(graph1.copy(), inplace=True, **layoutkw)
>>> nx.set_node_attributes(graph1, name='pin', values='true')
>>> graph4, _ = nx_agraph_layout(graph1.copy(), inplace=True, **layoutkw)
>>> # xdoc: +REQUIRES(--show)
>>> util.show_nx(graph1, layout='custom', pnum=(2, 2, 1), fnum=1)
>>> util.show_nx(graph2, layout='custom', pnum=(2, 2, 2), fnum=1)
>>> util.show_nx(graph3, layout='custom', pnum=(2, 2, 3), fnum=1)
>>> util.show_nx(graph4, layout='custom', pnum=(2, 2, 4), fnum=1)
>>> util.show_if_requested()
>>> g1pos = nx.get_node_attributes(graph1, 'pos')['1']
>>> g4pos = nx.get_node_attributes(graph4, 'pos')['1']
>>> g2pos = nx.get_node_attributes(graph2, 'pos')['1']
>>> g3pos = nx.get_node_attributes(graph3, 'pos')['1']
>>> print('g1pos = {!r}'.format(g1pos))
>>> print('g4pos = {!r}'.format(g4pos))
>>> print('g2pos = {!r}'.format(g2pos))
>>> print('g3pos = {!r}'.format(g3pos))
>>> assert np.all(g1pos == g4pos), 'points between 1 and 4 were pinned so they␣
```

(continues on next page)

(continued from previous page)

```
↪should be equal'
>>> #assert np.all(g2pos != g3pos), 'points between 2 and 3 were not pinned, so they␣
↪should be different'
```

assert np.all(nx.get_node_attributes(graph1, ‘pos’)[‘1’] == nx.get_node_attributes(graph4, ‘pos’)[‘1’]) assert np.all(nx.get_node_attributes(graph2, ‘pos’)[‘1’] == nx.get_node_attributes(graph3, ‘pos’)[‘1’])

graphid.util.**nx_ensure_agraph_color**(*graph*)

changes colors to hex strings on graph attrs

graphid.util.**parse_aedge_layout_attrs**(*aedge*, *translation=None*)

parse grpahviz splineType

graphid.util.**parse_anode_layout_attrs**(*anode*)

graphid.util.**parse_html_graphviz_attrs**()

graphid.util.**parse_point**(*ptstr*)

graphid.util.**patch_pygraphviz**()

Hacks around a python3 problem in 1.3.1 of pygraphviz

graphid.util.**show_nx**(*graph*, *with_labels=True*, *fnum=None*, *pnum=None*, *layout='agraph'*, *ax=None*, *pos=None*, *img_dict=None*, *title=None*, *layoutkw=None*, *verbose=None*, ***kwargs*)

Parameters

- **graph** (*networkx.Graph*)
- **with_labels** (*bool*) – (default = True)
- **fnum** (*int*) – figure number(default = None)
- **pnum** (*tuple*) – plot number(default = None)
- **layout** (*str*) – (default = ‘agraph’)
- **ax** (*None*) – (default = None)
- **pos** (*None*) – (default = None)
- **img_dict** (*dict*) – (default = None)
- **title** (*str*) – (default = None)
- **layoutkw** (*None*) – (default = None)
- **verbose** (*bool*) – verbosity flag(default = None)

Kwargs:

use_image, framewidth, modify_ax, as_directed, hacknoedge, hacknode, arrow_width, fontsize, fontweight, fontname, fontfamilty, fontproperties

CommandLine

```
python -m graphid.util.util_graphviz show_nx --show
```

Example

```
>>> # xdoctest: +REQUIRES(module:pygraphviz)
>>> from graphid.util.util_graphviz import *  # NOQA
>>> graph = nx.DiGraph()
>>> graph.add_nodes_from(['a', 'b', 'c', 'd'])
>>> graph.add_edges_from({'a': 'b', 'b': 'c', 'b': 'd', 'c': 'd'}.items())
>>> nx.set_node_attributes(graph, name='shape', values='rect')
>>> nx.set_node_attributes(graph, name='image', values={'a': util.grab_test_imgpath(
↪'carl.jpg')})
>>> nx.set_node_attributes(graph, name='image', values={'d': util.grab_test_imgpath(
↪'astro.png')})
>>> #nx.set_node_attributes(graph, name='height', values=100)
>>> with_labels = True
>>> fnum = None
>>> pnum = None
>>> e = show_nx(graph, with_labels, fnum, pnum, layout='agraph')
>>> util.show_if_requested()
```

graphid.util.**stack_graphs**(*graph_list*, *vert=False*, *pad=None*)

graphid.util.**translate_graph**(*graph*, *t_xy*)

graphid.util.**translate_graph_to_origin**(*graph*)

graphid.util.**group_pairs**(*pair_list*)

Groups a list of items using the first element in each pair as the item and the second element as the groupid.

Parameters

pair_list (*list*) – list of 2-tuples (item, groupid)

Returns

groupid_to_items: maps a groupid to a list of items

Return type

dict

graphid.util.**grouping_delta**(*old*, *new*, *pure=True*)

Finds what happened to the old groups to form the new groups.

Parameters

- **old** (*set of frozensets*) – old grouping
- **new** (*set of frozensets*) – new grouping
- **pure** (*bool*) – hybrids are separated from pure merges and splits if pure is True, otherwise hybrid cases are grouped in merges and splits.

Returns

delta: dictionary of changes containing the merges, splits,
unchanged, and hybrid cases. Except for unchanged, case a subdict with new and old

keys. For splits / merges, one of these contains nested sequences to indicate what the split / merge is. Also reports elements added and removed between old and new if the flattened sets are not the same.

Return type
dict

Notes

merges - which old groups were merged into a single new group. splits - which old groups were split into multiple new groups. hybrid - which old groups had split/merge actions applied. unchanged - which old groups are the same as new groups.

Example

```
>>> # xdoc: +IGNORE_WHITESPACE
>>> old = [
>>>     [20, 21, 22, 23], [1, 2], [12], [13, 14], [3, 4], [5, 6,11],
>>>     [7], [8, 9], [10], [31, 32], [33, 34, 35], [41, 42, 43, 44, 45]
>>> ]
>>> new = [
>>>   [20, 21], [22, 23], [1, 2], [12, 13, 14], [4], [5, 6, 3], [7, 8],
>>>   [9, 10, 11], [31, 32, 33, 34, 35],   [41, 42, 43, 44], [45],
>>> ]
>>> delta = grouping_delta(old, new)
>>> assert set(old[0]) in delta['splits']['old']
>>> assert set(new[3]) in delta['merges']['new']
>>> assert set(old[1]) in delta['unchanged']
>>> result = ub.urepr(delta, nl=2, sort=True, nobr=1, sk=True)
>>> print(result)
hybrid: {
    merges: [{{10}, {11}, {9}}, {{3}, {5, 6}}, {{4}}, {{7}, {8}}],
    new: {{3, 5, 6}, {4}, {7, 8}, {9, 10, 11}},
    old: {{10}, {3, 4}, {5, 6, 11}, {7}, {8, 9}},
    splits: [{{10}}, {{11}, {5, 6}}, {{3}, {4}}, {{7}}, {{8}, {9}}],
},
items: {
    added: {},
    removed: {},
},
merges: {
    new: [{12, 13, 14}, {31, 32, 33, 34, 35}],
    old: [{{12}, {13, 14}}, {{31, 32}, {33, 34, 35}}],
},
splits: {
    new: [{{20, 21}, {22, 23}}, {{41, 42, 43, 44}, {45}}],
    old: [{20, 21, 22, 23}, {41, 42, 43, 44, 45}],
},
unchanged: {
    {1, 2},
},
```

Example

```
>>> old = [
>>>     [1, 2, 3], [4], [5, 6, 7, 8, 9], [10, 11, 12]
>>> ]
>>> new = [
>>>     [1], [2], [3, 4], [5, 6, 7], [8, 9, 10, 11, 12]
>>> ]
>>> # every case here is hybrid
>>> pure_delta = grouping_delta(old, new, pure=True)
>>> assert len(list(ub.flatten(pure_delta['merges'].values()))) == 0
>>> assert len(list(ub.flatten(pure_delta['splits'].values()))) == 0
>>> delta = grouping_delta(old, new, pure=False)
>>> delta = order_dict_by(delta, ['unchanged', 'splits', 'merges'])
>>> result = ub.urepr(delta, nl=2, sort=True, sk=True)
>>> print(result)
{
    items: {
        added: {},
        removed: {},
    },
    merges: [
        [{3}, {4}],
        [{10, 11, 12}, {8, 9}],
    ],
    splits: [
        [{1}, {2}, {3}],
        [{5, 6, 7}, {8, 9}],
    ],
    unchanged: {},
}
```

Example

```
>>> delta = grouping_delta([[1, 2, 3]], [])
>>> assert len(delta['items']['removed']) == 3
>>> delta = grouping_delta([], [[1, 2, 3]])
>>> assert len(delta['items']['added']) == 3
>>> delta = grouping_delta([[1]], [[1, 2, 3]])
>>> assert len(delta['items']['added']) == 2
>>> assert len(delta['unchanged']) == 1
```

graphid.util.**order_dict_by**(*dict_*, *key_order*)

Reorders items in a dictionary according to a custom key order

Parameters

- **dict_** (*dict_*) – a dictionary
- **key_order** (*list*) – custom key order

Returns

sorted_dict

Return type
OrderedDict

Example

```
>>> dict_ = {1: 1, 2: 2, 3: 3, 4: 4}
>>> key_order = [4, 2, 3, 1]
>>> sorted_dict = order_dict_by(dict_, key_order)
>>> result = ('sorted_dict = %s' % (ub.urepr(sorted_dict, nl=False),))
>>> print(result)
>>> assert result == 'sorted_dict = {4: 4, 2: 2, 3: 3, 1: 1}'
```

graphid.util.**sort_dict**(*dict_*, *part='keys'*, *key=None*, *reverse=False*)

sorts a dictionary by its values or its keys

Parameters

- **dict_** (*dict_*) – a dictionary
- **part** (*str*) – specifies to sort by keys or values
- **key** (*Optional[func]*) – a function that takes specified part and returns a sortable value
- **reverse** (*bool*) – (Defaults to False) - True for descinding order. False for ascending order.

Returns
sorted dictionary

Return type
OrderedDict

Example

```
>>> dict_ = {'a': 3, 'c': 2, 'b': 1}
>>> results = []
>>> results.append(sort_dict(dict_, 'keys'))
>>> results.append(sort_dict(dict_, 'vals'))
>>> results.append(sort_dict(dict_, 'vals', lambda x: -x))
>>> result = ub.urepr(results)
>>> print(result)
[
    {'a': 3, 'b': 1, 'c': 2},
    {'b': 1, 'c': 2, 'a': 3},
    {'a': 3, 'c': 2, 'b': 1},
]
```

graphid.util.**sortedby**(*item_list*, *key_list*, *reverse=False*)

sorts `item_list` using key_list

Parameters

- **list_** (*list*) – list to sort
- **key_list** (*list*) – list to sort by
- **reverse** (*bool*) – sort order is descending (largest first) if reverse is True else acscending (smallest first)

Returns
list_ sorted by the values of another list. defaults to ascending order

Return type
list

SeeAlso:
sortedby2

Examples

```
>>> list_    = [1, 2, 3, 4, 5]
>>> key_list = [2, 5, 3, 1, 5]
>>> result = sortedby(list_, key_list, reverse=True)
>>> print(result)
[5, 2, 3, 1, 4]
```

graphid.util.**convert_colorspace**(*img*, *dst_space*, *src_space='BGR'*, *copy=False*, *dst=None*)

Converts colorspace of img. Convinience function around cv2.cvtColor

Parameters

- **img** (*ndarray[uint8_t, ndim=2]*) – image data
- **colorspace** (*str*) – RGB, LAB, etc
- **dst_space** (*unicode*) – (default = u'BGR')

Returns
img - image data

Return type
ndarray[uint8_t, ndim=2]

Example

```
>>> convert_colorspace(np.array([[[0, 0, 1]]], dtype=np.float32), 'LAB', src_space=
↪'RGB')
>>> convert_colorspace(np.array([[[0, 1, 0]]], dtype=np.float32), 'LAB', src_space=
↪'RGB')
>>> convert_colorspace(np.array([[[1, 0, 0]]], dtype=np.float32), 'LAB', src_space=
↪'RGB')
>>> convert_colorspace(np.array([[[1, 1, 1]]], dtype=np.float32), 'LAB', src_space=
↪'RGB')
>>> convert_colorspace(np.array([[[0, 0, 1]]], dtype=np.float32), 'HSV', src_space=
↪'RGB')
```

graphid.util.**ensure_float01**(*img*, *dtype=<class 'numpy.float32'>*, *copy=True*)

Ensure that an image is encoded using a float properly

graphid.util.**get_num_channels**(*img*)

Returns the number of color channels

graphid.util.**imread**(*fpath*, ***kw*)

reads image data in BGR format

Example

```
>>> # xdoctest: +SKIP("use kwimage.imread")
>>> import ubelt as ub
>>> import tempfile
>>> from os.path import splitext  # NOQA
>>> fpath = ub.grabdata('https://i.imgur.com/oHGsmvF.png', fname='carl.png')
>>> #fpath = ub.grabdata('http://www.topcoder.com/contest/problem/UrbanMapper3D/JAX_
↪Tile_043_DTM.tif')
>>> ext = splitext(fpath)[1]
>>> img1 = imread(fpath)
>>> # Check that write + read preserves data
>>> tmp = tempfile.NamedTemporaryFile(suffix=ext)
>>> imwrite(tmp.name, img1)
>>> img2 = imread(tmp.name)
>>> assert np.all(img2 == img1)
```

Example

```
>>> # xdoctest: +SKIP("use kwimage.imread")
>>> import tempfile
>>> import ubelt as ub
>>> #img1 = (np.arange(0, 12 * 12 * 3).reshape(12, 12, 3) % 255).astype(np.uint8)
>>> img1 = imread(ub.grabdata('http://i.imgur.com/iXNf4Me.png', fname='ada.png'))
>>> tmp_tif = tempfile.NamedTemporaryFile(suffix='.tif')
>>> tmp_png = tempfile.NamedTemporaryFile(suffix='.png')
>>> imwrite(tmp_tif.name, img1)
>>> imwrite(tmp_png.name, img1)
>>> tif_im = imread(tmp_tif.name)
>>> png_im = imread(tmp_png.name)
>>> assert np.all(tif_im == png_im)
```

Example

```
>>> # xdoctest: +SKIP("use kwimage.imread")
>>> from graphid.util.util_image import *
>>> import tempfile
>>> import ubelt as ub
>>> #img1 = (np.arange(0, 12 * 12 * 3).reshape(12, 12, 3) % 255).astype(np.uint8)
>>> tif_fpath = ub.grabdata('https://ghostscript.com/doc/tiff/test/images/rgb-3c-
↪16b.tiff')
>>> img1 = imread(tif_fpath)
>>> tmp_tif = tempfile.NamedTemporaryFile(suffix='.tif')
>>> tmp_png = tempfile.NamedTemporaryFile(suffix='.png')
>>> imwrite(tmp_tif.name, img1)
>>> imwrite(tmp_png.name, img1)
>>> tif_im = imread(tmp_tif.name)
>>> png_im = imread(tmp_png.name)
>>> assert np.all(tif_im == png_im)
```

graphid.util.**imwrite**(*fpath*, *image*, ***kw*)

writes image data in BGR format

class graphid.util.**KWSpec**(*spec*)

Bases: object

Safer keyword arguments with keyword specifications.

graphid.util.**all_dict_combinations**(*varied_dict*)

Parameters
varied_dict (*dict*) – a dict with lists of possible parameter settings

Returns
dict_list a list of dicts correpsonding to all combinations of params settings

Return type
list

Example

```
>>> varied_dict = {'logdist_weight': [0.0, 1.0], 'pipeline_root': ['vsmany'], 'sv_on
↪': [True, False, None]}
>>> dict_list = all_dict_combinations(varied_dict)
>>> result = str(ub.urepr(dict_list))
>>> print(result)
[
    {'logdist_weight': 0.0, 'pipeline_root': 'vsmany', 'sv_on': True},
    {'logdist_weight': 0.0, 'pipeline_root': 'vsmany', 'sv_on': False},
    {'logdist_weight': 0.0, 'pipeline_root': 'vsmany', 'sv_on': None},
    {'logdist_weight': 1.0, 'pipeline_root': 'vsmany', 'sv_on': True},
    {'logdist_weight': 1.0, 'pipeline_root': 'vsmany', 'sv_on': False},
    {'logdist_weight': 1.0, 'pipeline_root': 'vsmany', 'sv_on': None},
]
```

graphid.util.**aslist**(*sequence*)

Ensures that the sequence object is a Python list. Handles, numpy arrays, and python sequences (e.g. tuples, and iterables).

Parameters
sequence (*sequence*) – a list-like object

Returns
list_ - *sequence* as a Python list

Return type
list

Example

```
>>> s1 = [1, 2, 3]
>>> s2 = (1, 2, 3)
>>> assert aslist(s1) is s1
>>> assert aslist(s2) is not s2
>>> aslist(np.array([[1, 2], [3, 4], [5, 6]]))
[[1, 2], [3, 4], [5, 6]]
>>> aslist(range(3))
[0, 1, 2]
```

class graphid.util.**classproperty**(*fget=None*, *fset=None*, *fdel=None*, *doc=None*)

Bases: property

Decorates a method turning it into a classattribute

References

https://stackoverflow.com/questions/1697501/python-staticmethod-with-property

graphid.util.**cprint**(*text*, *color=None*)

provides some color to terminal output

Parameters

- **text** (*str*)
- **color** (*str*)

Example0:

```
>>> import pygments.console
>>> msg_list = list(pygments.console.codes.keys())
>>> color_list = list(pygments.console.codes.keys())
>>> [cprint(text, color) for text, color in zip(msg_list, color_list)]
```

Example1:

```
>>> import pygments.console
>>> print('line1')
>>> cprint('line2', 'red')
>>> cprint('line3', 'blue')
>>> cprint('line4', 'magenta')
>>> cprint('line5', 'reset')
>>> cprint('line5', 'magenta')
>>> print('line6')
```

graphid.util.**delete_dict_keys**(*dict_*, *key_list*)

Removes items from a dictionary inplace. Keys that do not exist are ignored.

Parameters

- **dict_** (*dict*) – dict like object with a __del__ attribute
- **key_list** (*list*) – list of keys that specify the items to remove

Example

```
>>> dict_ = {'bread': 1, 'churches': 1, 'cider': 2, 'very small rocks': 2}
>>> key_list = ['duck', 'bread', 'cider']
>>> delete_dict_keys(dict_, key_list)
>>> result = ub.urepr(dict_, nl=False)
>>> print(result)
{'churches': 1, 'very small rocks': 2}
```

graphid.util.**delete_items_by_index**(*list_*, *index_list*, *copy=False*)

Remove items from `list_` at positions specified in `index_list` The original `list_` is preserved if `copy` is True

Parameters

- **list_** (*list*)
- **index_list** (*list*)
- **copy** (*bool*) – preserves original list if True

Example

```
>>> list_ = [8, 1, 8, 1, 6, 6, 3, 4, 4, 5, 6]
>>> index_list = [2, -1]
>>> result = delete_items_by_index(list_, index_list)
>>> print(result)
[8, 1, 1, 6, 6, 3, 4, 4, 5]
```

graphid.util.**ensure_iterable**(*obj*)

Parameters
obj (*scalar or iterable*)

Returns
obj if it was iterable otherwise [obj]

Return type
it3erable

Timeit:
%timeit util.ensure_iterable([1]) %timeit util.ensure_iterable(1) %timeit util.ensure_iterable(np.array(1)) %timeit util.ensure_iterable([1]) %timeit [1]

Example

```
>>> obj_list = [3, [3], '3', (3,), [3,4,5]]
>>> result = [ensure_iterable(obj) for obj in obj_list]
>>> result = str(result)
>>> print(result)
[[3], [3], ['3'], (3,), [3, 4, 5]]
```

graphid.util.**estarmap**(*func*, *iter_*, ***kwargs*)

Eager version of it.starmap from itertools

Note this is inefficient and should only be used when prototyping and debugging.

graphid.util.**flag_None_items**(*list_*)

graphid.util.**get_timestamp**(*format_='iso'*, *use_second=False*, *delta_seconds=None*, *isutc=False*, *timezone=False*)

Parameters

- **format_** (*str*) – (tag, printable, filename, other)
- **use_second** (*bool*)
- **delta_seconds** (*None*)

Returns
stamp

Return type
str

Example

```
>>> format_ = 'printable'
>>> use_second = False
>>> delta_seconds = None
>>> stamp = get_timestamp(format_, use_second, delta_seconds)
>>> print(stamp)
>>> assert len(stamp) == len('15:43:04 2015/02/24')
```

graphid.util.**highlight_regex**(*str_*, *pat*, *reflags=0*, *color='red'*)

FIXME Use pygments instead

graphid.util.**isect**(*list1*, *list2*)

returns list1 elements that are also in list2. preserves order of list1

intersect_ordered

Parameters

- **list1** (*list*)
- **list2** (*list*)

Returns
new_list

Return type
list

Example

```
>>> list1 = ['featweight_rowid', 'feature_rowid', 'config_rowid', 'featweight_
↪forground_weight']
>>> list2 = [u'featweight_rowid']
>>> result = isect(list1, list2)
>>> print(result)
['featweight_rowid']
```

graphid.util.**iteritems_sorted**(*dict_*)

change to iteritems ordered

graphid.util.**make_index_lookup**(*list_*, *dict_factory=<class 'dict'>*)

Parameters
list_ (*list*) – assumed to have unique items

Returns
mapping from item to index

Return type
dict

Example

```
>>> list_ = [5, 3, 8, 2]
>>> idx2_item = make_index_lookup(list_)
>>> result = ub.urepr(idx2_item, nl=False, sort=1)
>>> assert list(ub.take(idx2_item, list_)) == list(range(len(list_)))
>>> print(result)
{2: 3, 3: 1, 5: 0, 8: 2}
```

graphid.util.**partial_order**(*list_*, *part*)

graphid.util.**randn**(*mean=0*, *std=1*, *shape=[]*, *a_max=None*, *a_min=None*, *rng=None*)

graphid.util.**regex_word**(*w*)

graphid.util.**replace_nones**(*list_*, *repl=-1*)

Recursively removes Nones in all lists and sublists and replaces them with the repl variable

Parameters

- **list_** (*list*)
- **repl** (*obj*) – replacement value

Returns
list

Example

```
>>> list_ = [None, 0, 1, 2]
>>> repl = -1
>>> repl_list = replace_nones(list_, repl)
>>> result = str(repl_list)
>>> print(result)
[-1, 0, 1, 2]
```

graphid.util.**safe_argmax**(*arr*, *fill=nan*, *finite=False*, *nans=True*)

Doctest

```
>>> assert safe_argmax([np.nan, np.nan], nans=False) == 0
>>> assert safe_argmax([-100, np.nan], nans=False) == 0
>>> assert safe_argmax([np.nan, -100], nans=False) == 1
>>> assert safe_argmax([-100, 0], nans=False) == 1
>>> assert np.isnan(safe_argmax([]))
```

graphid.util.**safe_extreme**(*arr*, *op*, *fill=nan*, *finite=False*, *nans=True*)

Applies an exterme operation to an 1d array (typically max/min) but ensures a value is always returned even in operations without identities. The default identity must be specified using the *fill* argument.

Parameters

- **arr** (*ndarray*) – 1d array to take extreme of
- **op** (*func*) – vectorized operation like np.max to apply to array
- **fill** (*float*) – return type if arr has no elements (default = nan)
- **finite** (*bool*) – if True ignores non-finite values (default = False)
- **nans** (*bool*) – if False ignores nans (default = True)

graphid.util.**safe_max**(*arr*, *fill=nan*, *finite=False*, *nans=True*)

Parameters

- **arr** (*ndarray*) – 1d array to take max of
- **fill** (*float*) – return type if arr has no elements (default = nan)
- **finite** (*bool*) – if True ignores non-finite values (default = False)
- **nans** (*bool*) – if False ignores nans (default = True)

Example

```
>>> arrs = [[], [np.nan], [-np.inf, np.nan, np.inf], [np.inf], [np.inf, 1], [0, 1]]
>>> arrs = [np.array(arr) for arr in arrs]
>>> fill = np.nan
>>> results1 = [safe_max(arr, fill, finite=False, nans=True) for arr in arrs]
>>> results2 = [safe_max(arr, fill, finite=True, nans=True) for arr in arrs]
>>> results3 = [safe_max(arr, fill, finite=True, nans=False) for arr in arrs]
>>> results4 = [safe_max(arr, fill, finite=False, nans=False) for arr in arrs]
>>> results = [results1, results2, results3, results4]
>>> result = ('results = %s' % (ub.urepr(results, nl=1, sv=1),))
>>> print(result)
results = [
    [nan, nan, nan, inf, inf, 1],
    [nan, nan, nan, nan, 1.0, 1],
    [nan, nan, nan, nan, 1.0, 1],
    [nan, nan, inf, inf, inf, 1],
]
```

graphid.util.**safe_min**(*arr*, *fill=nan*, *finite=False*, *nans=True*)

Example

```
>>> arrs = [[], [np.nan], [-np.inf, np.nan, np.inf], [np.inf], [np.inf, 1], [0, 1]]
>>> arrs = [np.array(arr) for arr in arrs]
>>> fill = np.nan
>>> results1 = [safe_min(arr, fill, finite=False, nans=True) for arr in arrs]
>>> results2 = [safe_min(arr, fill, finite=True, nans=True) for arr in arrs]
>>> results3 = [safe_min(arr, fill, finite=True, nans=False) for arr in arrs]
>>> results4 = [safe_min(arr, fill, finite=False, nans=False) for arr in arrs]
>>> results = [results1, results2, results3, results4]
>>> result = ('results = %s' % (ub.urepr(results, nl=1, sv=1),))
>>> print(result)
results = [
    [nan, nan, nan, inf, 1.0, 0],
    [nan, nan, nan, nan, 1.0, 0],
    [nan, nan, nan, nan, 1.0, 0],
    [nan, nan, -inf, inf, 1.0, 0],
]
```

graphid.util.**setdiff**(*list1*, *list2*)

returns list1 elements that are not in list2. preserves order of list1

Parameters

- **list1** (*list*)
- **list2** (*list*)

Returns

new_list

Return type

list

Example

```
>>> list1 = ['featweight_rowid', 'feature_rowid', 'config_rowid', 'featweight_
↪forground_weight']
>>> list2 = [u'featweight_rowid']
>>> new_list = setdiff(list1, list2)
>>> result = ub.urepr(new_list, nl=False)
>>> print(result)
['feature_rowid', 'config_rowid', 'featweight_forground_weight']
```

graphid.util.**snapped_slice**(*size*, *frac*, *n*)

Creates a slice spanning *n* items in a list of length *size* at position *frac*.

Parameters

- **size** (*int*) – length of the list
- **frac** (*float*) – position in the range [0, 1]
- **n** (*int*) – number of items in the slice

Returns

slice object that best fits the criteria

Return type
slice

SeeAlso:
take_percentile_parts

Example:

Example

```
>>> # DISABLE_DOCTEST
>>> print(snapped_slice(0, 0, 10))
>>> print(snapped_slice(1, 0, 10))
>>> print(snapped_slice(100, 0, 10))
>>> print(snapped_slice(9, 0, 10))
>>> print(snapped_slice(100, 1, 10))
pass
```

graphid.util.**stats_dict**(*list_*, *axis=None*, *use_nan=False*, *use_sum=False*, *use_median=False*, *size=False*)

Parameters

- **list_** (*listlike*) – values to get statistics of
- **axis** (*int*) – if *list_* is ndarray then this specifies the axis

Returns

stats: dictionary of common numpy statistics
(min, max, mean, std, nMin, nMax, shape)

Return type
OrderedDict

Examples0:

```
>>> # xdoctest: +IGNORE_WHITESPACE
>>> import numpy as np
>>> axis = 0
>>> np.random.seed(0)
>>> list_ = np.random.rand(10, 2).astype(np.float32)
>>> stats = stats_dict(list_, axis, use_nan=False)
>>> result = str(ub.urepr(stats, nl=1, precision=4, with_dtype=True))
>>> print(result)
{
    'mean': np.array([0.5206, 0.6425], dtype=np.float32),
    'std': np.array([0.2854, 0.2517], dtype=np.float32),
    'max': np.array([0.9637, 0.9256], dtype=np.float32),
    'min': np.array([0.0202, 0.0871], dtype=np.float32),
    'nMin': np.array([1, 1], dtype=np.int32),
    'nMax': np.array([1, 1], dtype=np.int32),
    'shape': (10, 2),
}
```

Examples1:

```
>>> import numpy as np
>>> axis = 0
>>> rng = np.random.RandomState(0)
>>> list_ = rng.randint(0, 42, size=100).astype(np.float32)
>>> list_[4] = np.nan
>>> stats = stats_dict(list_, axis, use_nan=True)
>>> result = str(ub.urepr(stats, precision=1, sk=True))
>>> print(result)
{mean: 20.0, std: 13.2, max: 41.0, min: 0.0, nMin: 7, nMax: 3, shape: (100,),␣
↪num_nan: 1,}
```

graphid.util.**take_percentile_parts**(*arr*, *front=None*, *mid=None*, *back=None*)

Take parts from front, back, or middle of a list

Example

```
>>> arr = list(range(20))
>>> front = 3
>>> mid = 3
>>> back = 3
>>> result = take_percentile_parts(arr, front, mid, back)
>>> print(result)
[0, 1, 2, 9, 10, 11, 17, 18, 19]
```

graphid.util.**where**(*flag_list*)

takes flags returns indexes of True values

graphid.util.**apply_grouping**(*items*, *groupxs*, *axis=0*)

applies grouping from group_indicies apply_grouping

Parameters

- **items** (*ndarray*)
- **groupxs** (*list of ndarrays*)

Returns

grouped items

Return type

list of ndarrays

SeeAlso:

group_indices invert_apply_grouping

Example

```
>>> # xdoctest: +IGNORE_WHITESPACE
>>> idx2_groupid = np.array([2, 1, 2, 1, 2, 1, 2, 3, 3, 3, 3])
>>> items        = np.array([1, 8, 5, 5, 8, 6, 7, 5, 3, 0, 9])
>>> (keys, groupxs) = group_indices(idx2_groupid)
>>> grouped_items = apply_grouping(items, groupxs)
>>> result = str(grouped_items)
>>> print(result)
[array([8, 5, 6]), array([1, 5, 8, 7]), array([5, 3, 0, 9])]
```

graphid.util.**atleast_nd**(*arr*, *n*, *front=False*)

View inputs as arrays with at least n dimensions. TODO: Submit as a PR to numpy

Parameters

- **arr** (*array_like*) – One array-like object. Non-array inputs are converted to arrays. Arrays that already have n or more dimensions are preserved.
- **n** (*int*) – number of dimensions to ensure
- **tofront** (*bool*) – if True new dimensions are added to the front of the array. otherwise they are added to the back.

Returns

An array with `a.ndim >= n`. Copies are avoided where possible, and views with three or more dimensions are returned. For example, a 1-D array of shape `(N,)` becomes a view of shape `(1, N, 1)`, and a 2-D array of shape `(M, N)` becomes a view of shape `(M, N, 1)`.

Return type

ndarray

Example

```
>>> n = 2
>>> arr = np.array([1, 1, 1])
>>> arr_ = atleast_nd(arr, n)
>>> result = ub.urepr(arr_.tolist(), nl=0)
>>> print(result)
[[1], [1], [1]]
```

Example

```
>>> n = 4
>>> arr1 = [1, 1, 1]
>>> arr2 = np.array(0)
>>> arr3 = np.array([[[[[1]]]]])
>>> arr1_ = atleast_nd(arr1, n)
>>> arr2_ = atleast_nd(arr2, n)
>>> arr3_ = atleast_nd(arr3, n)
>>> result1 = ub.urepr(arr1_.tolist(), nl=0)
>>> result2 = ub.urepr(arr2_.tolist(), nl=0)
>>> result3 = ub.urepr(arr3_.tolist(), nl=0)
```

(continues on next page)

(continued from previous page)

```
>>> result = '\n'.join([result1, result2, result3])
>>> print(result)
[[[[1]]], [[[1]]], [[[1]]]]
[[[[0]]]]
[[[[[1]]]]]
```

Benchmark

import ubelt N = 100

t1 = ubelt.Timerit(N, label='mine') for timer in t1:

arr = np.empty((10, 10)) with timer:

atleast_nd(arr, 3)

t2 = ubelt.Timerit(N, label='baseline') for timer in t2:

arr = np.empty((10, 10)) with timer:

np.atleast_3d(arr)

graphid.util.**group_indices**(*idx2_groupid*, *assume_sorted=False*)

Parameters
idx2_groupid (*ndarray*) – numpy array of group ids (must be numeric)

Returns
(keys, groupxs)

Return type
tuple (ndarray, list of ndarrays)

Example0:

```
>>> # xdoctest: +IGNORE_WHITESPACE
>>> idx2_groupid = np.array([2, 1, 2, 1, 2, 1, 2, 3, 3, 3, 3])
>>> (keys, groupxs) = group_indices(idx2_groupid)
>>> result = ub.urepr((keys, groupxs), nobr=True, with_dtype=True)
>>> print(result)
```

np.array([1, 2, 3], dtype=np.int64), [

np.array([1, 3, 5], dtype=np.int64), np.array([0, 2, 4, 6], dtype=np.int64), np.array([7, 8, 9, 10], dtype=np.int64)...

Example1:

```
>>> # xdoctest: +IGNORE_WHITESPACE
>>> idx2_groupid = np.array([[  24], [ 129], [ 659], [ 659], [ 24],
...       [659], [ 659], [ 822], [ 659], [ 659], [24]])
>>> # 2d arrays must be flattened before coming into this function so
>>> # information is on the last axis
>>> (keys, groupxs) = group_indices(idx2_groupid.T[0])
>>> result = ub.urepr((keys, groupxs), nobr=True, with_dtype=True)
>>> print(result)
```

np.array([24, 129, 659, 822], dtype=np.int64), [

np.array([0, 4, 10], dtype=np.int64), np.array([1], dtype=np.int64), np.array([2, 3, 5, 6, 8, 9], dtype=np.int64), np.array([7], dtype=np.int64)...

Example2:

```
>>> # xdoctest: +IGNORE_WHITESPACE
>>> idx2_groupid = np.array([True, True, False, True, False, False, True])
>>> (keys, groupxs) = group_indices(idx2_groupid)
>>> result = ub.urepr((keys, groupxs), nobr=True, with_dtype=True)
>>> print(result)
```

np.array([False, True], dtype=bool), [

np.array([2, 4, 5], dtype=np.int64), np.array([0, 1, 3, 6], dtype=np.int64)...

Timeit:

import numba group_indices_numba = numba.jit(group_indices) group_indices_numba(idx2_groupid)

SeeAlso:

apply_grouping

References

http://stackoverflow.com/questions/4651683/ numpy-grouping-using-itertools-groupby-performance

---

Todo: Look into np.split http://stackoverflow.com/questions/21888406/ getting-the-indexes-to-the-duplicate-columns-of-a-numpy-array

---

graphid.util.**group_items**(*item_list*, *groupid_list*, *assume_sorted=False*, *axis=None*)

graphid.util.**isect_flags**(*arr*, *other*)

Example

```
>>> arr = np.array([
>>>     [1, 2, 3, 4],
>>>     [5, 6, 3, 4],
>>>     [1, 1, 3, 4],
>>> ])
>>> other = np.array([1, 4, 6])
>>> mask = isect_flags(arr, other)
>>> print(mask)
[[ True False False  True]
 [False  True False  True]
 [ True  True False  True]]
```

graphid.util.**iter_reduce_ufunc**(*ufunc*, *arr_iter*, *out=None*)

constant memory iteration and reduction

applys ufunc from left to right over the input arrays

Example

```
>>> arr_list = [
...     np.array([0, 1, 2, 3, 8, 9]),
...     np.array([4, 1, 2, 3, 4, 5]),
...     np.array([0, 5, 2, 3, 4, 5]),
...     np.array([1, 1, 6, 3, 4, 5]),
...     np.array([0, 1, 2, 7, 4, 5])
... ]
>>> memory = np.array([9, 9, 9, 9, 9, 9])
>>> gen_memory = memory.copy()
>>> def arr_gen(arr_list, gen_memory):
...     for arr in arr_list:
...         gen_memory[:] = arr
...         yield gen_memory
>>> print('memory = %r' % (memory,))
>>> print('gen_memory = %r' % (gen_memory,))
>>> ufunc = np.maximum
>>> res1 = iter_reduce_ufunc(ufunc, iter(arr_list), out=None)
>>> res2 = iter_reduce_ufunc(ufunc, iter(arr_list), out=memory)
>>> res3 = iter_reduce_ufunc(ufunc, arr_gen(arr_list, gen_memory), out=memory)
>>> print('res1       = %r' % (res1,))
>>> print('res2       = %r' % (res2,))
>>> print('res3       = %r' % (res3,))
>>> print('memory     = %r' % (memory,))
>>> print('gen_memory = %r' % (gen_memory,))
>>> assert np.all(res1 == res2)
>>> assert np.all(res2 == res3)
```

graphid.util.**ensure_rng**(*rng*, *api='numpy'*)

Returns a random number generator

Parameters

seed – if None, then the rng is unseeded. Otherwise the seed can be an integer or a RandomState class

Example

```
>>> rng = ensure_rng(None)
>>> ensure_rng(0).randint(0, 1000)
684
>>> ensure_rng(np.random.RandomState(1)).randint(0, 1000)
37
```

Example

```
>>> num = 4
>>> print('--- Python as PYTHON ---')
>>> py_rng = random.Random(0)
>>> pp_nums = [py_rng.random() for _ in range(num)]
>>> print(pp_nums)
>>> print('--- Numpy as PYTHON ---')
>>> np_rng = ensure_rng(random.Random(0), api='numpy')
>>> np_nums = [np_rng.rand() for _ in range(num)]
>>> print(np_nums)
>>> print('--- Numpy as NUMPY---')
>>> np_rng = np.random.RandomState(seed=0)
>>> nn_nums = [np_rng.rand() for _ in range(num)]
>>> print(nn_nums)
>>> print('--- Python as NUMPY---')
>>> py_rng = ensure_rng(np.random.RandomState(seed=0), api='python')
>>> pn_nums = [py_rng.random() for _ in range(num)]
>>> print(pn_nums)
>>> assert np_nums == pp_nums
>>> assert pn_nums == nn_nums
```

graphid.util.**random_combinations**(*items*, *size*, *num=None*, *rng=None*)

Yields *num* combinations of length *size* from items in random order

Parameters

- **items** (*List*) – pool of items to choose from
- **size** (*int*) – number of items in each combination
- **num** (*None, default=None*) – number of combinations to generate
- **rng** (*int | RandomState, default=None*) – seed or random number generator

Yields

tuple – combo

Example

```
>>> import ubelt as ub  # NOQA
>>> items = list(range(10))
>>> size = 3
>>> num = 5
>>> rng = 0
>>> combos = list(random_combinations(items, size, num, rng))
>>> result = ('combos = %s' % (ub.urepr(combos),))
>>> print(result)
```

Example

```
>>> import ubelt as ub  # NOQA
>>> items = list(zip(range(10), range(10)))
>>> size = 3
>>> num = 5
>>> rng = 0
>>> combos = list(random_combinations(items, size, num, rng))
>>> result = ('combos = %s' % (ub.urepr(combos),))
>>> print(result)
```

graphid.util.**random_product**(*items*, *num=None*, *rng=None*)

Yields *num* items from the cartesian product of items in a random order.

Parameters

items (*list of sequences*) – items to get caresian product of packed in a list or tuple. (note this deviates from api of it.product)

Example

```
>>> items = [(1, 2, 3), (4, 5, 6, 7)]
>>> rng = 0
>>> list(random_product(items, rng=0))
>>> list(random_product(items, num=3, rng=0))
```

graphid.util.**shuffle**(*items*, *rng=None*)

Shuffles a list inplace and then returns it for convinience

Parameters

- **items** (*list or ndarray*) – list to shuffl
- **rng** (*RandomState or int*) – seed or random number gen

Returns

this is the input, but returned for convinience

Return type

list

Example

```
>>> list1 = [1, 2, 3, 4, 5, 6]
>>> list2 = shuffle(list(list1), rng=1)
>>> assert list1 != list2
>>> result = str(list2)
>>> print(result)
[3, 2, 5, 1, 4, 6]
```

graphid.util.**alias_tags**(*tags_list*, *alias_map*)

update tags to new values

Parameters

- **tags_list** (*list*)

- **alias_map** (*list*) – list of 2-tuples with regex, value

Returns
updated tags

Return type
list

graphid.util.**build_alias_map**(*regex_map*, *tag_vocab*)

Constructs explicit mapping. Order of items in regex map matters. Items at top are given preference.

graphid.util.**filterflags_general_tags**(*tags_list*, *has_any=None*, *has_all=None*, *has_none=None*, *min_num=None*, *max_num=None*, *any_startswith=None*, *any_endswith=None*, *in_any=None*, *any_match=None*, *none_match=None*, *logic='and'*, *ignore_case=True*)

Parameters

- **tags_list** (*list*)
- **has_any** (*None*) – (default = None)
- **has_all** (*None*) – (default = None)
- **min_num** (*None*) – (default = None)
- **max_num** (*None*) – (default = None)

Notes

in_any should probably be ni_any

Example1:

```
>>> # ENABLE_DOCTEST
>>> tags_list = [['v'], [], ['P'], ['P'], ['n', 'o'], [], ['n', 'N'], ['e', 'i
↪', 'p', 'b', 'n'], ['n'], ['n'], ['N']]
>>> has_all = 'n'
>>> min_num = 1
>>> flags = filterflags_general_tags(tags_list, has_all=has_all, min_num=min_
↪num)
>>> result = list(ub.compress(tags_list, flags))
>>> print('result = %r' % (result,))
```

Example2:

```
>>> tags_list = [['vn'], ['vn', 'no'], ['P'], ['P'], ['n', 'o'], [], ['n', 'N
↪'], ['e', 'i', 'p', 'b', 'n'], ['n'], ['n', 'nP'], ['NP']]
>>> kwargs = {
>>>     'any_endswith': 'n',
>>>     'any_match': None,
>>>     'any_startswith': 'n',
>>>     'has_all': None,
>>>     'has_any': None,
>>>     'has_none': None,
>>>     'max_num': 3,
>>>     'min_num': 1,
>>>     'none_match': ['P'],
```

(continues on next page)

(continued from previous page)

```
>>> }
>>> flags = filterflags_general_tags(tags_list, **kwargs)
>>> filtered = list(ub.compress(tags_list, flags))
>>> result = ('result = %s' % (ub.urepr(filtered, nl=0),))
>>> print(result)
result = [['vn', 'no'], ['n', 'o'], ['n', 'N'], ['n'], ['n', 'nP']]
```

graphid.util.**tag_hist**(*tags_list*)

1.2 Submodules

1.2.1 graphid.api module

class graphid.api.**GraphID**

Bases: NiceRepr

Public API for the Graph ID algorithm

Example

```
>>> # DISABLE_DOCTEST
>>> for query in iter(self):
>>>     feedback = oracle.review(query)
>>>     self.add_feedback(feedback)
```

add_annots_from(*annots*)

add_edges_from(*edges*)

add_edge(*edge*, *evidence_decision=None*)

peek(*n=0*)

Look at the next *n* items in the priority queue. When n=0 we only return one item, otherwise we return a list of items. (Note: We only make gaurentees about the first)

subgraph(*aids*)

pccs()

Positive Connected Components

Yeilds:

list: list of aids indicating all annotations currently predicted
to be some specific individual / category.

list : list of aids : Current prediction of individuals.

is_consistent()

Returns
if any PCC contains a

Return type
bool

add_feedback(*edge*, ***kwargs*)

Adds the information from a review to the graph for consideration in the dynamic inference algorithm.

1.3 Module contents

CHAPTER

TWO

INDICES AND TABLES

- genindex
- modindex

PYTHON MODULE INDEX

g

INDEX

Symbols

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

Z