Turinys

1 Documentation of script of connected components

initial version of connect_arrays (incomplete)

```
def connect_arrays(arrays):
    conc = np.concatenate(arrays)
    u, indices = np.unique(conc, return_inverse=True)

    bars = [len(n) for n in arrays[:-1]]
    bars[0] -= 1
    mask = np.ones(len(indices) - 1, dtype=bool)
    mask[np.cumsum(bars)] = False

    nodes = np.arange(len(u))
    edges = (np.array([indices[1:], indices[:-1]]).T)[mask]

    graph = igraph.Graph()
    graph.add_vertices(nodes)
    graph.add_edges(edges)
    graph_tags = graph.clusters().membership
    values = pd.DataFrame(graph_tags).groupby([0]).indices.values()
    return [u[k] for k in values]
```

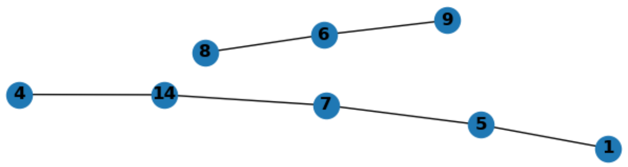
Given list of arrays, it merges intersecting ones and forms new list of array:

```
>>> connect_arrays([np.array([ 1, 5, 7]), np.array([ 4, 14, 7]), np.array([6,9]), np.array([6,8])])
[array([ 1,  4,  5,  7, 14]), array([6, 8, 9])]
```

how does it work

One can visualise result using these lines after creation of `nodes` and `edges`:

```
d = dict(zip(indices, conc))
G.add_nodes_from(nodes)
G.add_edges_from(edges)
nx.draw(G, labels=d, font_weight='bold')
plt.show()
```



At first it finds all the unique values of concatenation of lists, in sorted order and their indices in concatenated list:

```
conc = np.concatenate(arrays)
u, indices = np.unique(conc, return_inverse=True)
```

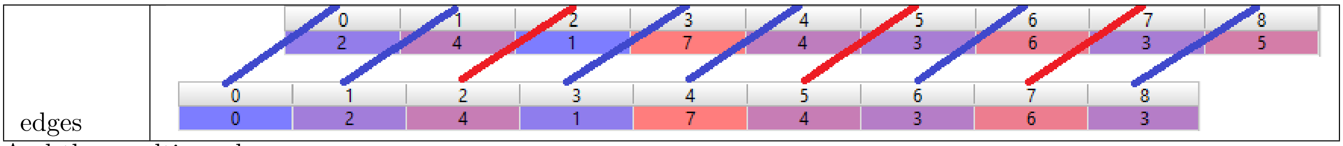
conc	1	5	7	4	14	7	6	9	6	8
u	1	4	5	6	7	8	9	14		
indices	0	2	4	1	7	4	3	6	3	5

We need to keep indices in memory because later algorithm will work with graph where only consecutive values of nodes (0, 1, 2...) are allowed. Secondly it uses lengths of initial arrays to create mask that tells bar positions of arrays to be concatenated:

mask		0	1	2	3	4	5	6	7	8	9
	0	True	True	False	True	True	False	True	False	True	True

Later this mask is used to filter locations of items of indices that are kept as edges:

```
nodes = np.arange(len(u))
edges = (np.array([indices[1:], indices[:-1]]).T)[mask[:-1]]
```



And the resulting edges are:

	0	1
0	2	0
1	4	2
2	7	1
3	4	7
4	6	3
5	5	3

`graph_tags` determines tags of connected components for each node like so:

	0	1	2	3	4	5	6	7
edges	0	0	0	1	0	1	1	0

`values` are indexes of groups of `graph_tags`, check also here.

final version of connect_arrays

`walk_components` (defined later) that uses output of `walk` needs to determine indexes of components that forms groups. This raises a need to extend `connect_arrays` method in order to return not only merged arrays but indexes of initial arrays that forms separate groups. One needs to add the following script in `def` after calculation of `values` and extend an output like so:

```
if return_groups:
    graph_tags = np.array(graph_tags)
    pid = [n[0] if len(n) else u[-1] + 1 for n in
           arrays]  # if items is zero, add non-existent item larger than any

    group_ix = np.searchsorted(u, pid)
    bad_indexes = group_ix == len(u)
    group_ix[bad_indexes] = 0  # stuff I forced to do that to avoid index error
    group_tags = graph_tags[group_ix]

    group_tags = group_tags.astype(float)
    group_tags[bad_indexes] = np.nan

    groups = pd.DataFrame(group_tags).groupby([0]).indices.values()
    return [u[k] for k in values], groups
```

Redefined `connect_arrays` method behaves this way now:

```
>>> connect_arrays([np.array([ 1, 5, 7]), np.array([ 4, 14, 7]), np.array([6,9]), np.array([6,8])],
return_groups=True)
[array([ 1,  4,  5,  7, 14]), array([6, 8, 9])],
dict_values([array([0, 1], dtype=int64), array([2, 3], dtype=int64)])
```

tag_merge

```
def tag_merge(comps):
    comps_merge = np.concatenate(comps)
    comps_tags = np.concatenate([np.repeat(i, len(comps[i])) for i in range(len(comps))])
    c_idx = np.argsort(comps_merge)
    comps_merge, comps_tags = comps_merge[c_idx], comps_tags[c_idx]
    return comps_merge, comps_tags
```

Given list of arrays that doesn't intersect pairwise, return concatenate these arrays and return sorted concatenation and corresponding tags of groups for each item.

`comps_merge` and `comps_tags`:

1	5	9	4	14	7	3	10	6	8
0	0	0	1	1	1	2	2	3	3

Output is both these arrays rearranged so that `comps_merge` is sorted:

1	3	4	5	6	7	8	9	10	14
0	2	1	0	3	1	3	0	2	1

mask_idx_over

Array that tells which elements of intersectable are in baselist. REQUIREMENT: baselist must be sorted. This problem was also proposed on StackOverflow

```
def mask_idx_over(baselist, intersectable, return_items = False):
    baselist_loc = np.searchsorted(baselist, intersectable)
    baselist_loc[baselist_loc == len(baselist)] = 0
    # finds which cell of baselist are in conjunction with intersectable
    mask = baselist[baselist_loc] == intersectable
    if return_items:
        return mask, baselist_loc[mask], intersectable[mask]
    return mask, baselist_loc[mask]
```

For `baselist = [1, 2, 4, 9]` and `intersectable = [1, 4, 3, 2, 5]` it returns `mask = [T, T, F, T, F]`. Returns also locations of intersection items in `baselist`. In a following example it is [0, 2, 1].

If `return_items` returns also intersection items, both sorted by presence in intersectable: [1, 4, 2].

intersect_components

Returns list of intersections of each item in `cc_list` with `base_comp`.

```
def intersect_components(base_comp, cc_list):
    masks_list = []
    for comp in cc_list:  # intersection of component with image matters
        mask, idx = mask_idx_over(base_comp, comp)
        masks_list.append(comp[mask])
    return masks_list
```

For `base_comp = [1,2,3,4,5,6,7,8,9,10]` and `cc_list = [[1,11], [2,12], [4,14,13,9]]` returns `masks_list = [[1], [2], [4,9]]`

walks

Given list of `base_comps`, return list of walk for each

```
def walk(base_comps, cc_list):
    comps_merge, comps_tags = tag_merge(base_comps)
    I = intersect_components(comps_merge, cc_list)
    ci_merge, ci_tags = tag_merge(I)

    walks = []
    for vcomp in base_comps:
        # find cells of vcomp that are in conjunction with cc_list
        mask, _, vcomp_intersection = mask_idx_over(ci_merge, vcomp, return_items=True)
        cc_subidx = [idx for idx in ci_tags[np.searchsorted(ci_merge, vcomp_intersection)]]
        walks.append(cc_subidx)
    return walks
```

```
walk([np.array([1,4,3,2,5]), np.array([6,7,10,9,8])],
cc_list=[np.array([1,11]), np.array([12,2]), np.array([4,14,13,9])])
>>> [[0, 2, 1], [2]]
```

For each array in `base_comps` determines ids of lists of `cc_list` for each item in `base_comps` that hits this item in at least one point. Like in this example:

- `array([1,4,3,2,5])` intersects with `cc_list[0]`, `cc_list[2]` and `cc_list[1]`.
- `array([6, 7, 10, 9, 8])` intersects with `cc_list[2]`.

walk_components

Given list of `base_comps`, return list of walk for each

```
def walk_components(base_comps, cc_list):
    cc_list = np.array(cc_list)
    walks = walk(base_comps, cc_list)
    _, groups = connect_arrays(walks, return_groups=True)
    suitable_groups = np.array(list(groups))
    # filter param reduces amount of groups, to be documented later
    suitable_walk_idx = np.concatenate(suitable_groups)
    walks = [walks[i] for i in suitable_walk_idx]
    return [np.concatenate(cc_list[subidx]) for subidx in walks]
```

```
walk([np.array([1,4,3,2,5]), np.array([6,7,10,9,8])],
cc_list=[np.array([1,11]), np.array([12,2]), np.array([4,14,13,9])])
>>> [[0, 2, 1], [2]]
walk_components([np.array([1,4,3,2,5]), np.array([6,7,10,9,8])],
cc_list=[np.array([1,11]), np.array([12,2]), np.array([4,14,13,9])])
>>> [array([ 1, 11,  4, 14, 13,  9, 12,  2]), array([ 4, 14, 13,  9])]
```

- `array([1,4,3,2,5])` intersects with `cc_list[0]`, `cc_list[2]` and `cc_list[1]`.
- `array([6, 7, 10, 9, 8])` intersects with `cc_list[2]`.