leidenalg Documentation

Release 0.10.2.dev15+g56e7241.d20231013

V.A. Traag

Oct 13, 2023

CONTENTS

1 Installation **1**

2 Introduction **3**
2.1 References . . . 6

3 Advanced **7**
3.1 Optimiser . . . 7
3.2 Resolution profile . . . 10
3.3 Fixed nodes . . . 11
3.4 Maximum community size . . . 11
3.5 References . . . 11

4 Multiplex **13**
4.1 Layer multiplex . . . 13
4.2 Slices to layers . . . 16
4.3 Temporal community detection . . . 19
4.4 References . . . 20

5 Implementation **21**
5.1 Python . . . 21

6 Reference **23**
6.1 Reference . . . 23

Python Module Index **51**

Index **53**

CHAPTER

ONE

INSTALLATION

In short: `pip install leidenalg`. Alternatively, use Anaconda and get the conda packages from the conda-forge channel, which supports both Unix, Mac OS and Windows.

For Unix like systems it is possible to install from source. For Windows this is overly complicated, and you are recommended to use the binary wheels. There are two things that are needed by this package: the igraph `C` core library and the python-igraph python package. For both, please see http://igraph.org.

Make sure you have all necessary tools for compilation. In Ubuntu this can be installed using `sudo apt-get install build-essential`, please refer to the documentation for your specific system. Make sure that not only `gcc` is installed, but also `g++`, as the `leidenalg` package is programmed in `C++`.

You can check if all went well by running a variety of tests using `python setup.py test`.

There are basically two installation modes, similar to the python-igraph package itself (from which most of the `setup.py` comes).

1. No `C` core library is installed yet. The `C` core library of igraph that is provided within the `leidenalg` package is compiled.
2. A `C` core library is already installed. In this case, you may link dynamically to the already installed version by specifying `--no-pkg-config`. This is probably also the version that is used by the igraph package, but you may want to double check this.

In case the `python-igraph` package is already installed before, make sure that both use the **same versions** (at least the same minor version, which should be API compatible).

CHAPTER

TWO

INTRODUCTION

The *leidenalg* package facilitates community detection of networks and builds on the package `igraph`. We abbreviate the *leidenalg* package as `la` and the `igraph` package as `ig` in all `Python` code throughout this documentation. Although the options in the *leidenalg* community detection package are extensive, most people are presumably simply interested in detecting communities with a robust method that works well. This introduction explains how to do that.

For those without patience (and some prior experience), if you simply want to detect communities given a graph `G` using modularity, you simply use

```
>>> partition = la.find_partition(G, la.ModularityVertexPartition);
```

That's it.

The result `partition` is in this case a *ModularityVertexPartition* which is derived from the `igraph` type `ig.VertexClustering`, see the documentation for more details.

Why then should you use this package rather than for example the Louvain algorithm `community_multilevel()` built into `igraph`? If you want to use modularity, and you work with a simple undirected, unweighted graph, then indeed you may use the built-in method. For anything else, the functionality is not built-in and this package is for you. Moreover, the Leiden algorithm is typically faster than the Louvain algorithm and returns partitions of a higher quality.

For those less familiar with `igraph`, let us work out an example more fully. First, we need to import the relevant packages:

```
>>> import igraph as ig
>>> import leidenalg as la
```

Let us then look at one of the most famous examples of network science: the Zachary karate club (it even has a prize named after it):

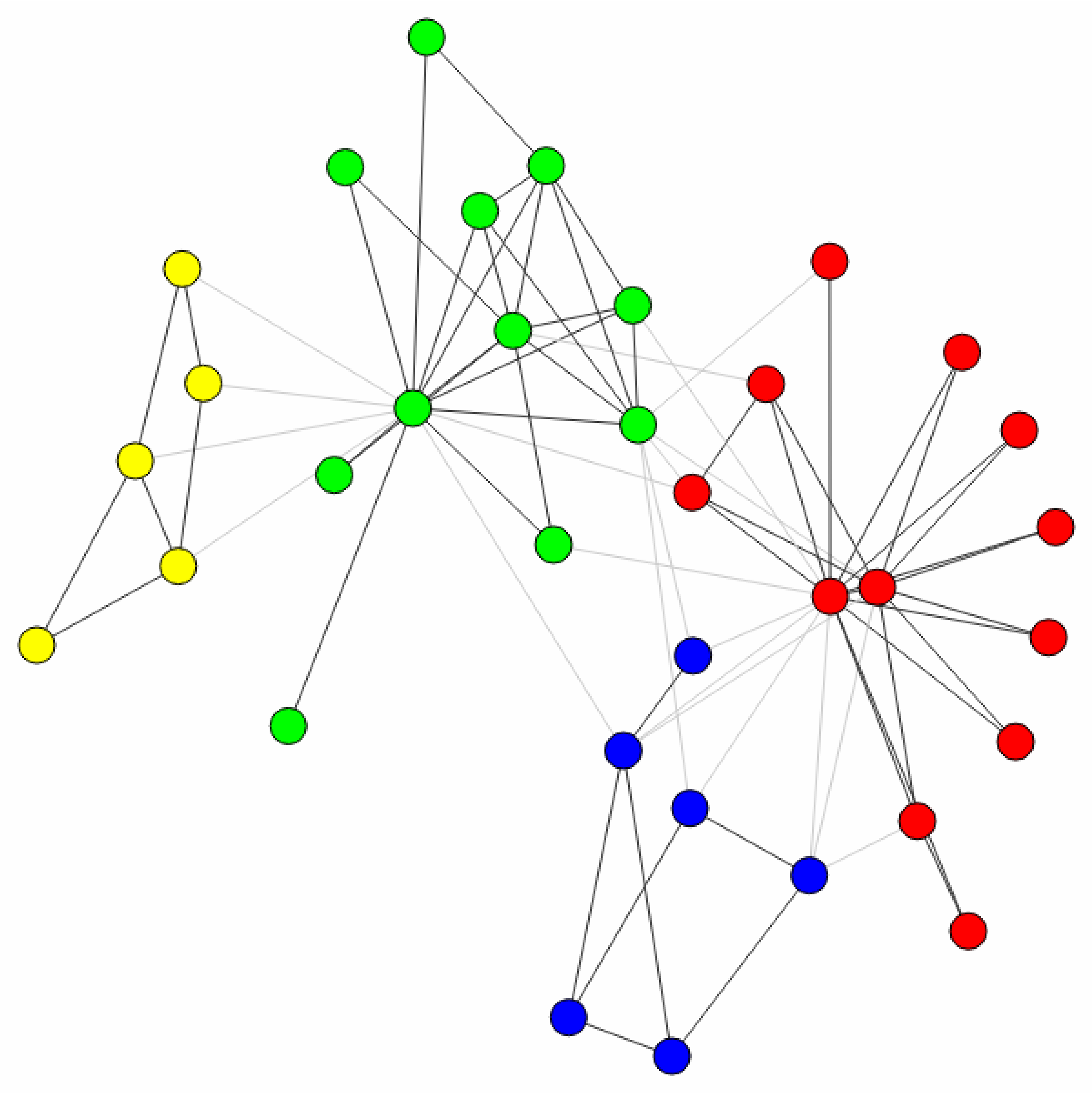
```
>>> G = ig.Graph.Famous('Zachary')
```

Now detecting communities with modularity is straightforward, as demonstrated earlier:

```
>>> partition = la.find_partition(G, la.ModularityVertexPartition)
```

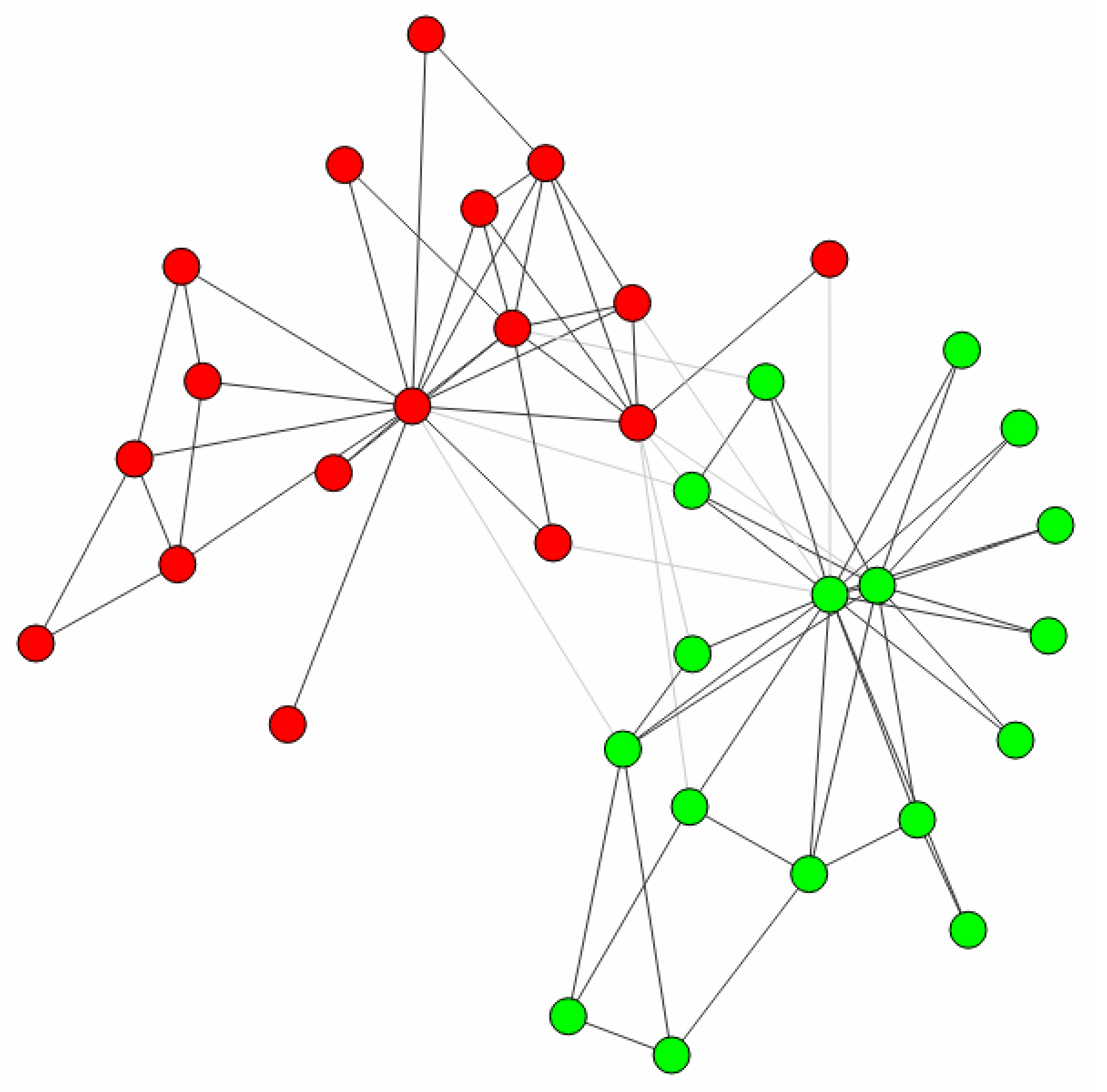
You can simply plot the results as follows:

```
>>> ig.plot(partition)
```



In this case, the algorithm actually finds the optimal partition (for small graphs like these you can check this using `community_optimal_modularity()` in the `igraph` package), but this is generally not the case (although the algorithm should do well). Although this is the optimal partition, it does not correspond to the split in two factions that was observed for this particular network. We can uncover that split in two using a different method, *`CPMVertexPartition`*:

```
>>> partition = la.find_partition(G, la.CPMVertexPartition,
...                                    resolution_parameter = 0.05);
>>> ig.plot(partition)
```



Note that any additional `**kwargs` passed to *`find_partition()`* is passed on to the constructor of the given `partition_type`. In this case, we can pass the `resolution_parameter`, but we could also pass `weights` or `node_sizes`.

This is the real benefit of using this package: it provides implementations for six different methods (see *Reference*), and works also on directed and weighted graphs. In addition, it also provides flexible functionality for customizing to some extent the optimisation routines (see *Advanced*). Finally, it also allows to work with more complex multiplex graphs (see *Multiplex*).

The Leiden algorithm [1] extends the Louvain algorithm [2], which is widely seen as one of the best algorithms for detecting communities. However, the Louvain algorithm can lead to arbitrarily badly connected communities, whereas the Leiden algorithm guarantees communities are well-connected. In fact, it converges towards a partition in which all subsets of all communities are locally optimally assigned. Finally, the Leiden algorithm is also much faster, because it relies on a fast local move routine. The "canonical" Leiden algorithm is implemented in `Java` and is faster than this implementation, but less extensive.

2.1 References

CHAPTER

THREE

ADVANCED

The basic interface explained in the *Introduction* should provide you enough to start detecting communities. However, perhaps you want to improve the partitions further or want to do some more advanced analysis. In this section, we will explain this in more detail.

3.1 Optimiser

Although the package provides simple access to the function `find_partition()`, there is actually an underlying `Optimiser` class that is doing the actual work. We can also explicitly construct an `Optimiser` object:

```
>>> optimiser = la.Optimiser()
```

The function `find_partition()` then does nothing else then calling `optimise_partition()` on the provided partition.

```
>>> diff = optimiser.optimise_partition(partition)
```

`optimise_partition()` simply tries to improve any provided partition. We can thus try to repeatedly call `optimise_partition()` to keep on improving the current partition:

```
>>> G = ig.Graph.Erdos_Renyi(100, p=5./100)
>>> partition = la.ModularityVertexPartition(G)
>>> diff = 1
>>> while diff > 0:
...   diff = optimiser.optimise_partition(partition)
```

Even if a call to `optimise_partition()` did not improve the current partition, it is still possible that a next call will improve the partition. Of course, if the current partition is already optimal, this will never happen, but it is not possible to decide whether a partition is optimal.

This functionality of repeating multiple iterations is actually already built-in. You can simply call

```
>>> diff = optimiser.optimise_partition(partition, n_iterations=10)
```

If `n_iterations < 0` the optimiser continues iterating until it encounters an iterations that did not improve the partition.

The `optimise_partition()` itself is built on two other basic algorithms: `move_nodes()` and `merge_nodes()`. You can also call these functions yourself. For example:

```
>>> diff = optimiser.move_nodes(partition)
```

or

```
>>> diff = optimiser.merge_nodes(partition)
```

The simpler Louvain algorithm aggregates the partition and repeats the *move_nodes()* on the aggregated partition. We can easily emulate that:

```
>>> partition = la.ModularityVertexPartition(G)
>>> while optimiser.move_nodes(partition) > 0:
...   partition = partition.aggregate_partition()
```

This summarises the whole Louvain algorithm in just three lines of code. Although this finds the final aggregate partition, it leaves unclear the actual partition on the level of the individual nodes. In order to do that, we need to update the membership based on the aggregate partition, for which we use the function *from_coarse_partition()*.

```
>>> partition = la.ModularityVertexPartition(G)
>>> partition_agg = partition.aggregate_partition()
>>> while optimiser.move_nodes(partition_agg) > 0:
...   partition.from_coarse_partition(partition_agg)
...   partition_agg = partition_agg.aggregate_partition()
```

Now `partition_agg` contains the aggregate partition and `partition` contains the actual partition of the original graph G. Of course, `partition_agg.quality() == partition.quality()` (save some rounding).

Instead of *move_nodes()*, you could also use *merge_nodes()*. These functions depend on choosing particular alternative communities: the documentation of the functions provides more detail.

One possibility is that rather than aggregating the partition based on the current partition, you can first refine the partition and then aggregate it. This is what is done in the Leiden algorithm, and can be done using the functions *move_nodes_constrained()* and *merge_nodes_constrained()*. Implementing this, you end up with the following high-level implementation of the Leiden algorithm:

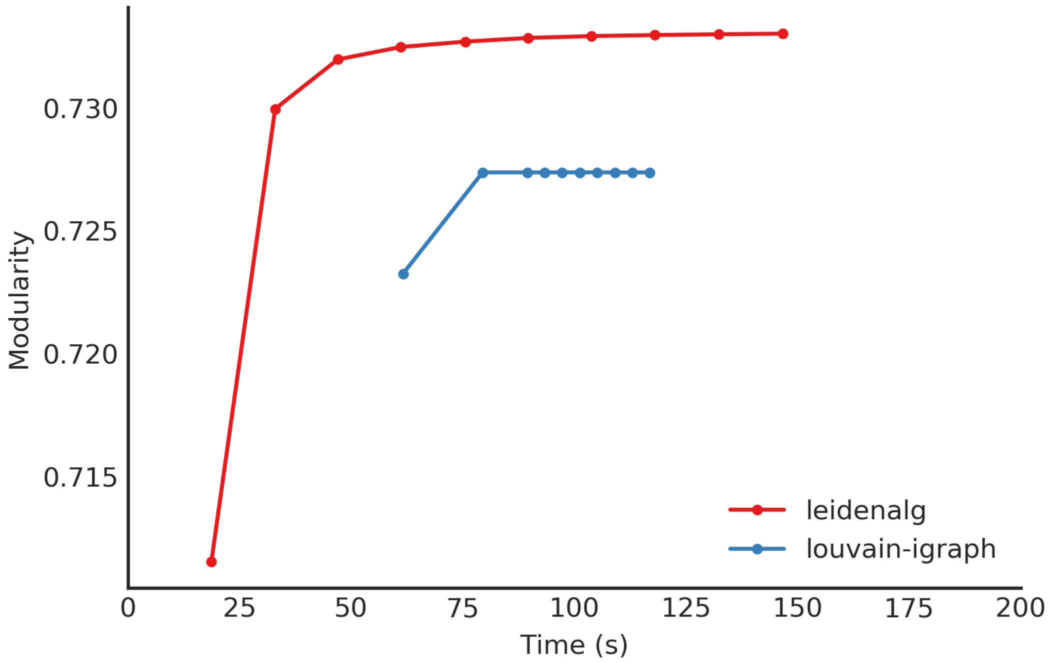
```
>>> # Set initial partition
>>> partition = la.ModularityVertexPartition(G)
>>> refined_partition = la.ModularityVertexPartition(G)
>>> partition_agg = refined_partition.aggregate_partition()
>>>
>>> while optimiser.move_nodes(partition_agg):
...
...   # Get individual membership for partition
...   partition.from_coarse_partition(partition_agg, refined_partition.membership)
...
...   # Refine partition
...   refined_partition = la.ModularityVertexPartition(G)
...   optimiser.merge_nodes_constrained(refined_partition, partition)
...
...   # Define aggregate partition on refined partition
...   partition_agg = refined_partition.aggregate_partition()
...
...   # But use membership of actual partition
...   aggregate_membership = [None] * len(refined_partition)
...   for i in range(G.vcount()):
...     aggregate_membership[refined_partition.membership[i]] = partition.membership[i]
...   partition_agg.set_membership(aggregate_membership)
```

These functions in turn rely on two key functions of the partition: `diff_move()` and `move_node()`. The first calculates the difference when moving a node, and the latter actually moves the node, and updates all necessary internal administration. The `move_nodes()` then does something as follows

```
>>> for v in G.vs:
...   best_comm = max(range(len(partition)),
...                   key=lambda c: partition.diff_move(v.index, c))
...   partition.move_node(v.index, best_comm)
```

The actual implementation is more complicated, but this gives the general idea.

This package builds on a previous implementation of the Louvain algorithm in louvain-igraph. To illustrate the difference between `louvain-igraph` and `leidenalg`, we ran both algorithms for 10 iterations on a Youtube network of more than 1 million nodes and almost 3 million edges.



The results are quite clear: Leiden is able to achieve a higher modularity in less time. It also points out that it is usually a good idea to run Leiden for at least two iterations; this is also the default setting.

Note that even if the Leiden algorithm did not find any improvement in this iteration, it is always possible that it will find some improvement in the next iteration.

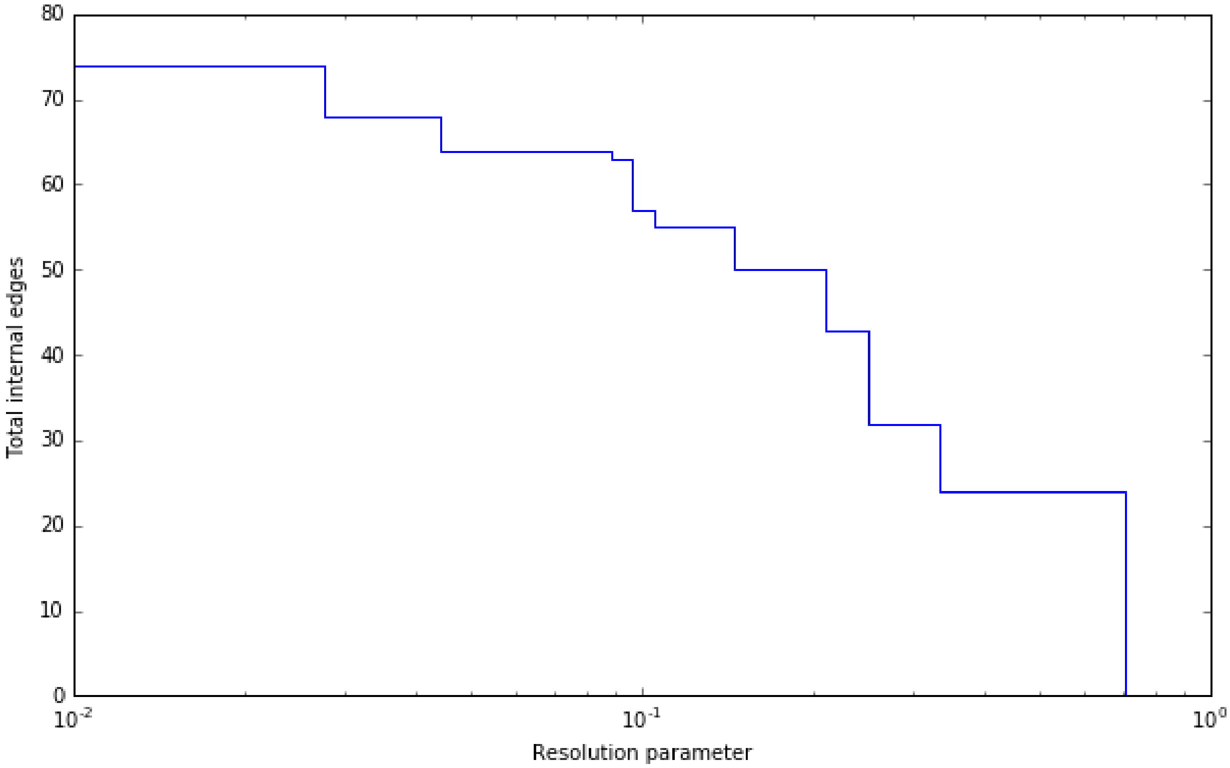
3.2 Resolution profile

Some methods accept so-called resolution parameters, such as *CPMVertexPartition* or *RBConfigurationVertexPartition*. Although some methods may seem to have some 'natural' resolution, in reality this is often quite arbitrary. However, the methods implemented here (which depend in a linear way on resolution parameters) allow for an effective scanning of a full range for the resolution parameter. In particular, these methods somehow can be formulated as $Q = E - \gamma N$ where E and N are some other quantities. In the case for *CPMVertexPartition* for example, $E = \sum_c m_c$ is the number of internal edges and $N = \sum_c \binom{n_c}{2}$ is the sum of the internal possible edges. The essential insight for these formulations[1] is that if there is an optimal partition for both γ_1 and γ_2 then the partition is also optimal for all $\gamma_1 \leq \gamma \leq \gamma_2$.

Such a resolution profile can be constructed using the *Optimiser* object.

```
>>> G = ig.Graph.Famous('Zachary')
>>> optimiser = la.Optimiser()
>>> profile = optimiser.resolution_profile(G, la.CPMVertexPartition,
...                                        resolution_range=(0,1))
```

Plotting the resolution parameter versus the total number of internal edges we thus obtain something as follows:



Now `profile` contains a list of partitions of the specified type (*CPMVertexPartition* in this case) for resolution parameters at which there was a change. In particular, `profile[i]` should be better until `profile[i+1]`, or stated otherwise for any resolution parameter between `profile[i].resolution_parameter` and `profile[i+1].resolution_parameter` the partition at position `i` should be better. Of course, there will be some variations because *optimise_partition()* will find partitions of varying quality. The change points can then also vary for different runs.

---

[1] Traag, V. A., Krings, G., & Van Dooren, P. (2013). Significant scales in community structure. Scientific Reports, 3, 2930. 10.1038/srep02930

This function repeatedly calls *`optimise_partition()`* and can therefore require a lot of time. Especially for resolution parameters right around a change point there may be many possible partitions, thus requiring a lot of runs.

3.3 Fixed nodes

For some purposes, it might be beneficial to only update part of a partition. For example, perhaps we previously already ran the Leiden algorithm on some dataset, and did some analysis on the resulting partition. If we then gather new data, and in particular new nodes, it might be useful to keep the previous community assignments fixed, while only updating the community assignments for the new nodes. This can be done using the `is_membership_fixed` argument of `find_partition()`, see[2] for some details.

For example, suppose we previously detected `partition` for graph `G`, which was extended to graph `G2`. Assuming that the previously exiting nodes are identical, we could create a new partition by doing

```
>>> new_membership = list(range(G2.vcount()))
... new_membership[:G.vcount()] = partition.membership
```

We can then only update the community assignments for the new nodes as follows

```
>>> new_partition = la.CPMVertexPartition(G2, new_membership,
...                                           resolution_parameter=partition.resolution_
↪parameter)
... is_membership_fixed = [i < G.vcount() for i in range(G2.vcount())]
>>> diff = optimiser.optimise_partition(new_partition, is_membership_fixed=is_membership_
↪fixed)
```

In this example we used *`CPMVertexPartition`*. but any other `VertexPartition` would work as well.

3.4 Maximum community size

In some cases, you may want to restrict the community sizes. It is possible to indicate this by setting the *`max_comm_size`* parameter so that this constraint is taken into account during optimisation. In addition, it is possible to pass this parameter directly when using *`find_partition()`*. For example

```
>>> partition = la.find_partition(G, la.ModularityVertexPartition, max_comm_size=10)
```

3.5 References

---

[2] Zanini, F., Berghuis, B. A., Jones, R. C., Robilant, B. N. di, Nong, R. Y., Norton, J., Clarke, Michael F., Quake, S. R. (2019). northstar: leveraging cell atlases to identify healthy and neoplastic cells in transcriptomes from human tumors. BioRxiv, 820928. 10.1101/820928

CHAPTER FOUR

MULTIPLEX

The implementation of multiplex community detection builds on ideas in[1]. The most basic form simply considers two or more graphs which are defined on the same vertex set, but which have differing edge sets. In this context, each node is identified with a single community, and cannot have different communities for different graphs. We call this *layers* of graphs in this context. This format is actually more flexible than it looks, but you have to construct the layer graphs in a smart way. Instead of having layers of graphs which are always identified on the same vertex set, you could define *slices* of graphs which do not necessarily have the same vertex set. Using slices we would like to assign a node to a community for each slice, so that the community for a node can be different for different slices, rather than always being the same for all layers. We can translate *slices* into *layers* but it is not an easy transformation to grasp fully. But by doing so, we can again rely on the same machinery we developed for dealing with layers.

Throughout the remained of this section, we assume an optimiser has been created:

```
>>> optimiser = la.Optimiser()
```

4.1 Layer multiplex

If we have two graphs which are identified on exactly the same vertex set, we say we have two *layers*. For example, suppose graph `G_telephone` contains the communication between friends over the telephone and that the graph `G_email` contains the communication between friends via mail. The exact same vertex set then means that `G_telephone.vs[i]` is identical to the node `G_email.vs[i]`. For each layer we can separately specify the type of partition that we look for. In principle they could be different for each layer, but for now we will assume the type of partition is the same for all layers. The quality of all partitions combined is simply the sum of the individual qualities for the various partitions, weighted by the `layer_weight`. If we denote by q_k the quality of layer k and the weight by w_k, the overall quality is then

$$q = \sum_k w_k q_k.$$

The optimisation algorithm is no different from the standard algorithm. We simply calculate the overall difference of moving a node to another community as the sum of the individual differences in all partitions. The rest (aggregating and repeating on the aggregate partition) simple proceeds as usual.

The most straightforward way to use this is then to use *`find_partition_multiplex()`*:

```
>>> membership, improv = la.find_partition_multiplex(
...                         [G_telephone, G_email],
...                         la.ModularityVertexPartition);
```

---

[1] Mucha, P. J., Richardson, T., Macon, K., Porter, M. A., & Onnela, J.-P. (2010). Community structure in time-dependent, multiscale, and multiplex networks. Science, 328(5980), 876–8. 10.1126/science.1184819

Note: You may need to carefully reflect how you want to weigh the importance of an individual layer. Since the *ModularityVertexPartition* is normalised by the number of links, you essentially weigh layers the same, independent of the number of links. This may be undesirable, in which case it may be better to use `RBConfigurationVertexPartition`, which is unnormalised. Alternatively, you may specify different `layer_weights`.

Similar to the simpler function *find_partition()*, it is a simple helper function. The function returns a membership vector, because the membership for all layers is identical. You can also control the partitions and optimisation in more detail. Perhaps it is better to use *CPMVertexPartition* with different resolution parameter for example for different layers of the graph. For example, using email creates a more connected structure because multiple people can be involved in a single mail, which may require a higher resolution parameter for the email graph.

```
>>> part_telephone = la.CPMVertexPartition(
...                    G_telephone, resolution_parameter=0.01);
>>> part_email = la.CPMVertexPartition(
...                    G_email, resolution_parameter=0.3);
>>> diff = optimiser.optimise_partition_multiplex(
...                    [part_telephone, part_email]);
```

Note that `part_telephone` and `part_email` contain exactly the same partition, in the sense that `part_telephone.membership == part_email.membership`. The underlying graph is of course different, and hence the individual quality will also be different.

Some layers may have a more important role in the partition and this can be indicated by the `layer_weight`. Using half the weight for the email layer for example would be possible as follows:

```
>>> diff = optimiser.optimise_partition_multiplex(
...   [part_telephone, part_email],
...   layer_weights=[1,0.5]);
```

4.1.1 Negative links

The layer weights are especially useful when negative links are present, representing for example conflict or animosity. Most methods (except CPM) only accept positive weights. In order to deal with graphs that do have negative links, a solution is to separate the graph into two layers: one layer with positive links, the other with only negative links[2]. In general, we would like to have relatively many positive links within communities, while for negative links the opposite holds: we want many negative links between communities. We can easily do this within the multiplex layer framework by passing in a negative layer weight. For example, suppose we have a graph `G` with possibly negative weights. We can then separate it into a positive and negative graph as follows:

```
>>> G_pos = G.subgraph_edges(G.es.select(weight_gt = 0), delete_vertices=False);
>>> G_neg = G.subgraph_edges(G.es.select(weight_lt = 0), delete_vertices=False);
>>> G_neg.es['weight'] = [-w for w in G_neg.es['weight']];
```

We can then simply detect communities using;

```
>>> part_pos = la.ModularityVertexPartition(G_pos, weights='weight');
>>> part_neg = la.ModularityVertexPartition(G_neg, weights='weight');
>>> diff = optimiser.optimise_partition_multiplex(
```

(continues on next page)

---

[2] Traag, V. A., & Bruggeman, J. (2009). Community detection in networks with positive and negative links. Physical Review E, 80(3), 036115. 10.1103/PhysRevE.80.036115

(continued from previous page)

```
...     [part_pos, part_neg],
...     layer_weights=[1,-1]);
```

4.1.2 Bipartite

For some methods it may be possible to to community detection in bipartite networks. Bipartite networks are special in the sense that they have only links between the two different classes, and no links within a class are allowed. For example, there might be products and customers, and there is a link between i and j if a product i is bought by a customer j. In this case, there are no links among products, nor among customers. One possible approach is simply project this bipartite network into the one or the other class and then detect communities. But then the correspondence between the communities in the two different projections is lost. Detecting communities in the bipartite network can therefore be useful.

Setting this up requires a bit of a creative approach, which is why it is also explicitly explained here. We will explain it for the CPM method, and then show how this works the same for some related measures. In the case of CPM you would like to be able to set three different resolution parameters: one for within each class γ_0, γ_1, and one for the links between classes, γ_{01}. Then the formulation would be

$$Q = \sum_{ij} [A_{ij} - (\gamma_0 \delta(s_i, 0) + \gamma_1 \delta(s_i, 1)) \delta(s_i, s_j) - \gamma_{01} (1 - \delta(s_i, s_j))] \delta(\sigma_i, \sigma_j)$$

where s_i denotes the bipartite class of a node and σ_i the community of the node as elsewhere in the documentation. Rewriting as a sum over communities gives a bit more insight

$$Q = \sum_c (e_c - \gamma_{01} 2 n_c(0) n_c(1) - \gamma_0 n_c^2(0) - \gamma_1 n_c^2(1))$$

where $n_c(0)$ is the number of nodes in community c of class 0 (and similarly for 1) and e_c is the number of edges within community c. We denote by $n_c = n_c(0) + n_c(1)$ the total number of nodes in community c. Note that

$$\begin{aligned} n_c^2 &= (n_c(0) + n_c(1))^2 \\ &= n_c(0)^2 + 2 n_c(0) n_c(1) + n_c(1)^2 \end{aligned}$$

We then create three different layers: (1) all nodes have `node_size = 1` and all relevant links; (2) only nodes of class 0 have `node_size = 1` and no links; (3) only nodes of class 1 have `node_size = 1` and no links. If we add the first with resolution parameter γ_{01}, and the others with resolution parameters $\gamma_{01} - \gamma_0$ and $\gamma_{01} - \gamma_1$, but the latter two with a layer weight of -1 while the first layer has layer weight 1, we obtain the following:

$$\begin{aligned} Q &= \sum_c (e_c - \gamma_{01} n_c^2) - \sum_c (-(\gamma_{01} - \gamma_0) n_c(0)^2) - \sum_c (-(\gamma_{01} - \gamma_1) n_c(0)^2) \\ &= \sum_c [e_c - \gamma_{01} 2 n_c(0) n_c(1) - \gamma_{01} n_c(0)^2 - \gamma_{01} n_c(1)^2) + (\gamma_{01} - \gamma_0) n_c(0)^2 + (\gamma_{01} - \gamma_1) n_c(1)^2] \\ &= \sum_c (e_c - \gamma_{01} 2 n_c(0) n_c(1) - \gamma_0 n_c(0)^2 - \gamma_1 n_c(1)^2) \end{aligned}$$

Hence detecting communities with these three layers corresponds to detecting communities in bipartite networks. Although we worked out this example for directed network including self-loops (since it is easiest), it works out similarly for undirected networks (with or without self-loops). This only corresponds to the CPM method. However, using a little additional trick, we can also make this work for modularity. Essentially, modularity is nothing else than CPM with the `node_size` set to the degree, and the resolution parameter set to $\gamma = \frac{1}{2m}$. In particular, in general (i.e. not specifically for bipartite graph) if `node_sizes=G.degree()` we then obtain

$$Q = \sum_{ij} A_{ij} - \gamma k_i k_j$$

In the case of bipartite graphs something similar is obtained, but then correctly adapted (as long as the resolution parameter is also appropriately rescaled). Note that this is only possible for modularity for undirected graphs. Hence, we can also detect communities in bipartite networks using modularity by using this little trick. The definition of modularity for bipartite graphs is identical to the formulation of bipartite modularity provided in[3].

All of this has been implemented in the constructor `Bipartite()`. You can simply pass in a bipartite network with the classes appropriately defined in `G.vs['type']` or equivalent. This function assumes the two classes are coded by `0` and `1`, and if this is not the case it will try to convert it into such categories by `ig.UniqueIdGenerator()`.

An explicit example of this:

```
>>> p_01, p_0, p_1 = la.CPMVertexPartition.Bipartite(G,
...                     resolution_parameter_01=0.1);
>>> diff = optimiser.optimise_partition_multiplex([p_01, p_0, p_1],
...                                          layer_weights=[1, -1, -1]);
```

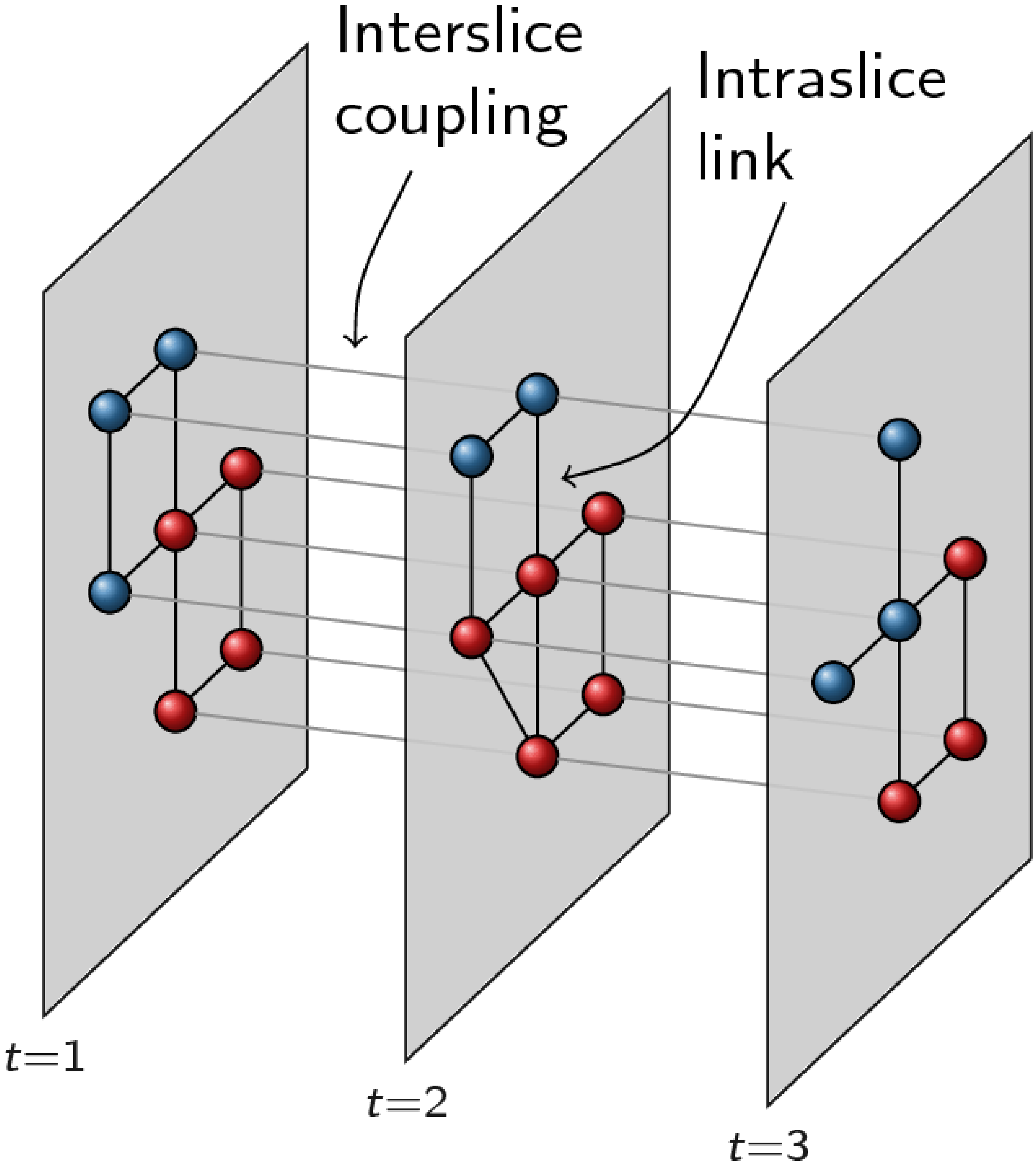
4.2 Slices to layers

The multiplex formulation as layers has two limitations: (1) each graph needs to have an identical vertex set; (2) each node is only in a single community. Ideally, one would like to relax both these requirements, so that you can work with graphs that do not need to have identical nodes and where nodes can be in different communities in different layers. For example, a person could be in one community when looking at his professional relations, but in another community looking at his personal relations. Perhaps more commonly: a person could be in one community at time 1 and in another community at time 2.

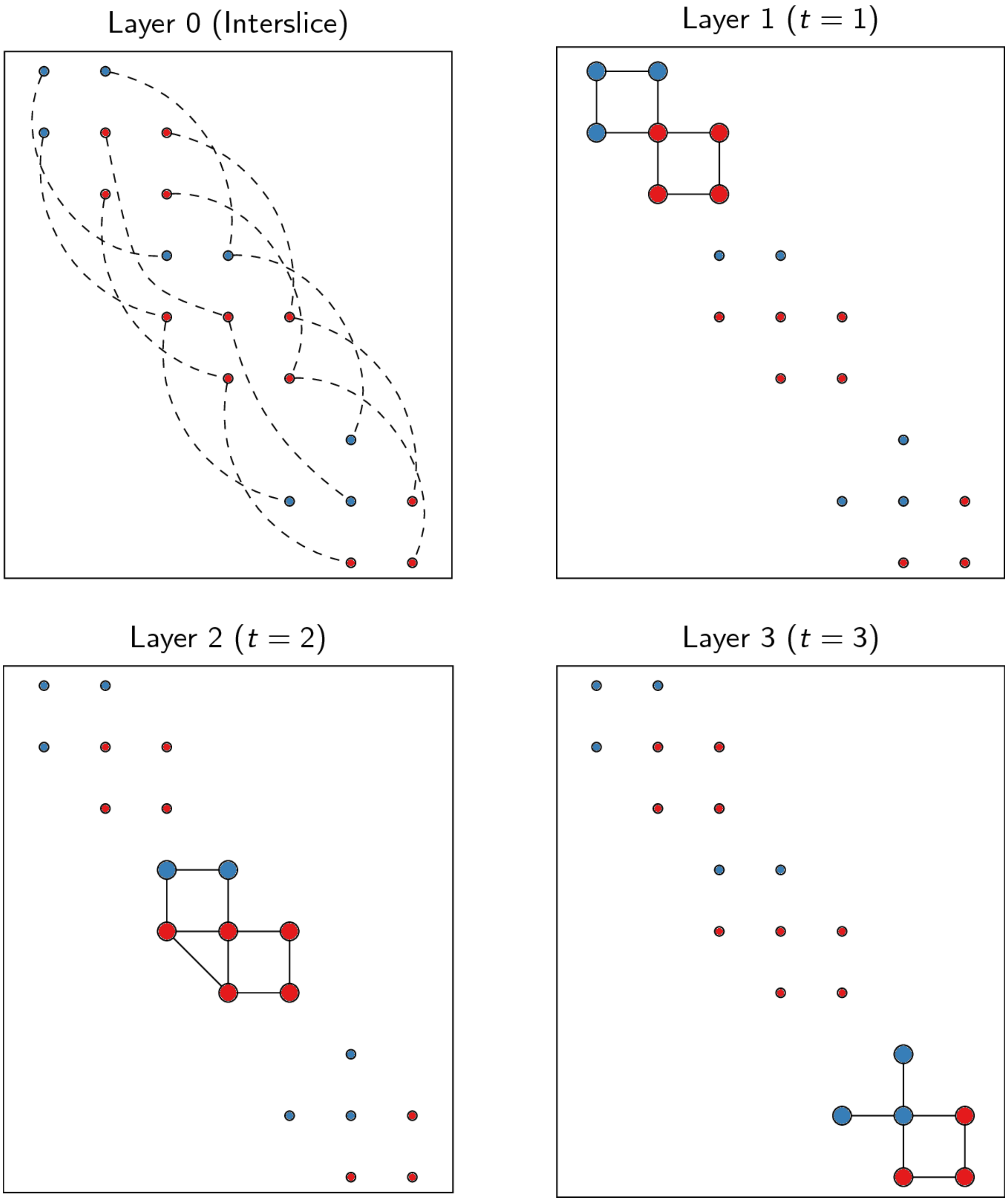
Fortunately, this is also possible with this package. We call the more general formulation *slices* in contrast to the *layers* required by the earlier functions. Slices are then just different graphs, which do not need to have the same vertex set in any way. The idea is to build one big graph out of all the slices and then decompose it again in layers that correspond with slices. The key element is that some slices are coupled: for example two consecutive time windows, or simply two different slices of types of relations. Because any two slices can be coupled in theory, we represent the coupling itself again with a graph. The nodes of this *coupling graph* thus are slices, and the (possibly weighted) links in the coupling graph represent the (possibly weighted) couplings between slices. Below an example with three different time slices, where slice 1 is coupled to slice 2, which in turn is coupled to slice 3:

---

[3] Barber, M. J. (2007). Modularity and community detection in bipartite networks. Physical Review E, 76(6), 066102. 10.1103/PhysRevE.76.066102



The coupling graph thus consists of three nodes and a simple line structure: `1 -- 2 -- 3`. We convert this into layers by putting all nodes of all slices in one big network. Each node is thus represented by a tuple `(node, slice)` in a certain sense. Out of this big network, we then only take those edges that are defined between nodes of the same slice, which then constitutes a single layer. Finally, we need one more layer for the couplings. In addition, for methods such as *`CPMVertexPartition`*, so-called `node_sizes` are required, and for them to properly function, they should be set to 0 (which is handled appropriately by the package). We thus obtain equally many layers as we have slices, and we need one more layer for representing the interslice couplings. For the example provided above, we thus obtain the following:



Layer 0 (Interslice)

Layer 1 ($t = 1$)

Layer 2 ($t = 2$)

Layer 3 ($t = 3$)

To transform slices into layers using a coupling graph, this package provides `layers_to_slices()`. For the example above, this would function as follows. First create the coupling graph assuming we have three slices `G_1`, `G_2` and `G_3`:

```
>>> G_coupling = ig.Graph.Formula('1 -- 2 -- 3');
>>> G_coupling.es['weight'] = 0.1; # Interslice coupling strength
>>> G_coupling.vs['slice'] = [G_1, G_2, G_3]
```

Then we convert them to layers

```
>>> layers, interslice_layer, G_full = la.slices_to_layers(G_coupling);
```

Now we still have to create partitions for all the layers. We can freely choose here to use the same partition types for all partitions, or to use different types for different layers.

> **Warning:** The interslice layer should usually be of type *`CPMVertexPartition`* with a `resolution_parameter=0` and `node_sizes` set to 0. The `G.vs[node_size]` is automatically set to 0 for all nodes in the interslice layer in *`slices_to_layers()`*, so you can simply pass in the attribute `node_size`. Unless you know what you are doing, simply use these settings.

> **Warning:** When using methods that accept a node_size argument, this should always be used. This is the case for *`CPMVertexPartition`*, *`RBERVertexPartition`*, *`SurpriseVertexPartition`* and *`SignificanceVertexPartition`*.

```
>>> partitions = [la.CPMVertexPartition(H, node_sizes='node_size',
...                                          weights='weight', resolution_
↪parameter=gamma)
...               for H in layers];
>>> interslice_partition = la.CPMVertexPartition(interslice_layer, resolution_
↪parameter=0,
...                                                    node_sizes='node_size', weights=
↪'weight');
```

You can then simply optimise these partitions as before using *`optimise_partition_multiplex()`*:

```
>>> diff = optimiser.optimise_partition_multiplex(partitions + [interslice_partition]);
```

4.3 Temporal community detection

One of the most common tasks for converting slices to layers is that we have slices at different points in time. We call this temporal community detection. Because it is such a common task, we provide several helper functions to simplify the above process. Let us assume again that we have three slices `G_1`, `G_2` and `G_3` as in the example above. The most straightforward function is *`find_partition_temporal()`*:

```
>>> membership, improvement = la.find_partition_temporal(
...                             [G_1, G_2, G_3],
...                             la.CPMVertexPartition,
...                             interslice_weight=0.1,
...                             resolution_parameter=gamma)
```

This function only returns the membership vectors for the different time slices, rather than actual partitions.

Rather than directly detecting communities, you can also obtain the actual partitions in a slightly more convenient way using *`time_slices_to_layers()`*:

```
>>> layers, interslice_layer, G_full = \
...               la.time_slices_to_layers([G_1, G_2, G_3],
```

(continues on next page)

(continued from previous page)

```
...                                               interslice_weight=0.1);
>>> partitions = [la.CPMVertexPartition(H, node_sizes='node_size',
...                                        weights='weight',
...                                        resolution_parameter=gamma)
...               for H in layers];
>>> interslice_partition = \
...               la.CPMVertexPartition(interslice_layer, resolution_parameter=0,
...                                        node_sizes='node_size', weights='weight');
>>> diff = optimiser.optimise_partition_multiplex(partitions + [interslice_partition]);
```

Both these functions assume that the interslice coupling is always identical for all slices. If you want more finegrained control, you will have to use the earlier explained functions.

4.4 References

CHAPTER
FIVE

IMPLEMENTATION

If you have a cool new idea for a better method, and you want to optimise it, you can easily plug it in the current package. This section explains how the package is setup internally, and how you can extend it. Most of this concerns C++, and Python only comes in when exposing the classes from C++.

The core of the Leiden algorithm is implemented in C++ in the package `libleidenalg`, available from https://github.com/vtraag/libleidenalg. This C++ library is used as the core for the Python package, which is just an interface to the underlying C++ library. New methods need to be added to the C++ library first, and then exposed to the Python interface. For more information, please see the contributing guide of the `libleidenalg` package at https://github.com/vtraag/libleidenalg/blob/main/CONTRIBUTING.md

5.1 Python

Exposing the method to `python` takes a bit more effort. There are various places in which you need to change/add things. In the following, we assume you created a new class called `CoolVertexPartition` in `libleidenalg`. Please then follow the following steps:

1. Your own new VertexPartition class should add some specific methods. In particular, you need to ensure you create a method

   ```
   CoolVertexPartition* CoolVertexPartition::create(Graph* graph)
   {
     return new CoolVertexPartition(graph);
   }
   ```

   and

   ```
   CoolVertexPartition* CoolVertexPartition::create(Graph* graph, vector<size_t> const&
   ↪ membership)
   {
     return new CoolVertexPartition(graph, membership);
   }
   ```

   These methods ensure that based on a current partition, we can create a new partition (without knowing its type).

2. In `python_partition_interface.cpp` some methods need to be added. In particular

   ```
   PyObject* _new_CoolVertexPartition(PyObject *self, PyObject *args, PyObject *keywds)
   ```

   You should be able to simply copy an existing method, and adapt it to your own needs.

3. These methods need to be exposed in `pynterface.h`. In particular, you need to add the method you created in step (2) to `leiden_funcs[]`. Again, you should be able to simply copy an existing line.

4. You can then finally create the Python class in `VertexPartition.py`. The base class derives from the `VertexClustering` from `igraph`, so that it is compatible with all operations in `igraph`. You should add the method as follows:

   ```
   class CoolVertexPartition(MutableVertexPartition):

     def __init__(self, ... ):
       ...
   ```

   Again, you should be able to copy the outline for another class and adapt it to your own needs. Don't forget to change to `docstring` to update the documentation so that everybody knows how your new cool method works.

5. Expose your newly created `python` class directly in `__init__.py` by importing it:

   ```
   from .VertexPartition import CoolVertexPartition
   ```

That's it! You're done and should now be able to find communities using your new `CoolVertexPartition`:

```
>>> la.find_partition(G, la.CoolVertexPartition);
```

CHAPTER

SIX

REFERENCE

6.1 Reference

6.1.1 Module functions

This package implements the Leiden algorithm in `C++` and exposes it to python. It relies on `(python-)igraph` for it to function. Besides the relative flexibility of the implementation, it also scales well, and can be run on graphs of millions of nodes (as long as they can fit in memory). Each method is represented by a different class, all of whom derive from *`MutableVertexPartition`*. In addition, multiplex graphs are supported as layers, which also supports multislice representations.

Examples

The simplest example just finds a partition using modularity

```
>>> G = ig.Graph.Tree(100, 3)
>>> partition = la.find_partition(G, la.ModularityVertexPartition)
```

Alternatively, one can access the different optimisation routines individually and construct partitions oneself. These partitions can then be optimised by constructing an *`Optimiser`* object and running *`optimise_partition()`*.

```
>>> G = ig.Graph.Tree(100, 3)
>>> partition = la.CPMVertexPartition(G, resolution_parameter = 0.1)
>>> optimiser = la.Optimiser()
>>> diff = optimiser.optimise_partition(partition)
```

The *`Optimiser`* class contains also the different subroutines that are used internally by *`optimise_partition()`*. In addition, through the Optimiser class there are various options available for changing some of the optimisation procedure which can affect both speed and quality, which are not immediately available in *`leidenalg.find_partition()`*.

`leidenalg.`**`find_partition`**(*graph*, *partition_type*, *initial_membership=None*, *weights=None*, *n_iterations=2*, *max_comm_size=0*, *seed=None*, ***kwargs*)

Detect communities using the default settings.

This function detects communities given the specified method in the `partition_type`. This should be type derived from *`VertexPartition.MutableVertexPartition`*, e.g. *`ModularityVertexPartition`* or *`CPMVertexPartition`*. Optionally an initial membership and edge weights can be provided. Remaining `**kwargs` are passed on to the constructor of the `partition_type`, including for example a `resolution_parameter`.

Parameters

- **graph** (`ig.Graph`) – The graph for which to detect communities.
- **partition_type** (type of :class:`) – The type of partition to use for optimisation.
- **initial_membership** (*list of int*) – Initial membership for the partition. If `None` then defaults to a singleton partition.
- **weights** (*list of double, or edge attribute*) – Weights of edges. Can be either an iterable or an edge attribute.
- **n_iterations** (*int*) – Number of iterations to run the Leiden algorithm. By default, 2 iterations are run. If the number of iterations is negative, the Leiden algorithm is run until an iteration in which there was no improvement.
- **max_comm_size** (*non-negative int*) – Maximal total size of nodes in a community. If zero (the default), then communities can be of any size.
- **seed** (*int*) – Seed for the random number generator. By default uses a random seed if nothing is specified.
- ****kwargs** – Remaining keyword arguments, passed on to constructor of `partition_type`.

Returns
The optimised partition.

Return type
partition

See also:

`Optimiser.optimise_partition()`

Examples

```
>>> G = ig.Graph.Famous('Zachary')
>>> partition = la.find_partition(G, la.ModularityVertexPartition)
```

`leidenalg.`**`find_partition_multiplex`**(*graphs*, *partition_type*, *layer_weights=None*, *n_iterations=2*, *max_comm_size=0*, *seed=None*, ***kwargs*)

Detect communities for multiplex graphs.

Each graph should be defined on the same set of vertices, only the edges may differ for different graphs. See *`Optimiser.optimise_partition_multiplex()`* for a more detailed explanation.

Parameters

- **graphs** (list of `ig.Graph`) – List of `ig.Graph` graphs to optimise.
- **partition_type** (type of `MutableVertexPartition`) – The type of partition to use for optimisation (identical for all graphs).
- **layer_weights** (*list of double*) – List containing weights of each layer. If None (the default), then all layers are weighted with 1.
- **n_iterations** (*int*) – Number of iterations to run the Leiden algorithm. By default, 2 iterations are run. If the number of iterations is negative, the Leiden algorithm is run until an iteration in which there was no improvement.
- **max_comm_size** (*non-negative int*) – Maximal total size of nodes in a community. If zero (the default), then communities can be of any size.

- **seed** (*int*) – Seed for the random number generator. By default uses a random seed if nothing is specified.
- ****kwargs** – Remaining keyword arguments, passed on to constructor of `partition_type`.

Returns

- *list of int* – membership of nodes.
- *float* – Improvement in quality of combined partitions, see *Optimiser.optimise_partition_multiplex()*.

Notes

We don't return a partition in this case because a partition is always defined on a single graph. We therefore simply return the membership (which is the same for all layers).

See also:

Optimiser.optimise_partition_multiplex(), *slices_to_layers()*

Examples

```
>>> n = 100
>>> G_1 = ig.Graph.Lattice([n], 1)
>>> G_2 = ig.Graph.Lattice([n], 1)
>>> membership, improvement = la.find_partition_multiplex([G_1, G_2],
...                                                        la.
↪ModularityVertexPartition)
```

`leidenalg.`**`find_partition_temporal`**(*graphs*, *partition_type*, *interslice_weight=1*, *slice_attr='slice'*, *vertex_id_attr='id'*, *edge_type_attr='type'*, *weight_attr='weight'*, *n_iterations=2*, *max_comm_size=0*, *seed=None*, ***kwargs*)

Detect communities for temporal graphs.

Each graph is considered to represent a time slice and does not necessarily need to be defined on the same set of vertices. Nodes in two consecutive slices are identified on the basis of the `vertex_id_attr`, i.e. if two nodes in two consecutive slices have an identical value of the `vertex_id_attr` they are coupled. The `vertex_id_attr` should hence be unique in each slice. The nodes are then coupled with a weight of `interslice_weight` which is set in the edge attribute `weight_attr`. No weight is set if the `interslice_weight` is None (i.e. corresponding in practice with a weight of 1). See *time_slices_to_layers()* for a more detailed explanation.

Parameters

- **graphs** (list of `ig.Graph`) – List of `leidenalg.VertexPartition` layers to optimise.
- **partition_type** (type of *VertexPartition.MutableVertexPartition*) – The type of partition to use for optimisation (identical for all graphs).
- **interslice_weight** (*float*) – The weight of the coupling between two consecutive time slices.
- **slice_attr** (*string*) – The vertex attribute to use for indicating the slice of a node.
- **vertex_id_attr** (*string*) – The vertex to use to identify nodes.
- **edge_type_attr** (*string*) – The edge attribute to use for indicating the type of link (*interslice* or *intraslice*).

- **`weight_attr`** (*`string`*) – The edge attribute used to indicate the weight.
- **`n_iterations`** (*`int`*) – Number of iterations to run the Leiden algorithm. By default, 2 iterations are run. If the number of iterations is negative, the Leiden algorithm is run until an iteration in which there was no improvement.
- **`max_comm_size`** (*`non-negative int`*) – Maximal total size of nodes in a community. If zero (the default), then communities can be of any size.
- **`seed`** (*`int`*) – Seed for the random number generator. By default uses a random seed if nothing is specified.
- **`**kwargs`** – Remaining keyword arguments, passed on to constructor of `partition_type`.

Returns

- *list of membership* – list containing for each slice the membership vector.
- *float* – Improvement in quality of combined partitions, see `Optimiser.optimise_partition_multiplex()`.

See also:

`time_slices_to_layers()`, `slices_to_layers()`

Examples

```
>>> n = 100
>>> G_1 = ig.Graph.Lattice([n], 1)
>>> G_1.vs['id'] = range(n)
>>> G_2 = ig.Graph.Lattice([n], 1)
>>> G_2.vs['id'] = range(n)
>>> membership, improvement = la.find_partition_temporal([G_1, G_2],
...                                                      la.
↪ModularityVertexPartition,
...                                                      interslice_weight=1)
```

`leidenalg.`**`slices_to_layers`**`(`*G_coupling*, *slice_attr='slice'*, *vertex_id_attr='id'*, *edge_type_attr='type'*, *weight_attr='weight'*`)`

Convert a coupling graph of slices to layers of graphs.

This function converts a graph of slices to layers so that they can be used with this package.

This function assumes that the slices are represented by nodes in `G_coupling`, and stored in the attribute `slice_attr`. In other words, `G_coupling.vs[slice_attr]` should contain `ig.Graph` s . The slices will be converted to layers, and nodes in different slices will be coupled if the two slices are connected in `G_coupling`. Nodes in two connected slices are identified on the basis of the `vertex_id_attr`, i.e. if two nodes in two connected slices have an identical value of the `vertex_id_attr` they will be coupled. The `vertex_id_attr` should hence be unique in each slice. Each node in the resulting layer graphs will contain two vertex attributes with the name of `slice_attr` and `vertex_id_attr` that refer respectively to the slice and id of the node.

The weight of the coupling is determined by the weight of this link in `G_coupling`, as determined by the `weight_attr`.

Parameters

- **`G_coupling`** (`ig.Graph`) – The graph connecting the different slices.
- **`slice_attr`** (*`string`*) – The vertex attribute which contains the slices.

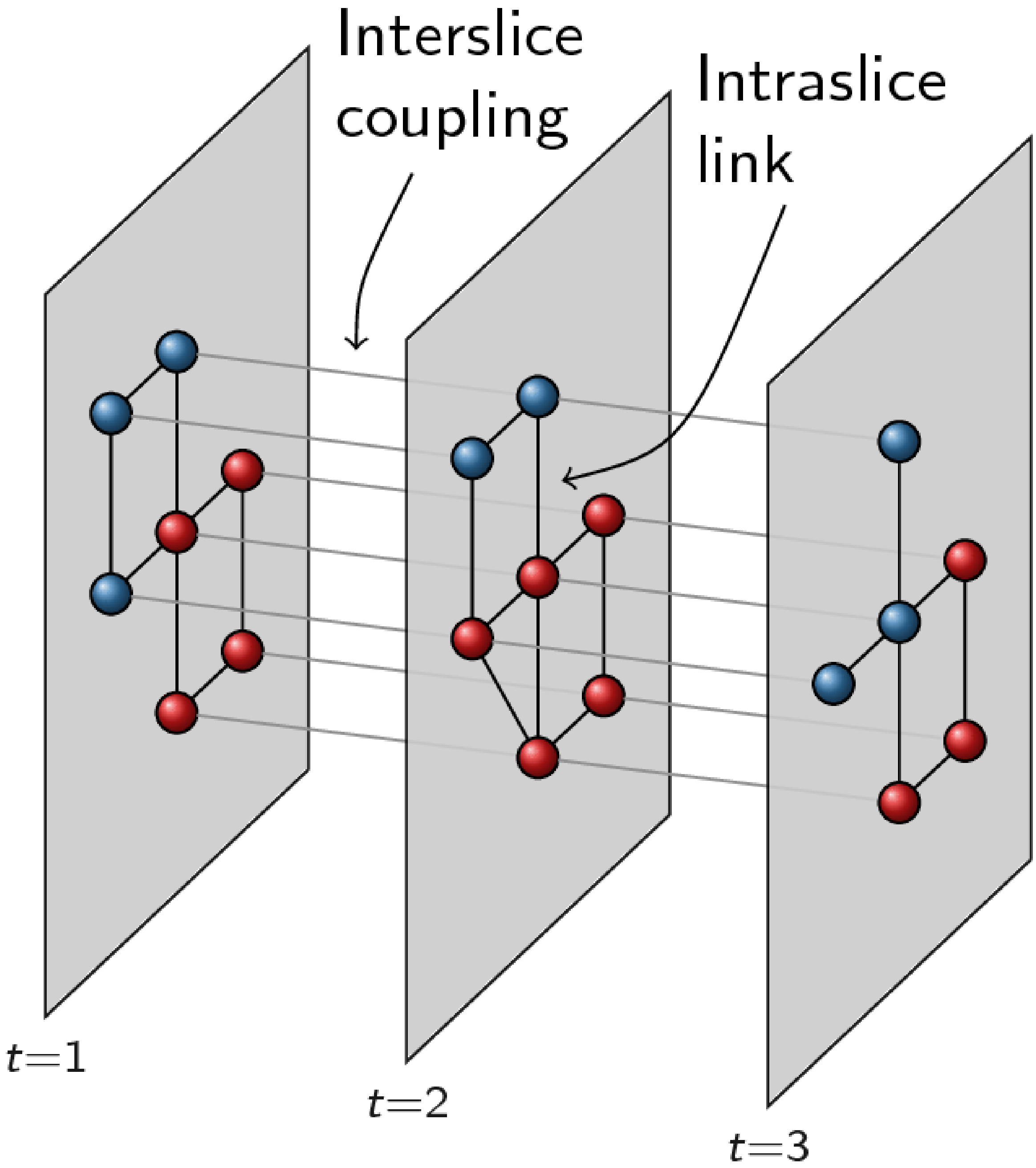
- **`vertex_id_attr`** (*string*) – The vertex attribute which is used to identify whether two nodes in two slices represent the same node, and hence, should be coupled.
- **`edge_type_attr`** (*string*) – The edge attribute to use for indicating the type of link (`interslice` or `intraslice`).
- **`weight_attr`** (*string*) – The edge attribute used to indicate the (coupling) weight.

Returns

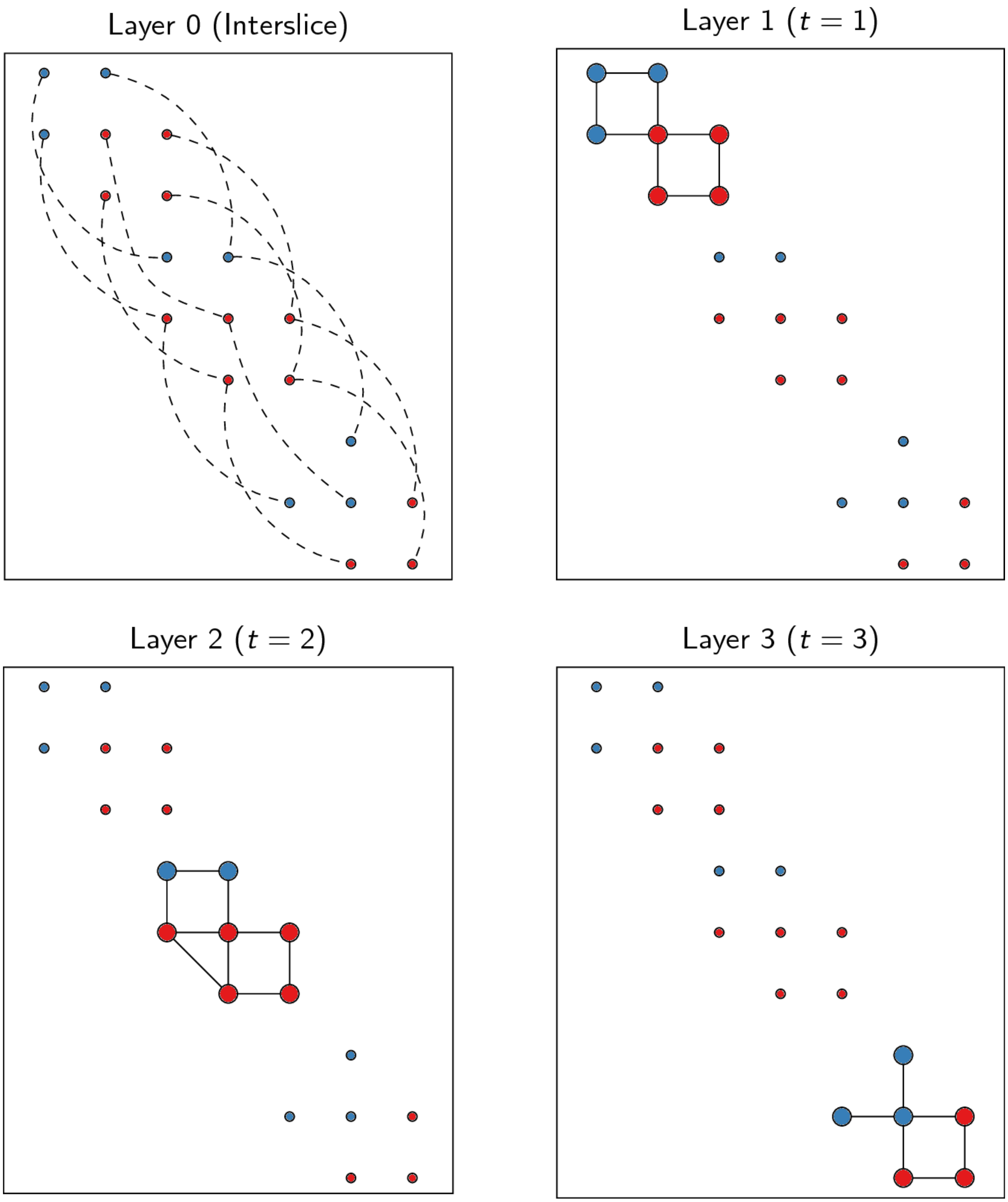
- **G_layers** (list of `ig.Graph`) – A list of slices converted to layers.
- **G_interslice** (`ig.Graph`) – The interslice coupling layer.
- **G** (`ig.Graph`) – The complete graph containing all layers and interslice couplings.

Notes

The distinction between slices and layers is not easy to grasp. Slices in this context refer to graphs that somehow represents different aspects of a network. The simplest example is probably slices that represents time: there are different snapshots network across time, and each snapshot is considered a slice. Some nodes may drop out of the network over time, while others enter the network. Edges may change over time, or the weight of the links may change over time. This is just the simplest example of a slice, and there may be different, more complex possibilities. Below an example with three time slices:



Now in order to optimise partitions across these different slices, we represent them slightly differently, namely as layers. The idea of layers is that all graphs always are defined on the same set of nodes, and that only the links differ for different layers. We thus create new nodes as combinations of original nodes and slices. For example, if node 1 existed in both slice 1 and in slice 2, we will thus create two nodes to build the layers: a node 1-1 and a node 1-2. Additionally, if the slices are connected in the slice graph, the two nodes would also be connected, so there would be a linke between node 1-1 and 1-2. Different slices will then correspond to different layers: each layer only contains the link for that particular slice. In addition, for methods such as *CPMVertexPartition*, so-called `node_sizes` are required, and for them to properly function, they should be set to 1 only for nodes of a layer that represent nodes of the corresponding slice and 0 for the other nodes (which is handled appropriately in this function, and stored in the vertex attribute `node_size`). Additionally, `node_sizes` should be set to 0 for the interslice coupling layer. We thus obtain equally many layers as we have slices, and we need one more layer for representing the interslice couplings. For the example provided above, we thus obtain the following:


Layer 0 (Interslice)
Layer 1 ($t = 1$)
Layer 2 ($t = 2$)
Layer 3 ($t = 3$)

The idea of doing community detection with slices is further detailed in [1].

References

See also:

find_partition_temporal(), *time_slices_to_layers()*

`leidenalg.`**`time_slices_to_layers`**(*graphs*, *interslice_weight=1*, *slice_attr='slice'*, *vertex_id_attr='id'*, *edge_type_attr='type'*, *weight_attr='weight'*)

Convert time slices to layer graphs.

Each graph is considered to represent a time slice. This function simply connects all the consecutive slices (i.e. the slice graph) with an `interslice_weight`. The further conversion is then delegated to *slices_to_layers()*, which also provides further details.

See also:

find_partition_temporal(), *slices_to_layers()*

6.1.2 Optimiser

`class leidenalg.`**`Optimiser`**

Bases: `object`

Class for doing community detection using the Leiden algorithm.

The Leiden algorithm [1] derives from the Louvain algorithm [2]. The Louvain algorithm has an elegant formulation. It consists of two phases: (1) move nodes between communities; (2) aggregate the graph. It then repeats these phases on the aggregate graph. The Leiden algorithm consists of three phases: (1) move nodes; (2) refine communities; (3) aggregate the graph based on the refinement. The Louvain algorithm can lead to arbitrarily badly connected communities, whereas the Leiden algorithm guarantees communities are well-connected. In fact, it converges towards a partition in which all subsets of all communities are locally optimally assigned. Finally, the Leiden algorithm is also much faster, because it relies on a fast local move routine.

There is one, rather technical, difference with the original Leiden algorithm. This implementation provides a general optimisation routine for any quality function. There is one aspect of the original Leiden algorithm that cannot be translated well in this framework: when merging subcommunities in the refinement procedure, it does not consider whether they are sufficiently well connected to the rest of the community. This implementation therefore does not guarantee subpartition γ-density. However, all other guarantees still hold:

After each iteration

1. γ-separation
2. γ-connectivity

After a stable iteration

3. Node optimality
4. Some subsets are locally optimally assigned

Asymptotically

5. Uniform γ-density
6. Subset optimality

The optimiser class provides a number of different methods for optimising a given partition. The overall optimisation procedure *optimise_partition()* calls either *move_nodes()* or *merge_nodes()* (which is controlled by *optimise_routine*) then aggregates the graph and repeats the same procedure. Possible, indicated by *refine_partition* the partition is refined before aggregating, meaning that subsets of communities are considered for moving around. Which routine is used for the refinement is indicated by `refine_routine`. For calculating the actual improvement of moving a node (corresponding a subset of nodes in the aggregate graph), the code relies on `diff_move()` which provides different values for different methods (e.g. modularity or CPM). The default settings are consistent with the Leiden algorithm. By not doing the refinement, you essentially get the Louvain algorithm with a fast local move. Finally, the Optimiser class provides a routine to construct a `resolution_profile()` on a resolution parameter.

References

Create a new Optimiser object

property consider_comms

Determine how alternative communities are considered for moving a node for *optimising* a partition.

Nodes will only move to alternative communities that improve the given quality function.

Notes

This attribute should be set to one of the following values

- `leidenalg.ALL_NEIGH_COMMS` Consider all neighbouring communities for moving.
- `leidenalg.ALL_COMMS` Consider all communities for moving. This is especially useful in the case of negative links, in which case it may be better to move a node to a non-neighbouring community.
- `leidenalg.RAND_NEIGH_COMM` Consider a random neighbour community for moving. The probability to choose a community is proportional to the number of neighbours a node has in that community.
- `leidenalg.RAND_COMM` Consider a random community for moving. The probability to choose a community is proportional to the number of nodes in that community.

property consider_empty_community

if `True` consider also moving nodes to an empty community (default).

Type

boolean

property max_comm_size

Constrain the maximal community size.

By default (zero), communities can be of any size. If this is set to a positive integer value, then communities will be constrained to be at most this total size.

merge_nodes(*partition*, *is_membership_fixed=None*, *consider_comms=None*)

Merge nodes for *optimising* the partition.

Parameters

- **partition** – The partition for which to merge nodes.
- **is_membership_fixed** (*list of bools or None*) – Boolean list of nodes that are not allowed to change community. The length of this list must be equal to the number of nodes. By default (None) all nodes can change community during the optimization.
- **consider_comms** – If `None` uses `consider_comms`, but can be set to something else.

Returns

Improvement in quality function.

Return type
float

Notes

This function loop over all nodes once and tries to merge them with another community. Merging in this case implies that a node will never be removed from a community, only merged with other communities.

See also:

`Optimiser.move_nodes()`, *`Optimiser.merge_nodes_constrained()`*

Examples

```
>>> G = ig.Graph.Famous('Zachary')
>>> optimiser = la.Optimiser()
>>> partition = la.ModularityVertexPartition(G)
>>> diff = optimiser.merge_nodes(partition)
```

`merge_nodes_constrained`(*partition*, *constrained_partition*, *consider_comms=None*)

Merge nodes for *refining* the partition.

Parameters

- **`partition`** – The partition for which to merge nodes.
- **`constrained_partition`** – The partition within which we may merge nodes.
- **`consider_comms`** – If `None` uses *`refine_consider_comms`*, but can be set to something else.

Returns
Improvement in quality function.

Return type
float

Notes

The idea is constrain the merging of nodes to another partition. In other words, if there is a partition `P` which we want to refine, we can then initialize a new singleton partition, and move nodes in that partition constrained to `P`.

See also:

`Optimiser.move_nodes_constrained()`, *`Optimiser.merge_nodes()`*

Examples

```
>>> G = ig.Graph.Famous('Zachary')
>>> optimiser = la.Optimiser()
>>> partition = la.ModularityVertexPartition(G)
>>> diff = optimiser.optimise_partition(partition)
>>> refine_partition = la.ModularityVertexPartition(G)
>>> diff = optimiser.move_nodes_constrained(refine_partition, partition)
```

`move_nodes`(*partition*, *is_membership_fixed=None*, *consider_comms=None*)

Move nodes to alternative communities for *optimising* the partition.

Parameters

- **`partition`** – The partition for which to move nodes.
- **`is_membership_fixed`** (*`list of bools or None`*) – Boolean list of nodes that are not allowed to change community. The length of this list must be equal to the number of nodes. By default (None) all nodes can change community during the optimization.
- **`consider_comms`** – If `None` uses *`consider_comms`*, but can be set to something else.

Returns

Improvement in quality function.

Return type

float

Notes

When moving nodes, the function loops over nodes and considers moving the node to an alternative community. Which community depends on `consider_comms`. The function terminates when no more nodes can be moved to an alternative community.

See also:

`Optimiser.move_nodes_constrained()`, *`Optimiser.merge_nodes()`*

Examples

```
>>> G = ig.Graph.Famous('Zachary')
>>> optimiser = la.Optimiser()
>>> partition = la.ModularityVertexPartition(G)
>>> diff = optimiser.move_nodes(partition)
```

`move_nodes_constrained`(*partition*, *constrained_partition*, *consider_comms=None*)

Move nodes to alternative communities for *refining* the partition.

Parameters

- **`partition`** – The partition for which to move nodes.
- **`constrained_partition`** – The partition within which we may move nodes.
- **`consider_comms`** – If `None` uses *`refine_consider_comms`*, but can be set to something else.

Returns
Improvement in quality function.

Return type
float

Notes

The idea is constrain the movement of nodes to alternative communities to another partition. In other words, if there is a partition `P` which we want to refine, we can then initialize a new singleton partition, and move nodes in that partition constrained to `P`.

See also:

`Optimiser.move_nodes()`, *`Optimiser.merge_nodes_constrained()`*

Examples

```
>>> G = ig.Graph.Famous('Zachary')
>>> optimiser = la.Optimiser()
>>> partition = la.ModularityVertexPartition(G)
>>> diff = optimiser.optimise_partition(partition)
>>> refine_partition = la.ModularityVertexPartition(G)
>>> diff = optimiser.move_nodes_constrained(refine_partition, partition)
```

`optimise_partition`(*partition*, *n_iterations=2*, *is_membership_fixed=None*)

Optimise the given partition.

Parameters

- **`partition`** – The *`MutableVertexPartition`* to optimise.
- **`n_iterations`** (*`int`*) – Number of iterations to run the Leiden algorithm. By default, 2 iterations are run. If the number of iterations is negative, the Leiden algorithm is run until an iteration in which there was no improvement.
- **`is_membership_fixed`** (*`list of bools or None`*) – Boolean list of nodes that are not allowed to change community. The length of this list must be equal to the number of nodes. By default (None) all nodes can change community during the optimization.

Returns
Improvement in quality function.

Return type
float

Examples

```
>>> G = ig.Graph.Famous('Zachary')
>>> optimiser = la.Optimiser()
>>> partition = la.ModularityVertexPartition(G)
>>> diff = optimiser.optimise_partition(partition)
```

or, fixing the membership of some nodes:

```
>>> is_membership_fixed = [False for v in G.vs]
>>> is_membership_fixed[4] = True
>>> is_membership_fixed[6] = True
>>> diff = optimiser.optimise_partition(partition, is_membership_fixed=is_
↪membership_fixed)
```

optimise_partition_multiplex(*partitions*, *layer_weights=None*, *n_iterations=2*, *is_membership_fixed=None*)

Optimise the given partitions simultaneously.

Parameters

- **partitions** – List of *MutableVertexPartition* layers to optimise.
- **layer_weights** – List of weights of layers.
- **is_membership_fixed** (*list of bools or None*) – Boolean list of nodes that are not allowed to change community. The length of this list must be equal to the number of nodes. By default (None) all nodes can change community during the optimization.
- **n_iterations** (*int*) – Number of iterations to run the Leiden algorithm. By default, 2 iterations are run. If the number of iterations is negative, the Leiden algorithm is run until an iteration in which there was no improvement.

Returns

Improvement in quality of combined partitions, see *Notes*.

Return type

float

Notes

This method assumes that the partitions are defined for graphs with the same vertices. The connections between the vertices may be different, but the vertices themselves should be identical. In other words, all vertices should have identical indices in all graphs (i.e. node *i* is assumed to be the same node in all graphs). The quality of the overall partition is simply the sum of the individual qualities for the various partitions, weighted by the layer_weight. If we denote by Q_k the quality of layer k and the weight by λ_k, the overall quality is then

$$Q = \sum_k \lambda_k Q_k.$$

This is particularly useful for graphs containing negative links. When separating the graph in two graphs, the one containing only the positive links, and the other only the negative link, by supplying a negative weight to the latter layer, we try to find relatively many positive links within a community and relatively many negative links between communities. Note that in this case it may be better to assign a node to a community to which it is not connected so that *consider_comms* may be better set to `leidenalg.ALL_COMMS`.

Besides multiplex graphs where each node is assumed to have a single community, it is also useful in the case of for example multiple time slices, or in situations where nodes can have different communities in different slices. The package includes some special helper functions for using *optimise_partition_multiplex()* in such cases, where there is a conversion required from (time) slices to layers suitable for use in this function.

See also:

slices_to_layers(), *time_slices_to_layers()*, *find_partition_multiplex()*, *find_partition_temporal()*

Examples

```
>>> G_pos = ig.Graph.SBM(100, pref_matrix=[[0.5, 0.1], [0.1, 0.5]], block_
↪sizes=[50, 50])
>>> G_neg = ig.Graph.SBM(100, pref_matrix=[[0.1, 0.5], [0.5, 0.1]], block_
↪sizes=[50, 50])
>>> optimiser = la.Optimiser()
>>> partition_pos = la.ModularityVertexPartition(G_pos)
>>> partition_neg = la.ModularityVertexPartition(G_neg)
>>> diff = optimiser.optimise_partition_multiplex(
...                     partitions=[partition_pos, partition_neg],
...                     layer_weights=[1,-1])
```

property optimise_routine

Determine the routine to use for *optimising* a partition.

Notes

This attribute should be set to one of the following values

- `leidenalg.MOVE_NODES` Use *move_nodes()*.
- `leidenalg.MERGE_NODES` Use *merge_nodes()*.

property refine_consider_comms

Determine how alternative communities are considered for moving a node when *refining* a partition.

Nodes will only move to alternative communities that improve the given quality function.

Notes

This attribute should be set to one of the following values

- `leidenalg.ALL_NEIGH_COMMS` Consider all neighbouring communities for moving.
- `leidenalg.ALL_COMMS` Consider all communities for moving. This is especially useful in the case of negative links, in which case it may be better to move a node to a non-neighbouring community.
- `leidenalg.RAND_NEIGH_COMM` Consider a random neighbour community for moving. The probability to choose a community is proportional to the number of neighbours a node has in that community.
- `leidenalg.RAND_COMM` Consider a random community for moving. The probability to choose a community is proportional to the number of nodes in that community.

property refine_partition

if `True` refine partition before aggregation.

Type

boolean

property refine_routine

Determine the routine to use for *refining* a partition.

Notes

This attribute should be set to one of the following values

- `leidenalg.MOVE_NODES` Use *`move_nodes()`*.
- `leidenalg.MERGE_NODES` Use *`merge_nodes()`*.

`resolution_profile`(*graph*, *partition_type*, *resolution_range*, *weights=None*, *bisect_func=<function Optimiser.<lambda>>*, *min_diff_bisect_value=1*, *min_diff_resolution=0.001*, *linear_bisection=False*, *number_iterations=1*, **kwargs)

Use bisectioning on the resolution parameter in order to construct a resolution profile.

Parameters

- **`graph`** – The graph for which to construct a resolution profile.
- **`partition_type`** – The type of *`MutableVertexPartition`* used to find a partition (must support resolution parameters obviously).
- **`resolution_range`** – The range of resolution values that we would like to scan.
- **`weights`** – If provided, indicates the edge attribute to use as a weight.
- **`bisect_func`** – The function used for bisectioning. For the methods currently implemented, this should usually not be altered.
- **`min_diff_bisect_value`** – The difference in the value returned by the bisect_func below which the bisectioning stops (i.e. by default, a difference of a single edge does not trigger further bisectioning).
- **`min_diff_resolution`** – The difference in resolution below which the bisectioning stops. For positive differences, the logarithmic difference is used by default, i.e. `diff = log(res_1) - log(res_2) = log(res_1/res_2)`, for which `diff > min_diff_resolution` to continue bisectioning. Set the linear_bisection to true in order to use only linear bisectioning (in the case of negative resolution parameters for example, which can happen with negative weights).
- **`linear_bisection`** – Whether the bisectioning will be done on a linear or on a logarithmic basis (if possible).
- **`number_iterations`** – Indicates the number of iterations of the algorithm to run. If negative (or zero) the algorithm is run until a stable iteration.

Returns

A list of partitions for different resolutions.

Return type

list of *`MutableVertexPartition`*

Examples

```
>>> G = ig.Graph.Famous('Zachary')
>>> optimiser = la.Optimiser()
>>> profile = optimiser.resolution_profile(G, la.CPMVertexPartition,
...                                        resolution_range=(0,1))
```

`set_rng_seed`(*value*)

Set the random seed for the random number generator.

Parameters

`value` – The integer seed used in the random number generator

6.1.3 MutableVertexPartition

`class leidenalg.VertexPartition.`**`MutableVertexPartition`**(*graph*, *initial_membership=None*)

Bases: `VertexClustering`

Contains a partition of a graph, derives from `ig.VertexClustering`. Please see the documentation of `ig.VertexClustering` for more details about its functionality.

This class contains the basic implementation for optimising a partition. Specifically, it implements all the administration necessary to keep track of the partition from various points of view. Internally, it keeps track of the number of internal edges (or total weight), the size of the communities, the total incoming degree (or weight) for a community, et cetera.

In order to keep the administration up-to-date, all changes in a partition should be done through `move_node()` or `set_membership()`. The first moves a node from one community to another, and updates the administration. The latter simply updates the membership vector and updates the administration.

The basic idea is that `diff_move()` computes the difference in the quality function if we would call `move_node()` for the same move. These functions are overridden in any derived classes to provide an actual implementation. These functions are used by `Optimiser` to optimise the partition.

> **Warning:** This base class should never be used in practice, since only derived classes provide an actual implementation.

Parameters

- **`graph`** – The `ig.Graph` on which this partition is defined.
- **`membership`** – The membership vector of this partition. `Membership[i] = c` implies that node `i` is in community `c`. If `None`, it is initialised with a singleton partition community, i.e. `membership[i] = i`.

`classmethod` **`FromPartition`**(*partition*, ***kwargs*)

Create a new partition from an existing partition.

Parameters

- **`partition`** – The `MutableVertexPartition` to replicate.
- **`**kwargs`** – Any remaining keyword arguments will be passed on to the constructor of the new partition.

Notes

This may for example come in handy when determining the quality of a partition using a different method. Suppose that we already have a partition `p` and that we want to determine the Significance of that partition. We can then simply use

```
>>> p = la.find_partition(ig.Graph.Famous('Zachary'),
...                       la.ModularityVertexPartition)
>>> sig = la.SignificanceVertexPartition.FromPartition(p).quality()
```

aggregate_partition(*membership_partition=None*)

Aggregate the graph according to the current partition and provide a default partition for it.

The aggregated graph can then be found as a parameter of the partition `partition.graph`.

Notes

This function contrasts to the function `cluster_graph` in igraph itself, which also provides the aggregate graph, but we may require setting the appropriate `resolution_parameter`, `weights` and `node_sizes`. In particular, this function also ensures that the quality defined on the aggregate partition is identical to the quality defined on the original partition.

Examples

```
>>> G = ig.Graph.Famous('Zachary')
>>> partition = la.find_partition(G, la.ModularityVertexPartition)
>>> aggregate_partition = partition.aggregate_partition(partition)
>>> aggregate_graph = aggregate_partition.graph
>>> aggregate_partition.quality() == partition.quality()
True
```

diff_move(*v*, *new_comm*)

Calculate the difference in the quality function if node `v` is moved to community `new_comm`.

Parameters

- **v** – The node to move.
- **new_comm** – The community to move to.

Returns

Difference in quality function.

Return type

float

Notes

The difference returned by diff_move should be equivalent to first determining the quality of the partition, then calling move_node, and then determining again the quality of the partition and looking at the difference. In other words

```
>>> partition = la.find_partition(ig.Graph.Famous('Zachary'),
...                               la.ModularityVertexPartition)
>>> diff = partition.diff_move(v=0, new_comm=0)
>>> q1 = partition.quality()
>>> partition.move_node(v=0, new_comm=0)
>>> q2 = partition.quality()
>>> round(diff, 10) == round(q2 - q1, 10)
True
```

Warning: Only derived classes provide actual implementations, the base class provides no implementation for this function.

from_coarse_partition(*partition*, *coarse_node=None*)

Update current partition according to coarser partition.

Parameters

- **partition** (`MutableVertexPartition`) – The coarser partition used to update the current partition.
- **coarse_node** (*list of int*) – The coarser node which represent the current node in the partition.

Notes

This function is to be used to determine the correct partition for an aggregated graph. In particular, suppose we move nodes and then get an aggregate graph.

```
>>> diff = optimiser.move_nodes(partition)
>>> aggregate_partition = partition.aggregate_partition()
```

Now we also move nodes in the aggregate partition

```
>>> diff = optimiser.move_nodes(aggregate_partition)
```

Now we improved the quality function of `aggregate_partition`, but this is not yet reflected in the original `partition`. We can thus call

```
>>> partition.from_coarse_partition(aggregate_partition)
```

so that `partition` now reflects the changes made to `aggregate_partition`.

The `coarse_node` can be used it the `aggregate_partition` is not defined based on the membership of this partition. In particular the membership of this partition is defined as follows:

```
>>> for v in G.vs:
...   partition.membership[v] = aggregate_partition.membership[coarse_node[v]]
```

If `coarse_node` is `None` it is assumed the coarse node was defined based on the membership of the current partition, so that

```
>>> for v in G.vs:
...   partition.membership[v] = aggregate_partition.membership[partition.
↪membership[v]]
```

This can be useful when the aggregate partition is defined on a more refined partition.

move_node(*v*, *new_comm*)

Move node `v` to community `new_comm`.

Parameters

- **v** – Node to move.
- **new_comm** – Community to move to.

Examples

```
>>> G = ig.Graph.Famous('Zachary')
>>> partition = la.ModularityVertexPartition(G)
>>> partition.move_node(0, 1)
```

`quality()`

The current quality of the partition.

`renumber_communities()`

Renumber the communities so that they are numbered in decreasing size.

Notes

The sort is not necessarily stable.

`set_membership`(*membership*)

Set membership.

`total_possible_edges_in_all_comms()`

The total possible number of edges in all communities.

Notes

If we denote by n_c the number of nodes in community c, this is simply

$$\sum_c \binom{n_c}{2}$$

`total_weight_from_comm`(*comm*)

The total weight (i.e. number of edges) from a community.

Parameters

`comm` – Community

Notes

This includes all edges, also the ones that are internal to a community. Sometimes this is also referred to as the community (out)degree.

See also:

`total_weight_to_comm()`, `total_weight_in_comm()`, `total_weight_in_all_comms()`

`total_weight_in_all_comms()`

The total weight (i.e. number of edges) within all communities.

Notes

This should be equal to simply the sum of `total_weight_in_comm` for all communities.

See also:

`total_weight_to_comm()`, `total_weight_from_comm()`, `total_weight_in_comm()`

`total_weight_in_comm`(*comm*)

The total weight (i.e. number of edges) within a community.

Parameters
`comm` – Community

See also:

`total_weight_to_comm()`, `total_weight_from_comm()`, `total_weight_in_all_comms()`

`total_weight_to_comm`(*comm*)

The total weight (i.e. number of edges) to a community.

Parameters
`comm` – Community

Notes

This includes all edges, also the ones that are internal to a community. Sometimes this is also referred to as the community (in)degree.

See also:

`total_weight_from_comm()`, `total_weight_in_comm()`, `total_weight_in_all_comms()`

`weight_from_comm`(*v*, *comm*)

The total number of edges (or sum of weights) to node `v` from community `comm`.

See also:

`weight_to_comm()`

`weight_to_comm`(*v*, *comm*)

The total number of edges (or sum of weights) from node `v` to community `comm`.

See also:

`weight_from_comm()`

6.1.4 ModularityVertexPartition

class `leidenalg.`**`ModularityVertexPartition`**(*graph*, *initial_membership=None*, *weights=None*)

Bases: *`MutableVertexPartition`*

Implements modularity. This quality function is well-defined only for positive edge weights.

Notes

The quality function is

$$Q = \frac{1}{2m} \sum_{ij} \left(A_{ij} - \frac{k_i k_j}{2m}\right) \delta(\sigma_i, \sigma_j)$$

where A is the adjacency matrix, k_i is the (weighted) degree of node i, m is the total number of edges (or total edge weight), σ_i denotes the community of node i and $\delta(\sigma_i, \sigma_j) = 1$ if $\sigma_i = \sigma_j$ and *0* otherwise.

This can alternatively be formulated as a sum over communities:

$$Q = \frac{1}{2m} \sum_{c} \left(m_c - \frac{K_c^2}{4m}\right)$$

where m_c is the number of internal edges (or total internal edge weight) of community c and $K_c = \sum_{i \mid \sigma_i = c} k_i$ is the total (weighted) degree of nodes in community c.

Note that for directed graphs a slightly different formulation is used, as proposed by Leicht and Newman [2]:

$$Q = \frac{1}{m} \sum_{ij} \left(A_{ij} - \frac{k_i^\mathrm{out} k_j^\mathrm{in}}{m}\right) \delta(\sigma_i, \sigma_j),$$

where k_i^out and k_i^in refers to respectively the outdegree and indegree of node i, and A_{ij} refers to an edge from i to j.

References

Parameters

- **graph** (`ig.Graph`) – Graph to define the partition on.
- **initial_membership** (*list of int*) – Initial membership for the partition. If `None` then defaults to a singleton partition.
- **weights** (*list of double, or edge attribute*) – Weights of edges. Can be either an iterable or an edge attribute.

6.1.5 RBConfigurationVertexPartition

class `leidenalg.`**`RBConfigurationVertexPartition`**(*graph*, *initial_membership=None*, *weights=None*, *resolution_parameter=1.0*)

Bases: `LinearResolutionParameterVertexPartition`

Implements Reichardt and Bornholdt’s Potts model with a configuration null model. This quality function is well-defined only for positive edge weights. This quality function uses a linear resolution parameter.

Notes

The quality function is

$$Q = \sum_{ij} \left(A_{ij} - \gamma \frac{k_i k_j}{2m} \right) \delta(\sigma_i, \sigma_j)$$

where A is the adjacency matrix, k_i is the (weighted) degree of node i, m is the total number of edges (or total edge weight), σ_i denotes the community of node i and $\delta(\sigma_i, \sigma_j) = 1$ if $\sigma_i = \sigma_j$ and *0* otherwise.

This can alternatively be formulated as a sum over communities:

$$Q = \sum_{c} \left(m_c - \gamma \frac{K_c^2}{4m} \right)$$

where m_c is the number of internal edges (or total internal edge weight) of community c and $K_c = \sum_{i \mid \sigma_i = c} k_i$ is the total (weighted) degree of nodes in community c.

Note that for directed graphs a slightly different formulation is used, as proposed by Leicht and Newman [2]:

$$Q = \sum_{ij} \left(A_{ij} - \gamma \frac{k_i^\mathrm{out} k_j^\mathrm{in}}{m} \right) \delta(\sigma_i, \sigma_j),$$

where k_i^out and k_i^in refers to respectively the outdegree and indegree of node i, and A_{ij} refers to an edge from i to j.

Note that this is the same as `ModularityVertexPartition` when setting $\gamma = 1$ and normalising by $2m$, or m for directed graphs.

References

Parameters

- **graph** (`ig.Graph`) – Graph to define the partition on.
- **initial_membership** (*list of int*) – Initial membership for the partition. If `None` then defaults to a singleton partition.
- **weights** (*list of double, or edge attribute*) – Weights of edges. Can be either an iterable or an edge attribute.
- **resolution_parameter** (*double*) – Resolution parameter.

property resolution_parameter

Resolution parameter.

6.1.6 RBERVertexPartition

class `leidenalg.`**RBERVertexPartition**(*graph*, *initial_membership=None*, *weights=None*, *node_sizes=None*, *resolution_parameter=1.0*)

Bases: `LinearResolutionParameterVertexPartition`

Implements Reichardt and Bornholdt's Potts model with an Erdős-Rényi null model. This quality function is well-defined only for positive edge weights. This quality function uses a linear resolution parameter.

Notes

The quality function is

$$Q = \sum_{ij} \left(A_{ij} - \gamma p\right) \delta(\sigma_i, \sigma_j)$$

where A is the adjacency matrix,

$$p = \frac{m}{\binom{n}{2}}$$

is the overall density of the graph, σ_i denotes the community of node i, $\delta(\sigma_i, \sigma_j) = 1$ if $\sigma_i = \sigma_j$ and *0* otherwise, and, finally γ is a resolution parameter.

This can alternatively be formulated as a sum over communities:

$$Q = \sum_{c} \left[m_c - \gamma p \binom{n_c}{2}\right]$$

where m_c is the number of internal edges of community c and n_c the number of nodes in community c.

References

Parameters

- **graph** (`ig.Graph`) – Graph to define the partition on.
- **initial_membership** (*list of int*) – Initial membership for the partition. If `None` then defaults to a singleton partition.
- **weights** (*list of double, or edge attribute*) – Weights of edges. Can be either an iterable or an edge attribute.
- **node_sizes** (*list of int, or vertex attribute*) – The quality function takes into account the size of a community, which is defined as the sum over the sizes of each individual node. By default, the node sizes are set to 1, meaning that the size of a community equals the number of nodes of a community. If a node already represents an aggregation, this could be reflect in its node size.
- **resolution_parameter** (*double*) – Resolution parameter.

6.1.7 CPMVertexPartition

class `leidenalg.`**`CPMVertexPartition`**(*graph*, *initial_membership=None*, *weights=None*, *node_sizes=None*, *resolution_parameter=1.0*, *correct_self_loops=None*)

Bases: `LinearResolutionParameterVertexPartition`

Implements the Constant Potts Model (CPM). This quality function is well-defined for both positive and negative edge weights. This quality function uses a linear resolution parameter.

Notes

The Constant Potts Model (CPM) quality function is

$$Q = \sum_{ij} (A_{ij} - \gamma) \delta(\sigma_i, \sigma_j)$$

where A is the adjacency matrix, σ_i denotes the community of node i, $\delta(\sigma_i, \sigma_j) = 1$ if $\sigma_i = \sigma_j$ and *0* otherwise, and, finally γ is a resolution parameter.

This can alternatively be formulated as a sum over communities:

$$Q = \sum_{c} \left[m_c - \gamma \binom{n_c}{2} \right]$$

where m_c is the number of internal edges of community c and n_c the number of nodes in community c.

The resolution parameter γ for this functions has a particularly simple interpretation. The internal density of communities

$$p_c = \frac{m_c}{\binom{n_c}{2}} \geq \gamma$$

is higher than γ, while the external density

$$p_{cd} = \frac{m_{cd}}{n_c n_d} \leq \gamma$$

is lower than γ. In other words, choosing a particular γ corresponds to choosing to find communities of a particular density, and as such defines communities. Finally, the definition of a community is in a sense independent of the actual graph, which is not the case for any of the other methods (see the reference for more detail).

References

Parameters

- **graph** (`ig.Graph`) – Graph to define the partition on.
- **initial_membership** (*list of int*) – Initial membership for the partition. If `None` then defaults to a singleton partition.
- **weights** (*list of double, or edge attribute*) – Weights of edges. Can be either an iterable or an edge attribute.
- **node_sizes** (*list of int, or vertex attribute*) – The quality function takes into account the size of a community, which is defined as the sum over the sizes of each individual node. By default, the node sizes are set to 1, meaning that the size of a community equals the number of nodes of a community. If a node already represents an aggregation, this could be reflect in its node size.
- **resolution_parameter** (*double*) – Resolution parameter.

classmethod Bipartite(*graph*, *resolution_parameter_01*, *resolution_parameter_0=0*, *resolution_parameter_1=0*, *degree_as_node_size=False*, *types='type'*, ***kwargs*)

Create three layers for bipartite partitions.

This creates three layers for bipartite partition necessary for detecting communities in bipartite networks. These three layers should be passed to `Optimiser.optimise_partition_multiplex()` with `layer_weights=[1,-1,-1]`. See *Notes* for more details.

Parameters

- **`graph`** (`ig.Graph`) – Graph to define the bipartite partitions on.
- **`resolution_parameter_01`** (*`double`*) – Resolution parameter for in between two classes.
- **`resolution_parameter_0`** (*`double`*) – Resolution parameter for class 0.
- **`resolution_parameter_1`** (*`double`*) – Resolution parameter for class 1.
- **`degree_as_node_size`** (*`boolean`*) – If `True` use degree as node size instead of 1, to mimic modularity, see *Notes*.
- **`types`** (*`vertex attribute or list`*) – Indicator of the class for each vertex. If not 0, 1, it is automatically converted.
- **`**kwargs`** – Additional arguments passed on to default constructor of *`CPMVertexPartition`*.

Returns

- `ig.CPMVertexPartition` – partition containing the bipartite graph and correct node sizes.
- `ig.CPMVertexPartition` – partition for type 0, containing the correct node sizes for type 0.
- `ig.CPMVertexPartition` – partition for type 1, containing the correct node sizes for type 1.

Notes

For bipartite networks, we would like to be able to set three different resolution parameters: one for within each class γ_0, γ_1, and one for the links between classes, γ_{01}. Then the formulation would be

$$Q = \sum_{ij} [A_{ij} - (\gamma_0 \delta(s_i, 0) + \gamma_1 \delta(s_i, 1)) \delta(s_i, s_j) - \gamma_{01} (1 - \delta(s_i, s_j))] \delta(\sigma_i, \sigma_j)$$

In terms of communities this is

$$Q = \sum_c (e_c - \gamma_{01} 2 n_c(0) n_c(1) - \gamma_0 n^2_c(0) - \gamma_1 n^2_c(1))$$

where $n_c(0)$ is the number of nodes in community c of class 0 (and similarly for 1) and e_c is the number of edges within community c. We denote by $n_c = n_c(0) + n_c(1)$ the total number of nodes in community c.

We achieve this by creating three layers : (1) all nodes have `node_size = 1` and all relevant links; (2) only nodes of class 0 have `node_size = 1` and no links; (3) only nodes of class 1 have `node_size = 1` and no links. If we add the first with resolution parameter γ_{01}, and the others with resolution parameters $\gamma_{01} - \gamma_0$ and $\gamma_{01} - \gamma_1$, but the latter two with a layer weight of -1 while the first layer has layer weight 1, we obtain the following:

$$\begin{aligned}
Q &= \sum_c (e_c - \gamma_{01} n_c^2) - \sum_c (-(\gamma_{01} - \gamma_0) n_c(0)^2) - \sum_c (-(\gamma_{01} - \gamma_1) n_c(1)^2) \\
&= \sum_c [e_c - \gamma_{01} 2 n_c(0) n_c(1) - \gamma_{01} n_c(0)^2 - \gamma_{01} n_c(1)^2) + (\gamma_{01} - \gamma_0) n_c(0)^2 + (\gamma_{01} - \gamma_1) n_c(1)^2] \\
&= \sum_c [e_c - \gamma_{01} 2 n_c(0) n_c(1) - \gamma_0 n_c(0)^2 - \gamma_1 n_c(1)^2]
\end{aligned}$$

Although the derivation above is using n_c^2, implicitly assuming a direct graph with self-loops, similar derivations can be made for undirected graphs using $\binom{n_c}{2}$, but the notation is then somewhat more convoluted.

If we set node sizes equal to the degree, we get something similar to modularity, except that the resolution parameter should still be divided by $2m$. In particular, in general (i.e. not specifically for bipartite graph) if `node_sizes=G.degree()` we then obtain

$$Q = \sum_{ij} A_{ij} - \gamma k_i k_j$$

In the case of bipartite graphs something similar is obtained, but then correctly adapted (as long as the resolution parameter is also appropriately rescaled).

Note: This function is not suited for directed graphs in the case of using the degree as node sizes.

6.1.8 SignificanceVertexPartition

`class leidenalg.`**`SignificanceVertexPartition`**(*graph*, *initial_membership=None*, *node_sizes=None*)

Bases: *`MutableVertexPartition`*

Implements Significance. This quality function is well-defined only for unweighted graphs.

Notes

The quality function is

$$Q = \sum_c \binom{n_c}{2} D(p_c \parallel p)$$

where n_c is the number of nodes in community c,

$$p_c = \frac{m_c}{\binom{n_c}{2}},$$

is the density of community c,

$$p = \frac{m}{\binom{n}{2}}$$

is the overall density of the graph, and finally

$$D(x \parallel y) = x \ln \frac{x}{y} + (1 - x) \ln \frac{1 - x}{1 - y}$$

is the binary Kullback-Leibler divergence.

For directed graphs simply multiply the binomials by 2. The expected Significance in Erdos-Renyi graphs behaves roughly as $\frac{1}{2} n \ln n$ for both directed and undirected graphs in this formulation.

Warning: This method is not suitable for weighted graphs.

References

Parameters

- **`graph`** (`ig.Graph`) – Graph to define the partition on.
- **`initial_membership`** (*`list of int`*) – Initial membership for the partition. If `None` then defaults to a singleton partition.
- **`node_sizes`** (*`list of int, or vertex attribute`*) – The quality function takes into account the size of a community, which is defined as the sum over the sizes of each individual node. By default, the node sizes are set to 1, meaning that the size of a community equals the number of nodes of a community. If a node already represents an aggregation, this could be reflect in its node size.

6.1.9 SurpriseVertexPartition

`class leidenalg.`**`SurpriseVertexPartition`**(*graph*, *initial_membership=None*, *weights=None*, *node_sizes=None*)

Bases: *`MutableVertexPartition`*

Implements (asymptotic) Surprise. This quality function is well-defined only for positive edge weights.

Notes

The quality function is

$$Q = mD(q \parallel \langle q \rangle)$$

where m is the number of edges,

$$q = \frac{\sum_c m_c}{m},$$

is the fraction of internal edges,

$$\langle q \rangle = \frac{\sum_c \binom{n_c}{2}}{\binom{n}{2}}$$

is the expected fraction of internal edges, and finally

$$D(x \parallel y) = x \ln \frac{x}{y} + (1 - x) \ln \frac{1 - x}{1 - y}$$

is the binary Kullback-Leibler divergence.

For directed graphs we can multiplying the binomials by 2, and this leaves $\langle q \rangle$ unchanged, so that we can simply use the same formulation. For weighted graphs we can simply count the total internal weight instead of the total number of edges for q, while $\langle q \rangle$ remains unchanged.

References

Parameters

- **`graph`** (`ig.Graph`) – Graph to define the partition on.
- **`initial_membership`** (*`list of int`*) – Initial membership for the partition. If `None` then defaults to a singleton partition.
- **`weights`** (*`list of double, or edge attribute`*) – Weights of edges. Can be either an iterable or an edge attribute.
- **`node_sizes`** (*`list of int, or vertex attribute`*) – The quality function takes into account the size of a community, which is defined as the sum over the sizes of each individual node. By default, the node sizes are set to 1, meaning that the size of a community equals the number of nodes of a community. If a node already represents an aggregation, this could be reflect in its node size.

- genindex
- search

PYTHON MODULE INDEX

l

`leidenalg`, 23

INDEX

A

aggregate_partition() (*leidenalg.VertexPartition.MutableVertexPartition method*), 38

B

Bipartite() (*leidenalg.CPMVertexPartition class method*), 46

C

consider_comms (*leidenalg.Optimiser property*), 31
consider_empty_community (*leidenalg.Optimiser property*), 31
CPMVertexPartition (*class in leidenalg*), 45

D

diff_move() (*leidenalg.VertexPartition.MutableVertexPartition method*), 39

F

find_partition() (*in module leidenalg*), 23
find_partition_multiplex() (*in module leidenalg*), 24
find_partition_temporal() (*in module leidenalg*), 25
from_coarse_partition() (*leidenalg.VertexPartition.MutableVertexPartition method*), 40
FromPartition() (*leidenalg.VertexPartition.MutableVertexPartition class method*), 38

L

leidenalg
    module, 23

M

max_comm_size (*leidenalg.Optimiser property*), 31
merge_nodes() (*leidenalg.Optimiser method*), 31
merge_nodes_constrained() (*leidenalg.Optimiser method*), 32
ModularityVertexPartition (*class in leidenalg*), 42
module
    leidenalg, 23
move_node() (*leidenalg.VertexPartition.MutableVertexPartition method*), 40
move_nodes() (*leidenalg.Optimiser method*), 33
move_nodes_constrained() (*leidenalg.Optimiser method*), 33
MutableVertexPartition (*class in leidenalg.VertexPartition*), 38

O

optimise_partition() (*leidenalg.Optimiser method*), 34
optimise_partition_multiplex() (*leidenalg.Optimiser method*), 35
optimise_routine (*leidenalg.Optimiser property*), 36
Optimiser (*class in leidenalg*), 30

Q

quality() (*leidenalg.VertexPartition.MutableVertexPartition method*), 41

R

RBConfigurationVertexPartition (*class in leidenalg*), 43
RBERVertexPartition (*class in leidenalg*), 44
refine_consider_comms (*leidenalg.Optimiser property*), 36
refine_partition (*leidenalg.Optimiser property*), 36
refine_routine (*leidenalg.Optimiser property*), 36
renumber_communities() (*leidenalg.VertexPartition.MutableVertexPartition method*), 41
resolution_parameter (*leidenalg.RBConfigurationVertexPartition property*), 44
resolution_profile() (*leidenalg.Optimiser method*), 37

S

set_membership() (*leide-*

nalg.VertexPartition.MutableVertexPartition method), 41
`set_rng_seed()` (*leidenalg.Optimiser method*), 37
`SignificanceVertexPartition` (*class in leidenalg*), 48
`slices_to_layers()` (*in module leidenalg*), 26
`SurpriseVertexPartition` (*class in leidenalg*), 49

T

`time_slices_to_layers()` (*in module leidenalg*), 30
`total_possible_edges_in_all_comms()` (*leidenalg.VertexPartition.MutableVertexPartition method*), 41
`total_weight_from_comm()` (*leidenalg.VertexPartition.MutableVertexPartition method*), 41
`total_weight_in_all_comms()` (*leidenalg.VertexPartition.MutableVertexPartition method*), 41
`total_weight_in_comm()` (*leidenalg.VertexPartition.MutableVertexPartition method*), 42
`total_weight_to_comm()` (*leidenalg.VertexPartition.MutableVertexPartition method*), 42

W

`weight_from_comm()` (*leidenalg.VertexPartition.MutableVertexPartition method*), 42
`weight_to_comm()` (*leidenalg.VertexPartition.MutableVertexPartition method*), 42