tnetwork Documentation

Remy Cazabet

Mar 21, 2022

Contents

`tnetwork` is a Python software package to manipulate temporal networks.

Date	Python Versions	Main Author	GitHub	pypl
2022-03-21	3.x	Rémy Cazabet	Source	Distribution

CHAPTER 1

tnetwork Dev Team

Name	Contribution
Rémy Cazabet	Initial development

CHAPTER 2

Documentation

2.1 Installation

2.1.1 Quick install

Get `tnetwork` from the Python Package Index at pypl.

or install it with

```
pip install tnetwork
```

and an attempt will be made to find and install an appropriate version that matches your operating system and Python version.

You can install the development version with

```
pip install git://github.com/Yquetzal/tnetwork.git
```

2.1.2 Installing from source

You can install from source by downloading a source archive file (tar.gz or zip) or by checking out the source files from the GitHub source code repository.

`tnetwork` is a pure Python package; you don't need a compiler to build or install it.

GitHub

Clone the tnetwork repostitory (see GitHub for options)

```
git clone https://github.com/Yquetzal/tnetwork.git
```

2.1.3 Requirements

Python

To use tnetwork you need Python 3.6 or later.

2.2 Quick Start

This is an introduction to the key functionalities of the tnetwork library. Check documentation for more details

[1]:
```
%load_ext autoreload
%autoreload 2

import tnetwork as tn
import networkx as nx
import seaborn as sns
```

2.2.1 Creating a dynamic graph

We create a dynamic graph object. Two types exist, using snapshot or interval respresentations. In this example, we use intervals

[2]:
```
my_d_graph = tn.DynGraphIG()
```

We add some nodes and edges. Intervals are inclusive on the left and non inclusive on the right: [start,end[

Note that if we add edges between nodes that are not present (b from 3 to 5), the corresponding node presence is automatically added

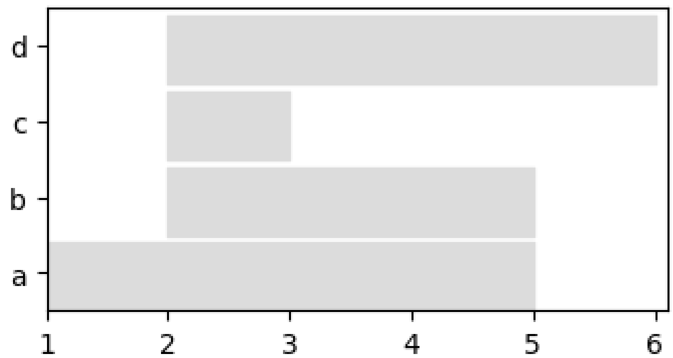
[3]:
```
my_d_graph.add_node_presence("a",(1,5)) #add node a during interval [1,5[
my_d_graph.add_nodes_presence_from(["a","b","c"],(2,3)) # add ndoes a,b,c from 2 to 3
my_d_graph.add_nodes_presence_from("d",(2,6)) #add node from 2 to 6

my_d_graph.add_interaction("a","b",(2,3)) # link nodes a and b from 2 to 3
my_d_graph.add_interactions_from(("b","d"),(2,5)) # link nodes b and d from 2 to 5
```

2.2.2 Visualizing your graph

We can visualize only nodes using a longitudinal representation

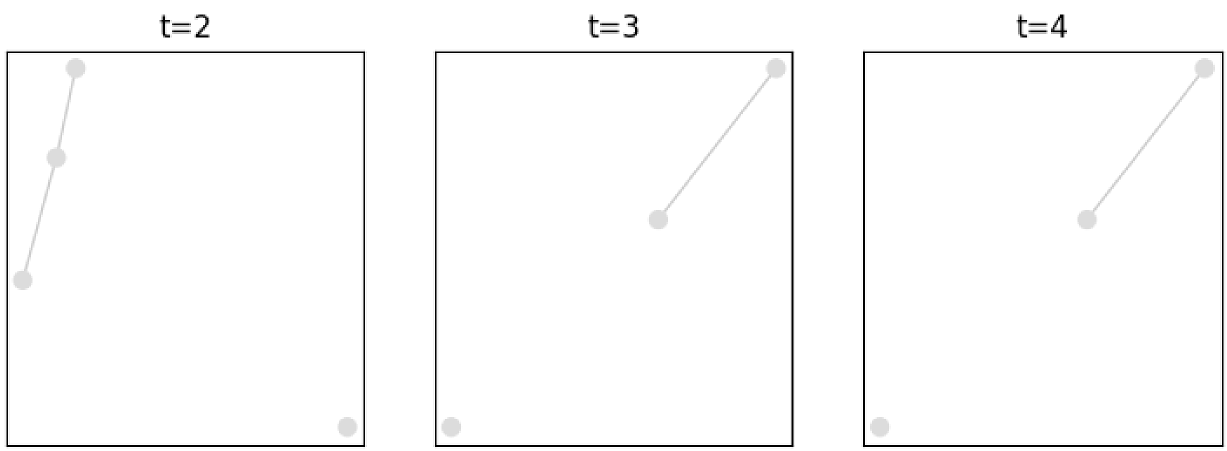
[4]:
```
plot = tn.plot_longitudinal(my_d_graph,width=400,height=200)
```

```
/usr/local/lib/python3.7/site-packages/numpy/core/numeric.py:2327: FutureWarning:␣
↪elementwise comparison failed; returning scalar instead, but in the future will␣
↪perform elementwise comparison
  return bool(asarray(a1 == a2).all())
```



Or visualize the whole graph at any given time

[5]:
```
plot = tn.plot_as_graph(my_d_graph,ts=[2,3,4],width=300,height=300)
```



2.2.3 Accessing graph information

We can query the graph at a given time and get a networkx object

[6]:
```
my_d_graph.graph_at_time(2).nodes()
```

[6]:
```
NodeView(('a', 'b', 'c', 'd'))
```

We can also query the presence periods of some nodes, for instance. Check documentation for more possibilities.

[7]:
```
my_d_graph.node_presence(["a","b"])
```

[7]:
```
{'a': [1,5[ , 'b': [2,5[ }
```

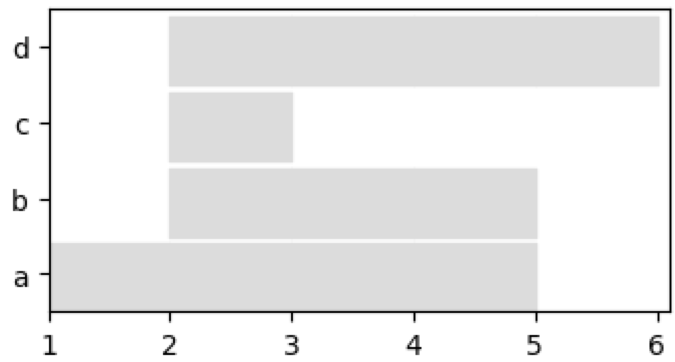
2.2.4 Conversion between snapshots<->interval representations

It is possible to transform an interval representation into a snapshot one, and reciprocally. We need to specify an aggregation step, i.e., each snapshot of the resulting dynamic graph corresponds to a period of the chosen length.

[8]:
```
my_d_graph_SN = my_d_graph.to_DynGraphSN(slices=1)
```
```
[(1, 2), (2, 3), (3, 4), (4, 5), (5, 6), (6, 7)]
```

We plot the graph to check that it has not changed (each snapshot has a duration of 1, a continuous horizontal line corresponds to a node present in several adjacent snapshots)

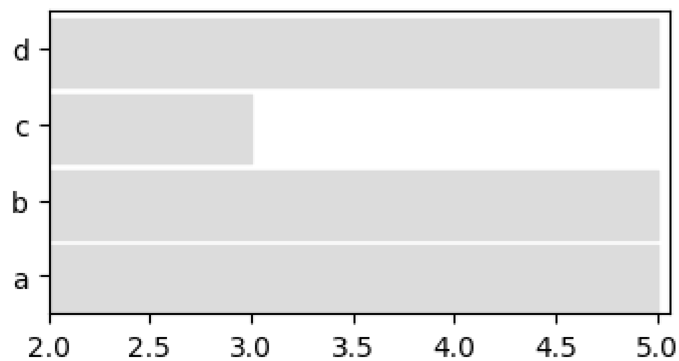
[9]:
```
to_plot = tn.plot_longitudinal(my_d_graph_SN,width=400,height=200)
```



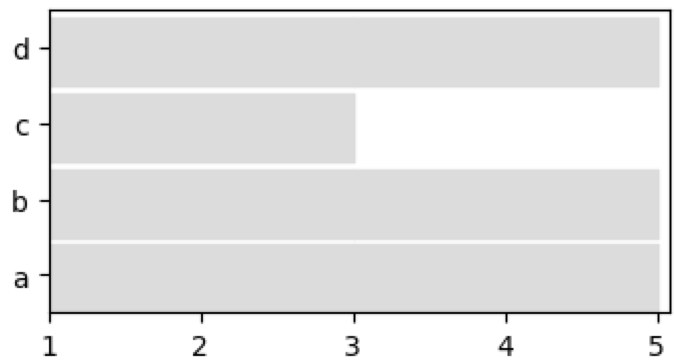
2.2.5 Slicing, aggregating

We can slice a dynamic network to keep only a chosen period, and re-aggregate it. Note that aggregation can be done according to dates (week, months…) if time values are provided as timestamps (see documentation for details)

[10]:
```
sliced = my_d_graph.slice(2,5)
to_plot = tn.plot_longitudinal(sliced,width=400,height=200)
```



[11]:
```
aggregated = my_d_graph_SN.aggregate_sliding_window(bin_size=2)
to_plot = tn.plot_longitudinal(aggregated,width=400,height=200)
```



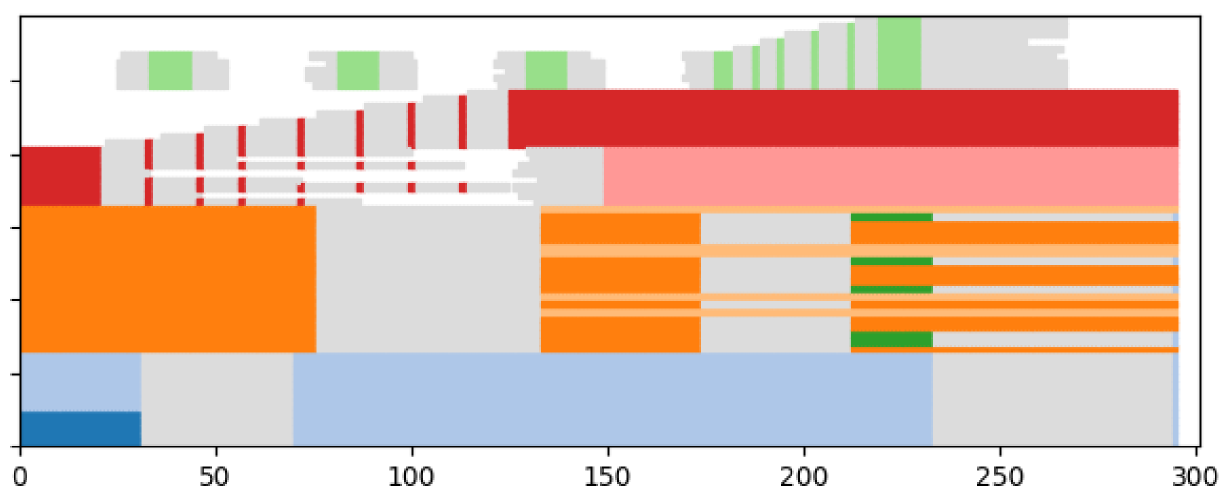
Generate and detect dynamic community structures

One of the key features of tnetwork is to be able to generate networks with community structures, and to detect dynamic communities in networks.

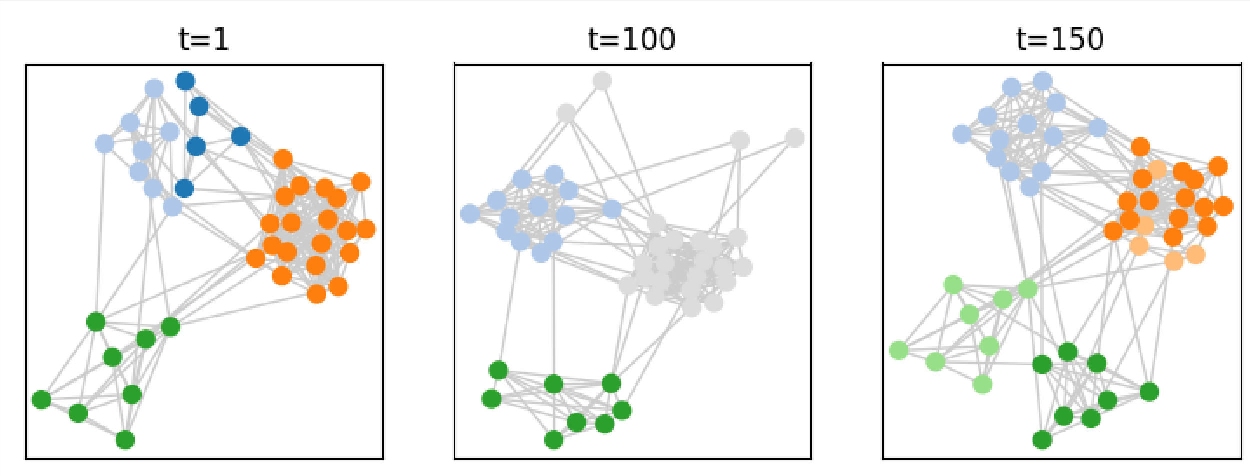
Let's start by generating a random toy model and plotting it with its communities represented as colors

[19]:
```
toy_graph,toy_ground_truth = tn.DCD.generate_toy_random_network(alpha=0.9,random_
↪noise=0.05)
plot = tn.plot_longitudinal(toy_graph,toy_ground_truth,height=300)
```

```
100% (26 of 26) |########################| Elapsed Time: 0:00:00 ETA:  00:00:00
```



[20]:
```
plot = tn.plot_as_graph(toy_graph,toy_ground_truth,ts=[1,100,150],width=300,
↪height=300)
```



We can then run a dynamic community detection algorithm on the graph. Several methods are available, check the documentation for more details

[21]:
```
dynamic_communities = tn.iterative_match(toy_graph)
```

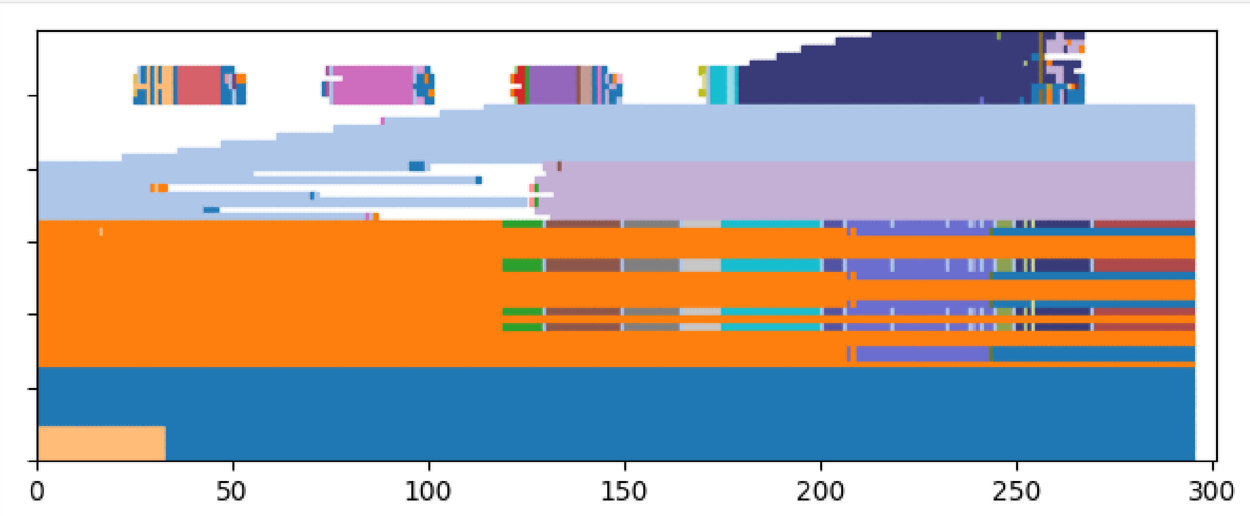
```
N/A% (0 of 295) |                        | Elapsed Time: 0:00:00 ETA:  --:--:--
```

```
starting no_smoothing
```

```
100% (295 of 295) |######################| Elapsed Time: 0:00:01 ETA:  00:00:00
```

Let's check what the communities found look like

[22]:
```
plot = tn.plot_longitudinal(communities=dynamic_communities,height=300)
```



Finally, we can evaluate the quality of this solution using some quality functions designed for dynamic communities, for instance:

[23]:
```
print("longitudinal similarity to ground truth: ",tn.longitudinal_similarity(toy_
↪ground_truth,dynamic_communities))
print("Partition smoothness SM-P: ",tn.SM_P(dynamic_communities))
```

```
longitudinal similarity to ground truth:  0.9108283486232346
Partition smoothness SM-P:  0.9318757198549844
```

```
[ ]:
```

2.3 Tutorials

All tutorials can be accessed as jupyter notebooks

2.3.1 Dynamic Network Classes

Table of Contents

1. *Creating a simple graph*

- *Using a snapshot representation*
- *Using an interval graph representation*
- *Using a Link Stream graph representation*

2. *Visualization*
3. *Conversion between graph types*
4. *Aggregation/Slicing*

- *Slicing*
- *Cumulated graphs*
- *Resampling*

If tnerwork library is not installed, you need to install it, for instance using the following command

```
[1]: #%%capture #avoid printing output
#!pip install --upgrade git+https://github.com/Yquetzal/tnetwork.git
```

```
[2]: %load_ext autoreload
%autoreload 2
import tnetwork as tn
```

Creating simple dynamic graphs and accessing their properties We will represent a graph with similar properties using snapshots and interval graphs

Using a snapshot representation

`DynGraphSN` is the class used to represent dynamic networks with snapshots (SN). The time at which each snapshot occurs is represented by an integer, which can be numbers in a sequence (1,2,3, etc.) or POSIX timestamps. A Frequency parameter allows to specify the time between each snapshot. By default, its value is 1. It is useful when there are missing snaphsots, e.g., like in SocioPatterns data, a snapshot every 20s, but many snapshots are empty.

```
[3]: dg_sn = tn.DynGraphSN(frequency=1)
dg_sn.add_node_presence("a",1) #add node a in snapshot 1
dg_sn.add_nodes_presence_from(["a","b","c"],[2,3,4,5]) #add nodes a,b,c in snapshots␣
↪2 to 5
```

(continues on next page)

(continued from previous page)

```
dg_sn.add_nodes_presence_from("d",[1,2,4,5]) #add node d in snapshots 1, 2, 4 and 5

dg_sn.add_interaction("a","b",2) #link a and b in snapshot 2
dg_sn.add_interaction("a","d",2) #link a and d in snapshot 2
dg_sn.add_interactions_from(("b","d"),[4,5]) #link b and d in snapshots 4 and 5
```

Using an interval graph representation.

DynGraphIG is the class used to represent dynamic networks with Interval Graphs (IG). Nodes and edges are present during time intervals, that are closed on the left and open on the right, e.g., (0,10) corresponds to the interval [0,10[, e.g., the node or edge exist from time 0 (included) to time 10 (excluded).

Note the similarity between the functions used for snapshots

Both graphs are equivalent if the snapshots of dg_sn have a duration of 1.

[4]:
```
dg_ig = tn.DynGraphIG()

dg_ig.add_node_presence("a",(1,2)) #add node a from time 1 to 2 (not included, time␣
↪duration =2-1 = 1)
dg_ig.add_nodes_presence_from(["a","b","c"],(2,6)) # add nodes a,b,c from 2 to 6
dg_ig.add_nodes_presence_from("d",[(1,3),(4,6)]) #add node d from 1 to 3 and from 4␣
↪to 6

dg_ig.add_interaction("a","b",(2,3)) # link nodes a and b from 2 to 3
dg_ig.add_interaction("a","d",(2,3)) # link nodes a and d from 2 to 3
dg_ig.add_interactions_from(("b","d"),(4,6)) # link nodes b and d from 4 to 6
```

Using a Link Stream representation

DynGraphLS is the class used to represent dynamic networks with Link Streams (LS). In a link stream, interactions are ponctual (no duration), but time is continuous. Nodes duration can be represented as intervals, or simply ignored. Note that if time is discrete, a link stream can represent data equivalent to a snapshot sequence: each edge of each snapshot is represented as an interaction at the corresponding time in the link stream. Discrete time can be handled using the frequency parameter of a link stream. In this example, we create a link stream equivalent to the one represented with other types.

Note the similarity between the functions used.

[5]:
```
dg_ls = tn.DynGraphLS(frequency=1)

dg_ls.add_node_presence("a",(1,2)) #add node a from time 1 to 2 (not included, time␣
↪duration =2-1 = 1)
dg_ls.add_nodes_presence_from(["a","b","c"],(2,6)) # add nodes a,b,c from 2 to 6
dg_ls.add_nodes_presence_from("d",[(1,3),(4,6)]) #add node d from 1 to 3 and from 4␣
↪to 6


dg_ls.add_interaction("a","b",2) #link a and b at time 2
dg_ls.add_interaction("a","d",2) #link b and d at time 2
dg_ls.add_interactions_from(("b","d"),[4,5]) #link b and d at times 4 and 5
```

```
('b', 'd')
```

Accessing functions

Using accessing functions, we can check that both graphs are very similar (Note that intervals are coded using the tnetwork.Intervals class, and are printed as [start,end[. Therefore, 2 snapshots of duration 1 at times 1 and 2 code a situation similar to an interval [1,3[

[6]:

```
print(dg_sn.graph_at_time(2).edges)
print(dg_ig.graph_at_time(2).edges)
print(dg_ls.graph_at_time(2).edges)
print(dg_sn.graph_at_time(4).edges)
print(dg_ig.graph_at_time(4).edges)
print(dg_ls.graph_at_time(4).edges)
```

```
[('a', 'b'), ('a', 'd')]
[('a', 'b'), ('a', 'd')]
[('a', 'b'), ('a', 'd')]
[('b', 'd')]
[('b', 'd')]
[('b', 'd')]
```

[7]:

```
print(dg_sn.node_presence())
print(dg_ig.node_presence())
print(dg_ls.node_presence())
```

```
{'a': [1, 2, 3, 4, 5], 'd': [1, 2, 4, 5], 'b': [2, 3, 4, 5], 'c': [2, 3, 4, 5]}
{'a': [1,6[ , 'b': [2,6[ , 'c': [2,6[ , 'd': [1,3[ [4,6[ }
{'a': [1,6[ , 'b': [2,6[ , 'c': [2,6[ , 'd': [1,3[ [4,6[ }
```

Visualization

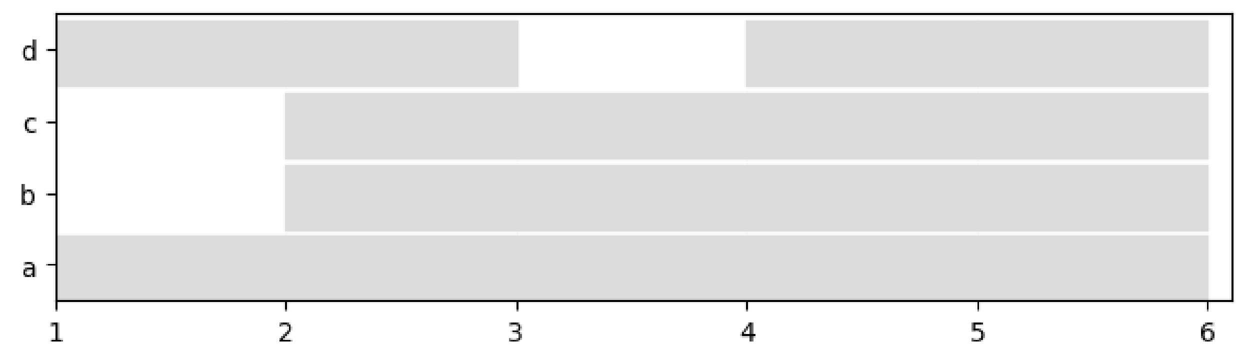
We can use a basic visualization to compare nodes presence of both representation.

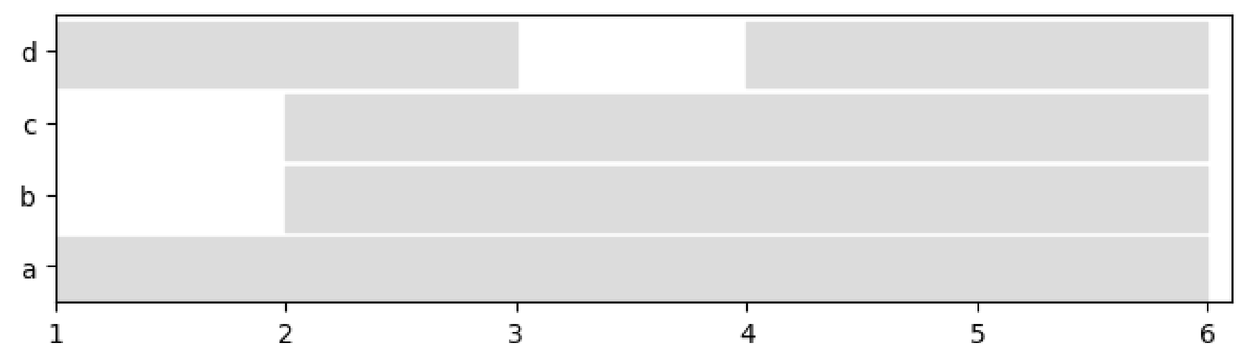
See the notebook on visualization to see more possibilities.

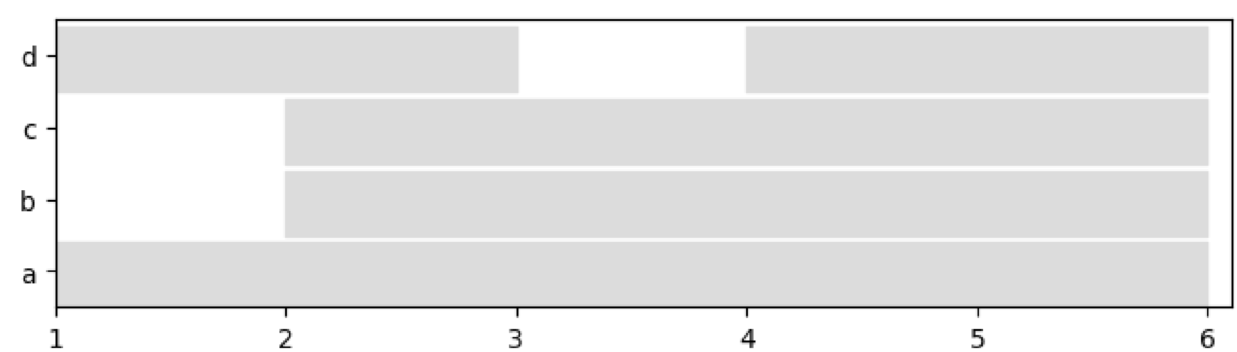
[8]:

```
plot = tn.plot_longitudinal(dg_sn,height=200)
plot = tn.plot_longitudinal(dg_ig,height=200)
plot = tn.plot_longitudinal(dg_ls,height=200)
```

```
/usr/local/lib/python3.7/site-packages/numpy/core/numeric.py:2327: FutureWarning:
↪elementwise comparison failed; returning scalar instead, but in the future will
↪perform elementwise comparison
  return bool(asarray(a1 == a2).all())
```

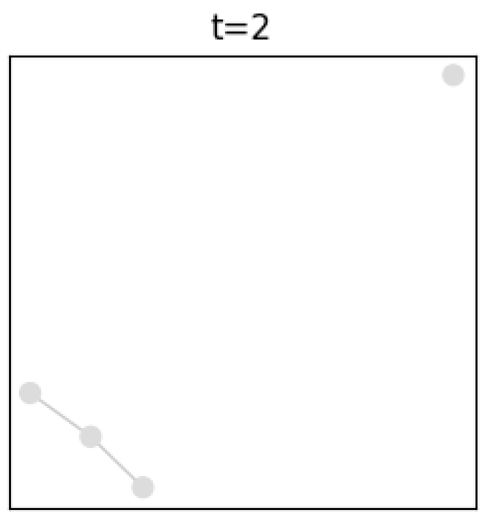




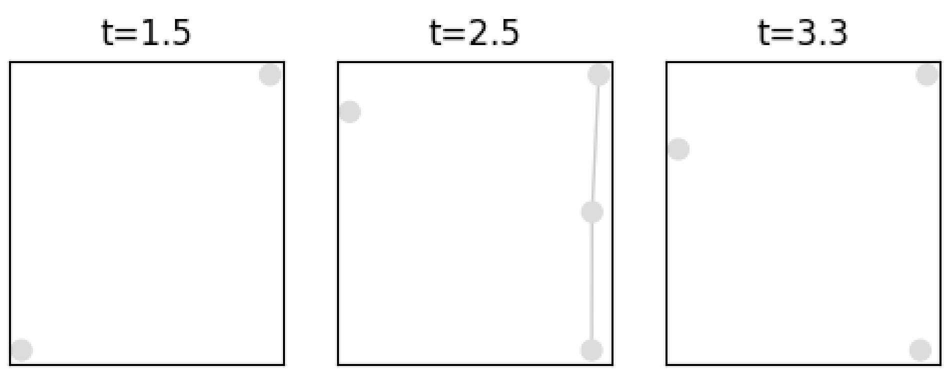


It is also possible to plot the graph at any given time.

[9]:
```
plot = tn.plot_as_graph(dg_sn,ts=2,auto_show=True,width=300,height=300)
```



[10]:
```
plot = tn.plot_as_graph(dg_ig,ts=[1.5,2.5,3.3],auto_show=True,width=200,height=200)
```



Conversion between snapshots and interval graphs

We convert the snapshot representation into an interval graph representation, using a snapshot lenght of 1.

We check that both graphs are now similar

[11]:
```
converted_to_IG = dg_sn.to_DynGraphIG()
print(converted_to_IG.node_presence())
print(dg_ig.node_presence())
print(converted_to_IG.edge_presence())
print(dg_ig.edge_presence())
```

```
{'a': [1,6[ , 'd': [1,3[ [4,6[ , 'b': [2,6[ , 'c': [2,6[ }
{'a': [1,6[ , 'b': [2,6[ , 'c': [2,6[ , 'd': [1,3[ [4,6[ }
{frozenset({'b', 'a'}): [(2, 3)], frozenset({'d', 'a'}): [(2, 3)], frozenset({'d', 'b
↪'}): [(4, 6)]}
{frozenset({'b', 'a'}): [(2, 3)], frozenset({'d', 'a'}): [(2, 3)], frozenset({'b', 'd
↪'}): [(4, 6)]}
```

Reciprocally, we transform the interval graph into a snapshot representation and check the similarity

```
[12]: converted_to_SN = dg_ig.to_DynGraphSN(slices=1)
print(converted_to_SN.node_presence())
print(dg_sn.node_presence())
print(converted_to_SN.edge_presence())
print(dg_sn.edge_presence())
```

```
{'a': [1, 2, 3, 4, 5], 'd': [1, 2, 4, 5], 'b': [2, 3, 4, 5], 'c': [2, 3, 4, 5]}
{'a': [1, 2, 3, 4, 5], 'd': [1, 2, 4, 5], 'b': [2, 3, 4, 5], 'c': [2, 3, 4, 5]}
{frozenset({'b', 'a'}): [2], frozenset({'d', 'a'}): [2], frozenset({'b', 'd'}): [4,
↪5]}
{frozenset({'b', 'a'}): [2], frozenset({'d', 'a'}): [2], frozenset({'b', 'd'}): [4,
↪5]}
```

```
[13]: converted_to_LS = dg_sn.to_DynGraphLS()
print(converted_to_LS.node_presence())
print(dg_sn.node_presence())
print(converted_to_LS.edge_presence())
print(dg_sn.edge_presence())
```

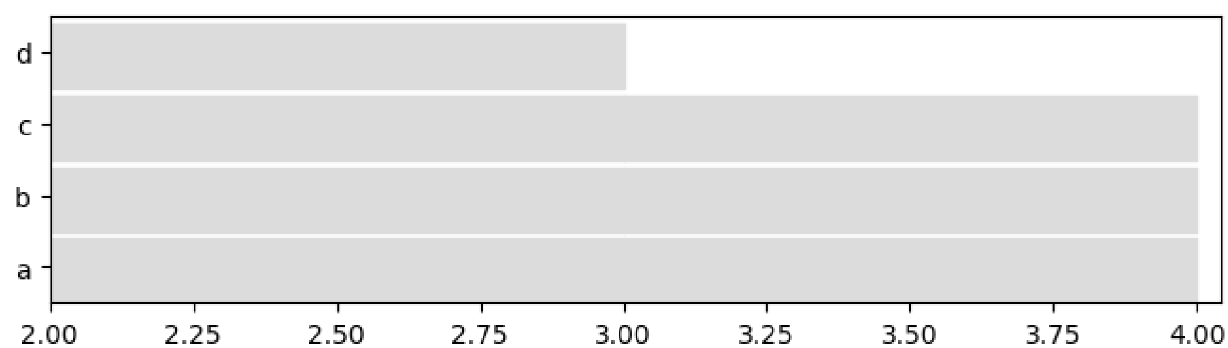
```
{'a': [1,6[ , 'd': [1,3[ [4,6[ , 'b': [2,6[ , 'c': [2,6[ }
{'a': [1, 2, 3, 4, 5], 'd': [1, 2, 4, 5], 'b': [2, 3, 4, 5], 'c': [2, 3, 4, 5]}
{frozenset({'b', 'a'}): SortedSet([2]), frozenset({'d', 'a'}): SortedSet([2]),
↪frozenset({'d', 'b'}): SortedSet([4, 5])}
{frozenset({'b', 'a'}): [2], frozenset({'d', 'a'}): [2], frozenset({'b', 'd'}): [4,
↪5]}
```

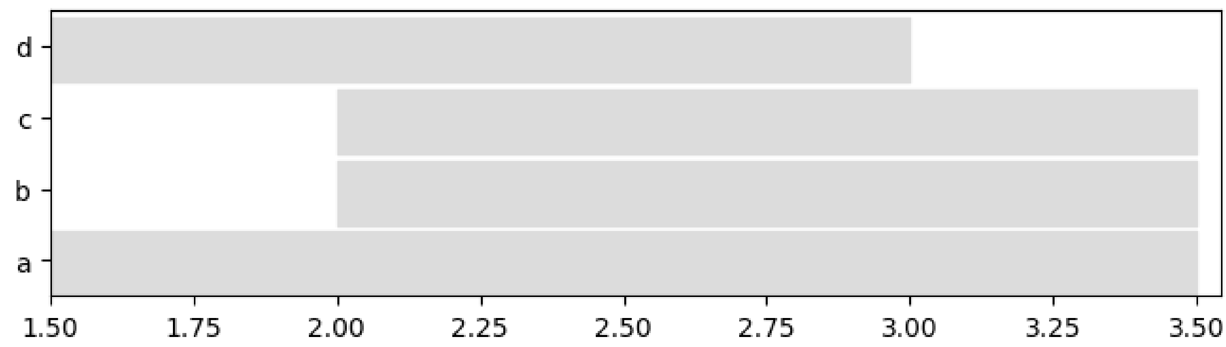
Aggregation/Slicing

Slicing

One can conserve only a chosen period using the slice function

```
[14]: sliced_SN = dg_sn.slice(2,4) #Keep only the snapshots from 2 to 4
sliced_IG = dg_ig.slice(1.5,3.5) #keep only what happens between 1.5 and 3.5 in the
↪interval graph

plot = tn.plot_longitudinal(sliced_SN,height=200)
plot = tn.plot_longitudinal(sliced_IG,height=200)
```

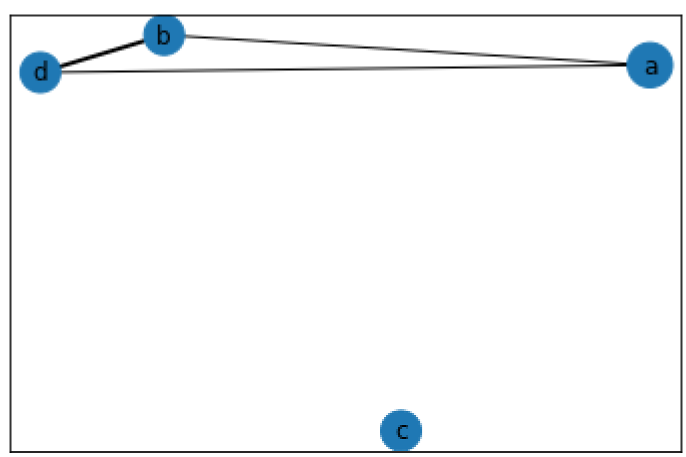




Creating cumulated graphs

It can be useful to create cumulated weighted graphs to summarize the presence of nodes and edges over a period

[15]:
```
import networkx as nx
%matplotlib inline
g_cumulated = dg_sn.cumulated_graph()

#Similarly for interval graphs:
#g_cumulated = dg_ig.cumulated_graph()

#Draw with node size and edge width propotional to weights in the cumulated graph
nx.draw_networkx(g_cumulated,node_size=[g_cumulated.nodes[n]['weight']*100 for n in g_
↪cumulated.nodes], width = [g_cumulated[u][v]['weight'] for u,v in g_cumulated.
↪edges])
```



Graphs can also be cumulated only over a specific period

[16]:
```
g_cumulated = dg_sn.cumulated_graph([1,2]) # create a static graph cumulating␣
↪snapshots
g_cumulated = dg_ig.cumulated_graph((1,3))
```

Resampling

Sometimes, it is useful to study dynamic network with a lesser temporal granularity than the original data.

Several functions can be used to aggregate dynamic graphs, thus yielding snapshots covering larger periods.

To exemplify this usage, we use a dataset from the sociopatterns project (http://www.sociopatterns.org) that can be loaded in a single command in the chosen format

[17]:
```
sociopatterns = tn.graph_socioPatterns2012(tn.DynGraphSN)
```

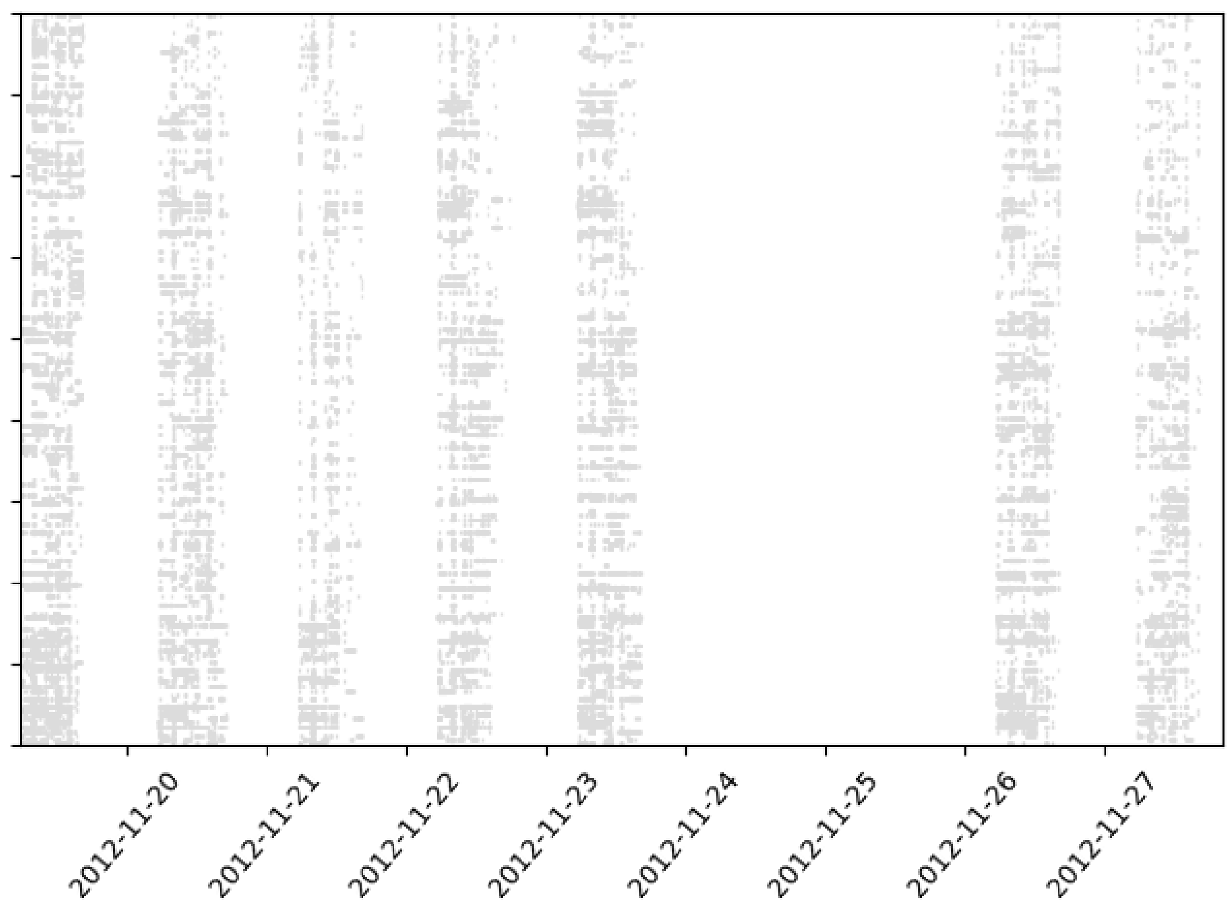
```
graph will be loaded as:  <class 'tnetwork.dyn_graph.dyn_graph_sn.DynGraphSN'>
```

For this original network loaded as a snapshot representation, we print the number of snapshots and the first and last dates (the dataset covers 9 days, including a week-end with no activity)

[18]:
```
from datetime import datetime
all_times = sociopatterns.snapshots_timesteps()
print("# snapshots:",len(all_times))
print("first date:",datetime.utcfromtimestamp(all_times[0])," laste date:",datetime.
↪utcfromtimestamp(all_times[-1]))
```

```
# snapshots: 11273
first date: 2012-11-19 05:36:20  laste date: 2012-11-27 16:14:40
```

[19]:
```
#Be careful, the plot takes a few seconds to draw.
to_plot_SN = tn.plot_longitudinal(sociopatterns,height=500,sn_duration=20,to_
↪datetime=True)
```



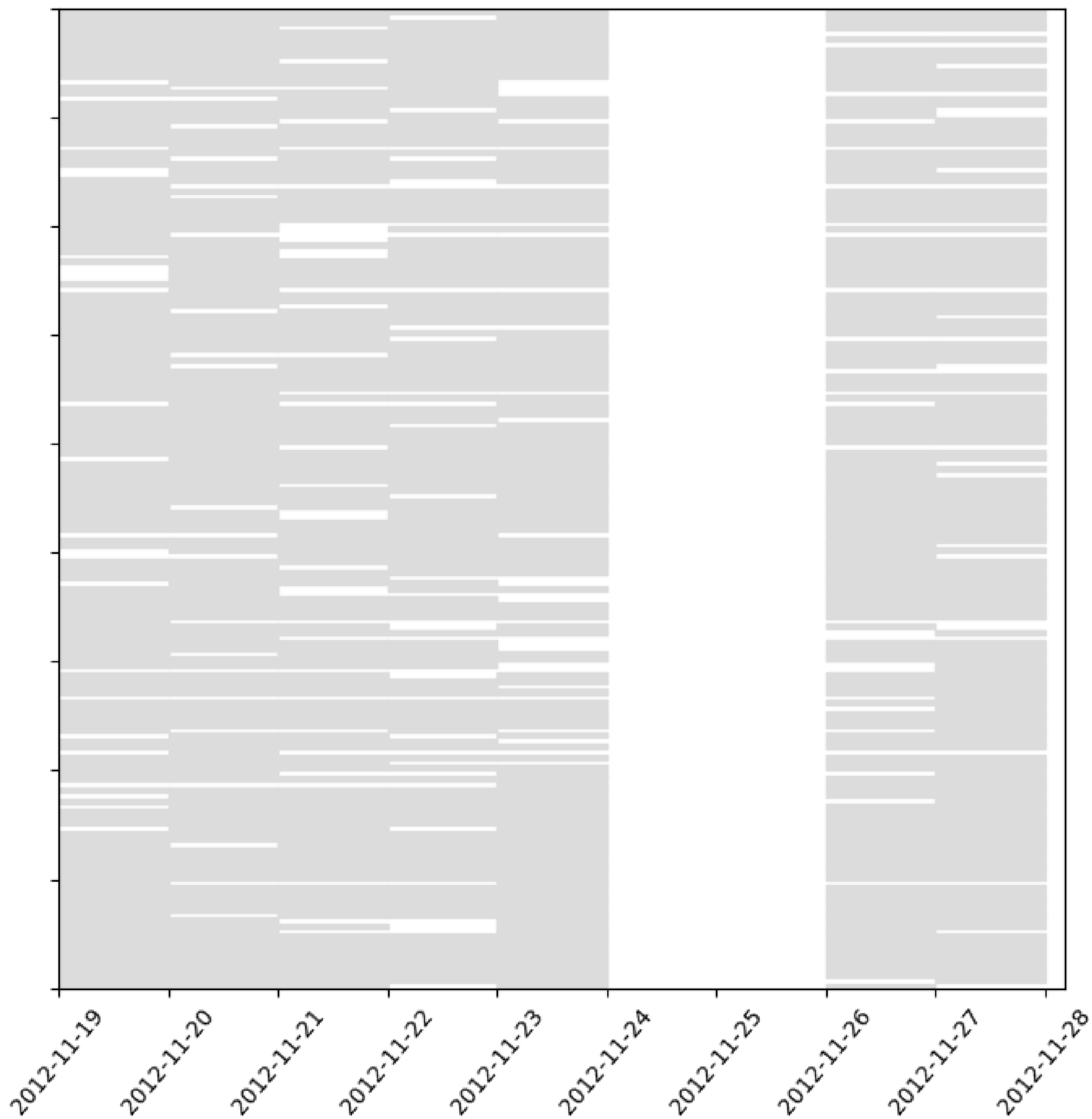
We then aggregate on fixed time periods using the `aggregate_time_period` function. Although there are several ways to call this function, the simplest one is using a string such as "day", "hour", "month", etc. Note how the beginning of the first snapshot is now on midnight of the day on which the first observation was made

[20]:
```
sociopatterns_Day = sociopatterns.aggregate_time_period("day")
```

[21]:
```

all_times = sociopatterns_Day.snapshots_timesteps()
print("# snapshots:",len(all_times))
print("first date:",datetime.utcfromtimestamp(all_times[0])," laste date:",datetime.
↪utcfromtimestamp(all_times[-1]))
```

```
# snapshots: 7
first date: 2012-11-19 00:00:00  laste date: 2012-11-27 00:00:00
```

[22]:
```
to_plot_SN = tn.plot_longitudinal(sociopatterns_Day,height=800,to_datetime=True,sn_
↪duration=24*60*60)
```



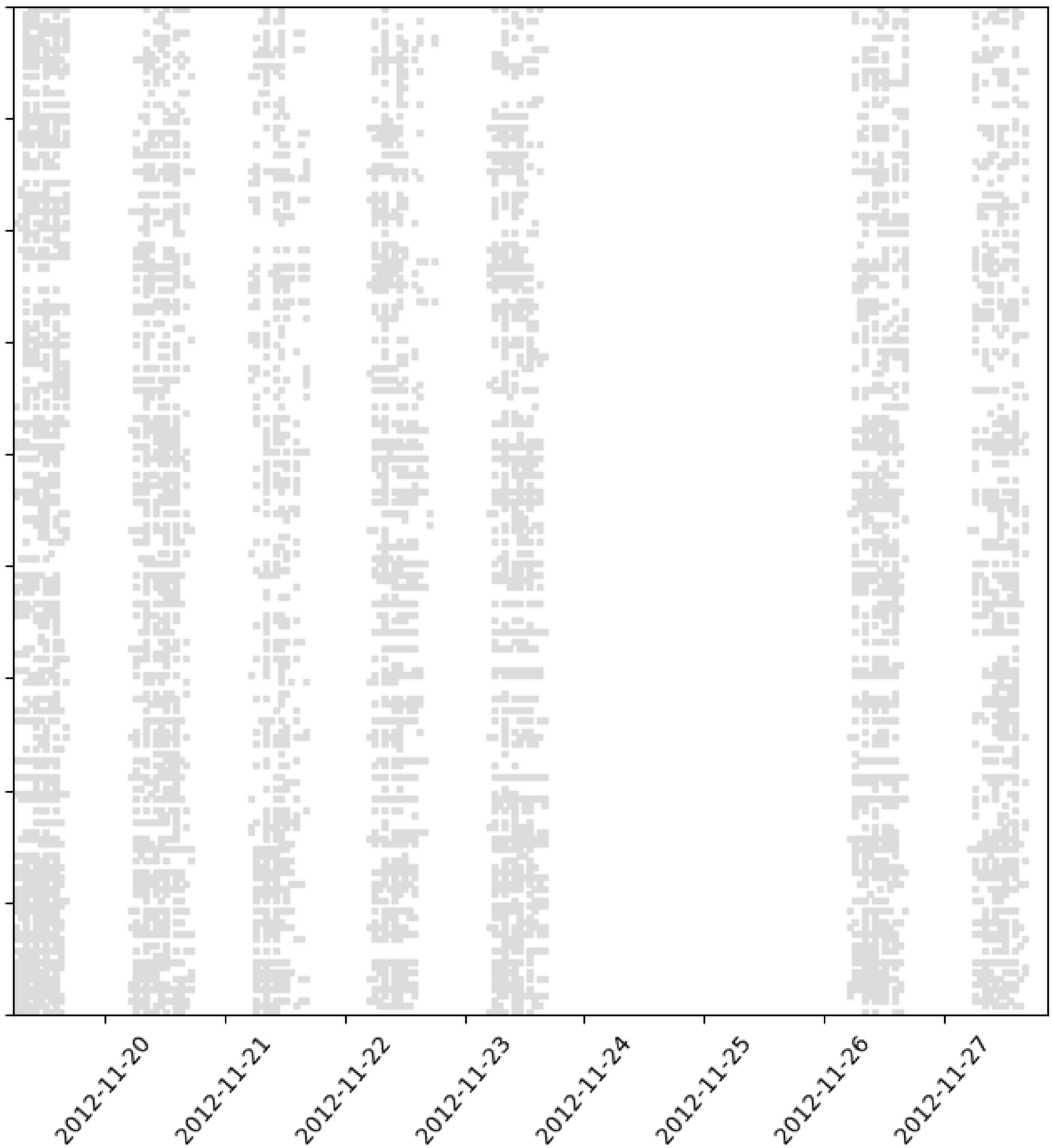
Another way to aggregate is to use sliding windows. In this example, we use non-overlapping windows of one hour, but it is possible to have other parameters, such as overlapping windows. Note how, this time, the first snapshot starts exactly at the time of the first observation in the original data

[23]:
```
sociopatterns_hour_window = sociopatterns.aggregate_sliding_window(bin_size=60*60)
```

[24]:
```
all_times = sociopatterns_hour_window.snapshots_timesteps()
print("# snapshots:",len(all_times))
print("first date:",datetime.utcfromtimestamp(all_times[0])," laste date:",datetime.
↪utcfromtimestamp(all_times[-1]))
```

```
# snapshots: 86
first date: 2012-11-19 05:36:20  laste date: 2012-11-27 15:36:20
```

[25]:
```
plot =tn.plot_longitudinal(sociopatterns_hour_window,height=800,to_datetime=True,sn_
↪duration=60*60)
```



[ ]:

[ ]:

2.3.2 Visualization

In this notebook, we will introduce the different types of visualization available in tnetwork.

There are two types: visualization of graphs at particular time (e.g., a particular snapshot), and visualization of the evolution of the community structure (longitudinal visualization)

If tnerwork library is not installed, you need to install it, for instance using the following command

[1]:
```
#%%capture #avoid printing output
#!pip install --upgrade git+https://github.com/Yquetzal/tnetwork.git
```

[2]:
```
import tnetwork as tn
import seaborn as sns
import pandas as pd
import networkx as nx
import numpy as np
```

Let's start with a toy example generated using tnetwork generator (see the corresponding documentation for details)

[3]:
```
my_scenario = tn.ComScenario()
[com1,com2] = my_scenario.INITIALIZE([6,6],["c1","c2"])
(com2,com3)=my_scenario.THESEUS(com2,delay=20)
my_scenario.DEATH(com2,delay=10)

(generated_network_IG,generated_comunities_IG) = my_scenario.run()
```

```
100% (8 of 8) |##########################| Elapsed Time: 0:00:00 ETA:  00:00:00
```

Cross-section visualization

One way to see a dynamic graph is to plot it as a series of standard static graph. We can start by plotting a single graph at a single time.

There are two libraries that can be used to render the plot: networkx (using matplotlib) or bokeh. matplotlib has the advantage of being more standard, while bokeh has the advantage of providing interactive graphs. This is especially useful to check who is each particular node or community in real datasets.

But Bokeh also has weaknesses: * It can alter the responsiveness of the netbook if large visualization are embedded in it * In some online notebooks e.g., google colab, embedding bokeh pictures in the notebook does not work well.

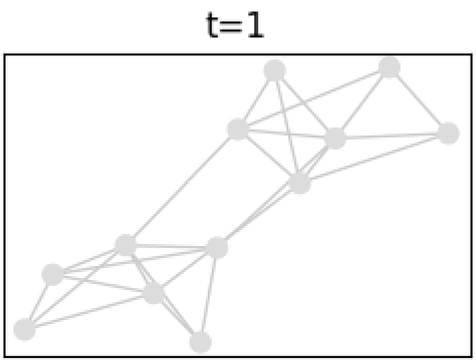
As a consequence, it is recommended to embed bokeh visualization in notebooks only for small graphs, and to open them in new windows for larger ones.

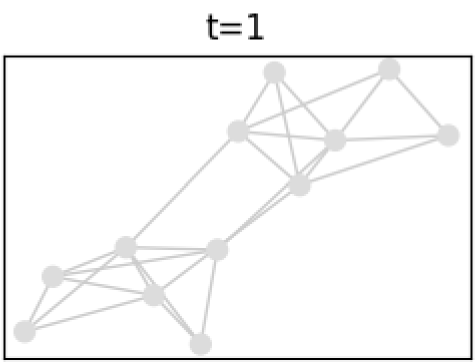
Let's start by plotting the networks in timestep 1 (ts=1). First, using matplotlib, the default option.

[4]:
```
tn.plot_as_graph(generated_network_IG,ts=1,width=300,height=200)
```

```
/usr/local/lib/python3.7/site-packages/numpy/core/numeric.py:2327: FutureWarning:
↪elementwise comparison failed; returning scalar instead, but in the future will
↪perform elementwise comparison
  return bool(asarray(a1 == a2).all())
```

[4]:





Then, using bokeh and the `auto_show` option. It won't work in google colab, see a solution below.

[5]:
```
tn.plot_as_graph(generated_network_IG,ts=1,width=600,height=300,bokeh=True,auto_
↪show=True)
```

Data type cannot be displayed: application/javascript, application/vnd.bokehjs_load.v0+json

Data type cannot be displayed: application/vnd.bokehjs_exec.v0+json, text/html

[5]:
```
Row(id='1080', ...)
```

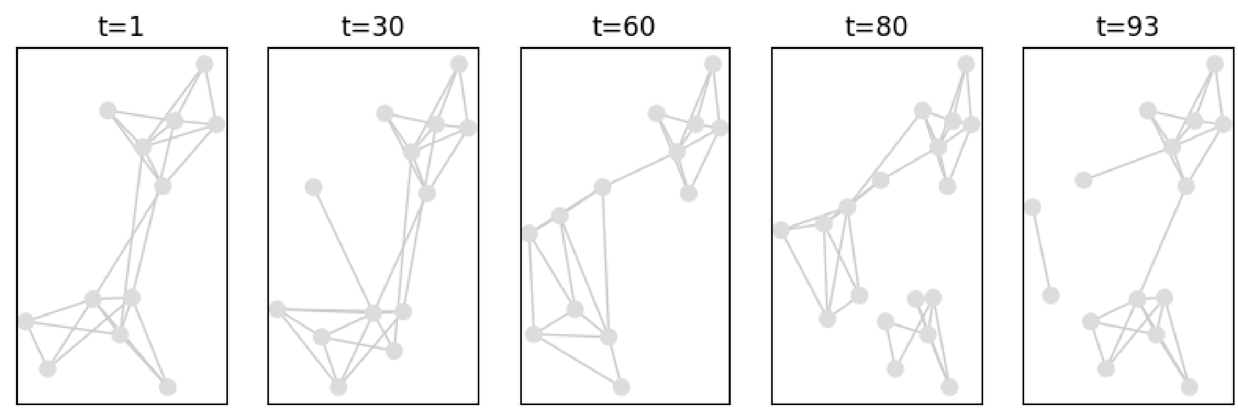
One can plot in a new window (and/or in a file) by ignoring the auto_show option, and instead receiving a figure, that we can manipulate as usual with bokeh

[6]:
```
from bokeh.plotting import figure, output_file, show
fig = tn.plot_as_graph(generated_network_IG,ts=1,width=600,height=300,bokeh=True)
output_file("fig.html")
show(fig)
```

Instead of plotting a single graph, we can plot several ones in a single call. Note that in this case, the position of nodes is common to all plots, and is decided based on the cumulated network

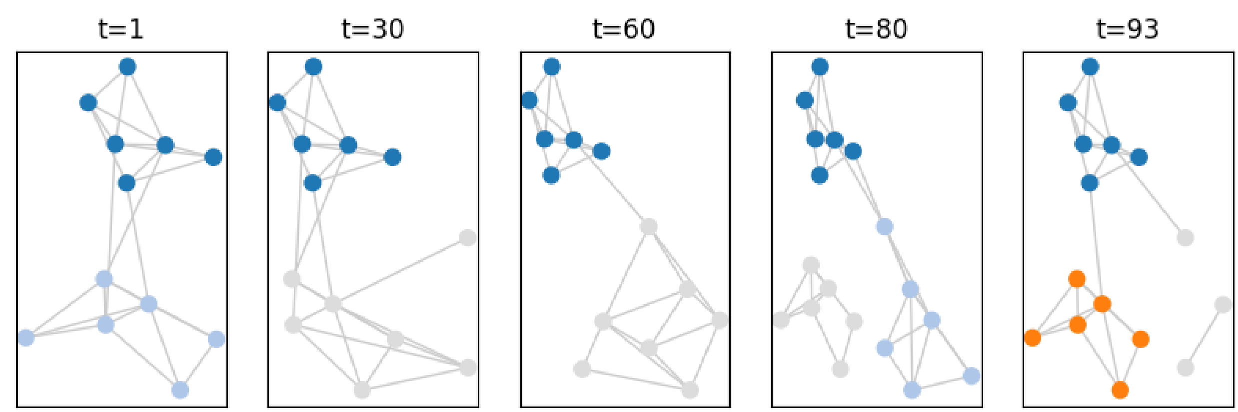
[7]:
```
from bokeh.plotting import figure, output_file, show
fig = tn.plot_as_graph(generated_network_IG,ts=[1,30,60,80,generated_network_IG.end()-
↪1],width=200,height=300)
```

```
/usr/local/lib/python3.7/site-packages/numpy/core/numeric.py:2327: FutureWarning:
↪elementwise comparison failed; returning scalar instead, but in the future will
↪perform elementwise comparison
  return bool(asarray(a1 == a2).all())
```



If we have dynamic communities associated with this dynamic graph, we can plot them too. Note that the same function accepts snapshots and interval graphs, but both the graph and the community structure must have the same format (SN or IG)

[8]:
```
from bokeh.plotting import figure, output_file, show
fig = tn.plot_as_graph(generated_network_IG,generated_comunities_IG,ts=[1,30,60,80,
↪generated_network_IG.end()-1],auto_show=True,width=200,height=300)
```



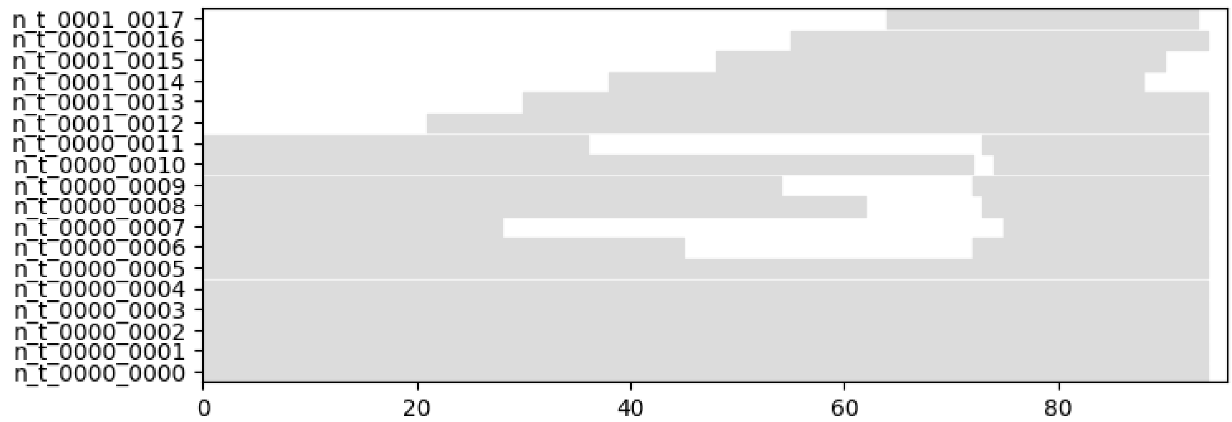
Longitudinal Visualization

The second type of visualization plots only nodes and not edges.

Time corresponds to the x axis, while each node has a fixed position on the y axis.

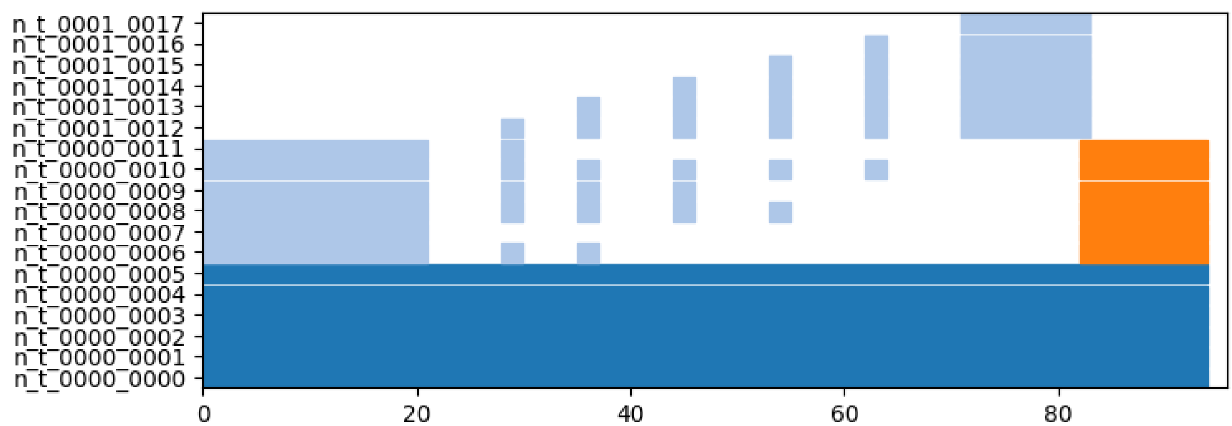
It is possible to plot only a dynamic graphs, without communities. White means that the node is not present or has no edges

[9]:
```
plot = tn.plot_longitudinal(generated_network_IG,height=300)
```



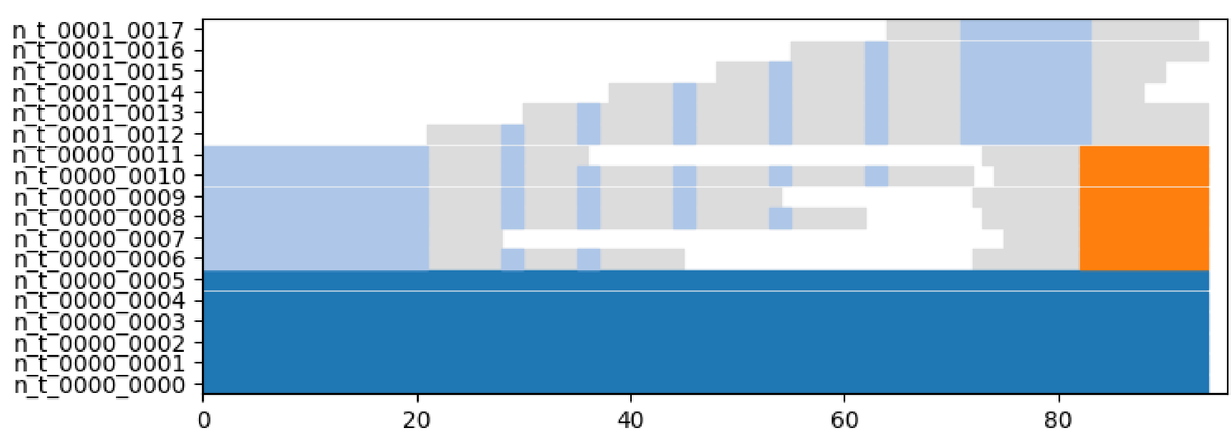
Or only communities, without a graph:

[10]:
```
plot = tn.plot_longitudinal(communities=generated_comunities_IG,height=300)
```



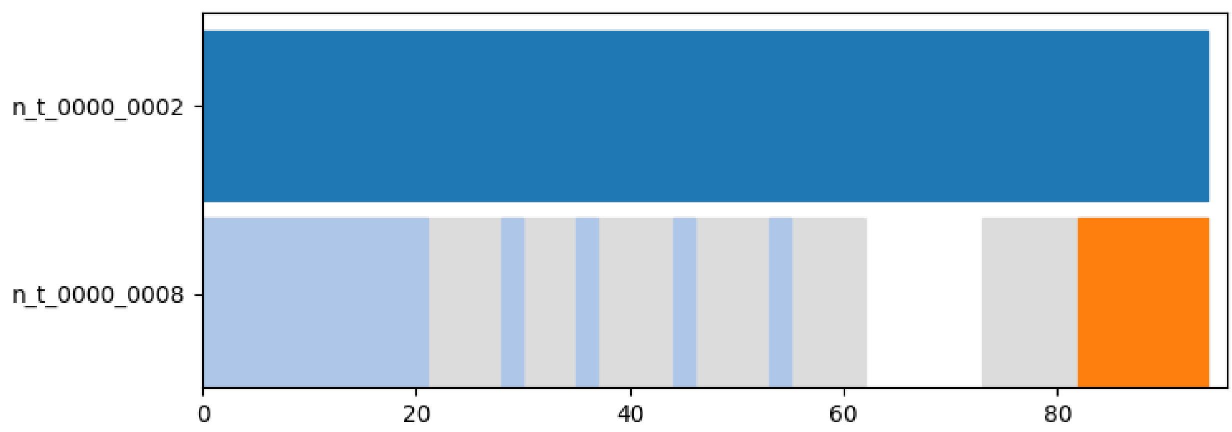
Or both on the same graph. The grey color always corresponds to nodes whithout communities. Other colors corresponds to communities

[11]:
```
plot = tn.plot_longitudinal(generated_network_IG,communities=generated_comunities_IG,
↪height=300)
```



It is possible to plot only a subset of nodes, and/or to plot them in a particular order

[12]:
```
plot = tn.plot_longitudinal(generated_network_IG,communities=generated_comunities_IG,
↪height=300,nodes=["n_t_0000_0008","n_t_0000_0002"])
```



Timestamps

It is common, when manipulating real data, to have dates in the form of timestamps. There is an option to automatically transform timestamps to dates on the x axis : `to_datetime`

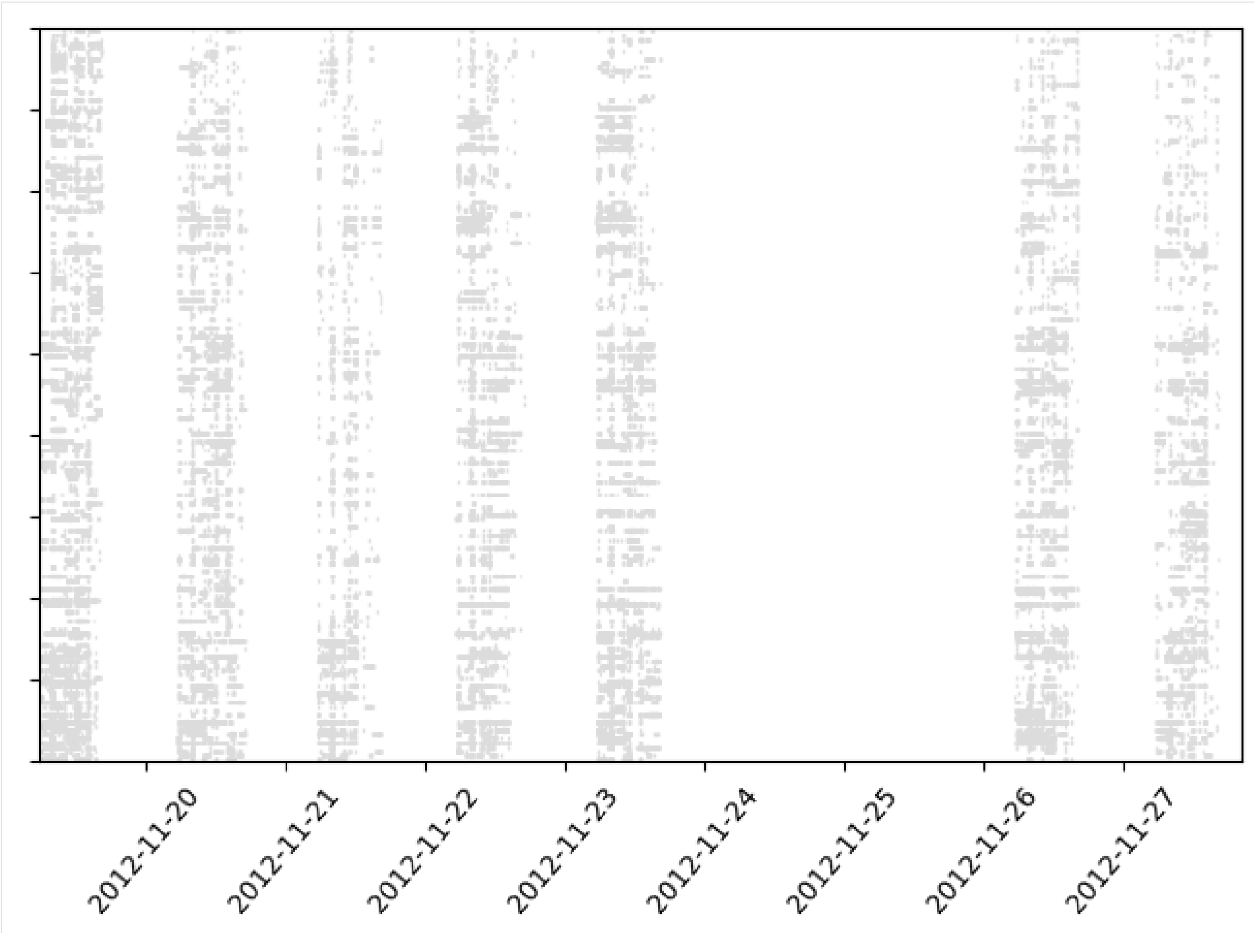
We give an example using the sociopatterns dataset

[14]:
```
sociopatterns = tn.graph_socioPatterns2012(format=tn.DynGraphSN)
```

```
graph will be loaded as:  <class 'tnetwork.dyn_graph.dyn_graph_sn.DynGraphSN'>
```

[15]:
```
#It takes a few seconds
to_plot_SN = tn.plot_longitudinal(sociopatterns,height=500,to_datetime=True)
```

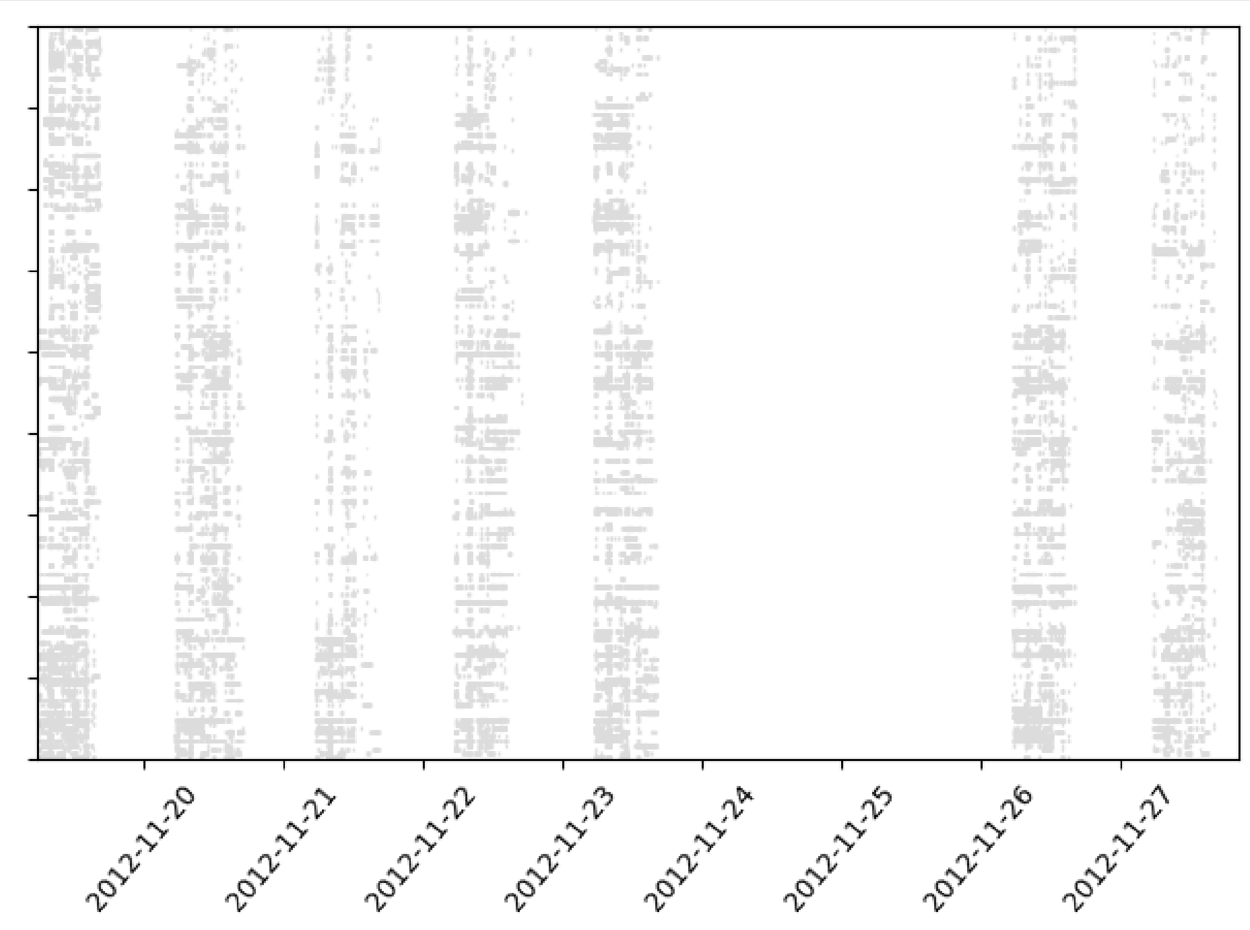
```
/usr/local/lib/python3.7/site-packages/numpy/core/numeric.py:2327: FutureWarning:␣
↪elementwise comparison failed; returning scalar instead, but in the future will␣
↪perform elementwise comparison
  return bool(asarray(a1 == a2).all())
```



Snapshot duration

By default, snapshots last until the next snapshot. If snapshots have a fix duration, there is a parameter to indicate this duration : `sn_duration`

[16]:
```
#in sociopatterns, there is an observed snapshot every 20 seconds.
to_plot_SN = tn.plot_longitudinal(sociopatterns,height=500,to_datetime=True,sn_
↪duration=20)
```



Bokeh longitudinal plots

Longitudinal plots can also use bokeh. It is clearly interesting to have ineractive plots in order to zoom on details or to check the name of communities or nodes. However, bokeh plots with large number of elements can quickly become unresponsive, that is why there are not used by default.

By adding the parameter `bokeh=True`, you can obtain a bokeh plot exactly like for the cross-section graphs, with or without the `auto_show` option.

```
[17]: tn.plot_longitudinal(generated_network_IG,communities=generated_comunities_IG,
↪height=300,bokeh=True,auto_show=True)
```

Data type cannot be displayed: application/javascript, application/vnd.bokehjs_load.v0+json

Data type cannot be displayed: application/vnd.bokehjs_exec.v0+json, text/html

```
[17]: Figure(id='1510', ...)
```

[18]:
```
from bokeh.plotting import figure, output_file, show
fig = tn.plot_longitudinal(sociopatterns,bokeh=True)
output_file("fig.html")
show(fig)
```

[ ]:
```

```

2.3.3 Dynamic Community classes

Table of Contents

1. *Initializing a dynamic community structure*

- [Using a snapshot representation]
- [Using an interval graph representation]

2. *Accessing properties of communities*
3. *Duration,frequencies of relations between nodes and communities*
4. *Visualization*
5. *Conversion between snapshots and interval graphs*
6. *Slicing*

If tnerwork library is not installed, you need to install it, for instance using the following command

[1]:
```
#%%capture #avoid printing output
#!pip install --upgrade git+https://github.com/Yquetzal/tnetwork.git
```

[2]:
```
%load_ext autoreload

%autoreload 2
import tnetwork as tn
```

Initializing a dynamic community structure ### With snapshots

[3]:
```
com_sn = tn.DynCommunitiesSN()
com_sn.add_affiliation("a","com1",1)
com_sn.add_affiliation({"b","c"},"com2",[2,3])
com_sn.add_affiliation("d",{"com1"},[1,3])
```

With Interval graphs

As with dynamic graphs, intervals are closed on the left and open on the right. Periods can be represented in different manners, as shown in the following example

[4]:
```
com_ig = tn.DynCommunitiesIG()
com_ig.add_affiliation("a","com1",(1,2))
com_ig.add_affiliation({"b","c"},"com2",tn.Intervals((2,4)))
com_ig.add_affiliation("d",{"com1"},[(1,2),(3,4)])
```

Accessing properties of communities

Check communities

We check the sate of communities at a particular time.

Static communities can be accessed in two forms

- in the **community** form (key = community ID, value = set of nodes)
- in **affiliation** form (key = a noe, value = set of communities)

Example, state of communities at time 3, in *community* and *affiliation* forms

[5]:
```
print(com_sn.communities(3))
print(com_sn.affiliations(3))
print(com_ig.communities(3))
print(com_ig.affiliations(3))
```

```
{'com2': {'c', 'b'}, 'com1': {'d'}}
{'c': {'com2'}, 'b': {'com2'}, 'd': {'com1'}}
{'com2': {'b', 'c'}, 'com1': {'d'}}
{'c': {'com2'}, 'b': {'com2'}, 'd': {'com1'}}
```

The same form exist to access dynamic communities. * communities form: for each community, for each of its nodes, presence time * affiliation form: for each node, for each of its communities, presence time

[6]:
```
print(com_sn.communities())
print(com_ig.communities())
print(com_sn.affiliations())
print(com_ig.affiliations())
```

```
{'com1': {'a': [1], 'd': [1, 3]}, 'com2': {'c': [2, 3], 'b': [2, 3]}}
{'com1': {'a': [1,2[ , 'd': [1,2[ [3,4[ }, 'com2': {'c': [2,4[ , 'b': [2,4[ }}
{'a': {'com1': [1]}, 'd': {'com1': [1, 3]}, 'c': {'com2': [2, 3]}, 'b': {'com2': [2,␣
↪3]}}
{'a': {'com1': [1,2[ }, 'c': {'com2': [2,4[ }, 'b': {'com2': [2,4[ }, 'd': {'com1':␣
↪[1,2[ [3,4[ }}
```

In each snapshot

For snapshots representation, it is also possible to obtain communities in each snapshot

[7]:
```
print(com_sn.snapshot_affiliations())
```

```
SortedDict({1: {'a': {'com1'}, 'd': {'com1'}}, 2: {'c': {'com2'}, 'b': {'com2'}}, 3: {
↪'c': {'com2'}, 'b': {'com2'}, 'd': {'com1'}}})
```

Duration/frequencies of relations between nodes and communities

One can check how long does a node belong to a community, in total

[8]:
```
print(com_sn.affiliations_durations("d","com1"))
print(com_ig.affiliations_durations("d","com1"))
```

```
2
2
```

One can also check directly the duration of affiliations of each node to each community

[9]:
```
print(com_sn.affiliations_durations())
print(com_ig.affiliations_durations())
```

```
{('a', 'com1'): 1, ('b', 'com2'): 2, ('c', 'com2'): 2, ('d', 'com1'): 2}
{('a', 'com1'): 1, ('b', 'com2'): 2, ('c', 'com2'): 2, ('d', 'com1'): 2}
```

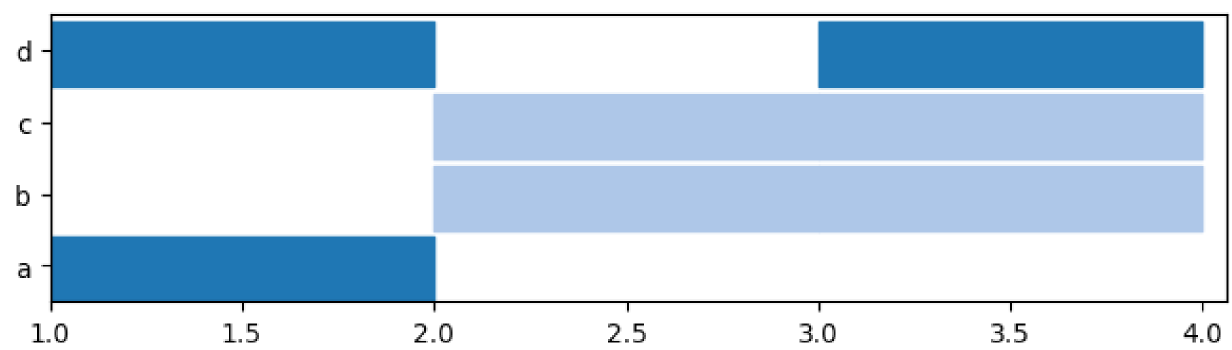
Visualization

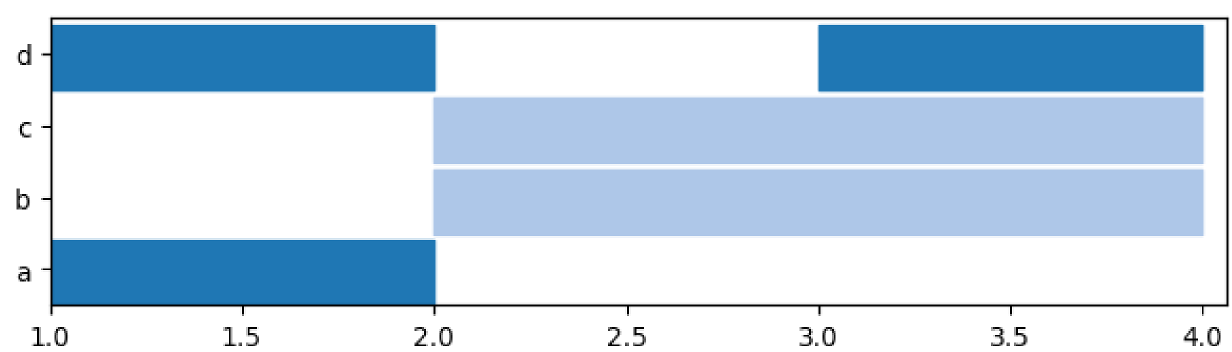
A simple example of visualization. To see more possibilities, see the dedicated section of the documentation

Note that it is the same function which is used to plot longitudinial graphs and communities. That is why we need to specify that what we provide corresponds to the `communities` parameter. One can provide both a graph and a dynamic community structure to this function.

[10]:
```
plot = tn.plot_longitudinal(communities=com_sn,height=200)
plot = tn.plot_longitudinal(communities=com_ig,height=200)
```

```
/usr/local/lib/python3.7/site-packages/numpy/core/numeric.py:2327: FutureWarning:␣
↪elementwise comparison failed; returning scalar instead, but in the future will␣
↪perform elementwise comparison
  return bool(asarray(a1 == a2).all())
```





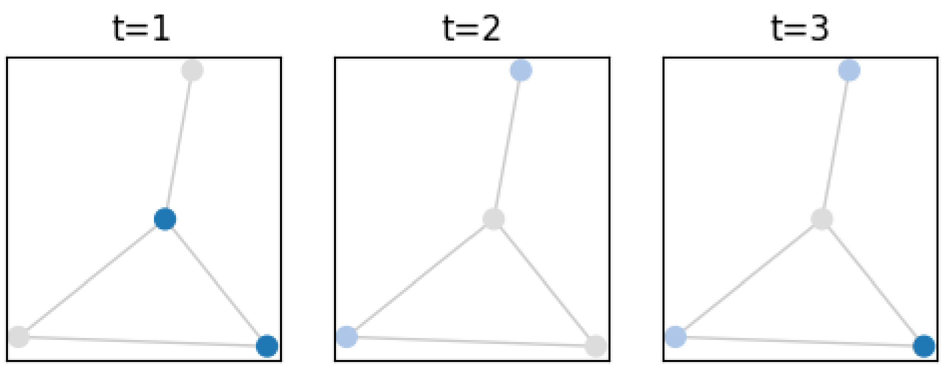
One can also plot a graph with nodes color corresponding to communities. In this example, we create a dynamic graph with a fix structure, and plot the communities we defined above

[11]:
```
graph_toy = tn.DynGraphIG()
graph_toy.add_interaction("a","b",(1,4))
graph_toy.add_interaction("a","c",(1,4))
graph_toy.add_interaction("a","d",(1,4))
```

(continues on next page)

(continued from previous page)

```
graph_toy.add_interaction("b","d",(1,4))

plot = tn.plot_as_graph(graph_toy,com_ig,[1,2,3],auto_show=True,width=200,height=200)
```



Conversion between snapshots and interval representation

Dynamic network representations can be converted by calling the appropriate function. * When converting to interval graphs, we provide the duration of each snapshots * When converting to snapshots, we provide the slices to which each snapshot should correspond. Note that it is tehrefore possible to have snapshots corresponding to overlapping periods

[12]:
```
converted_ig = com_sn.to_DynCommunitiesIG(1)
print(converted_ig.communities())
print(com_ig.communities())
```

```
{'com1': {'a': [1,2[ , 'd': [1,2[ [3,4[ }, 'com2': {'c': [2,4[ , 'b': [2,4[ }}
{'com1': {'a': [1,2[ , 'd': [1,2[ [3,4[ }, 'com2': {'c': [2,4[ , 'b': [2,4[ }}
```

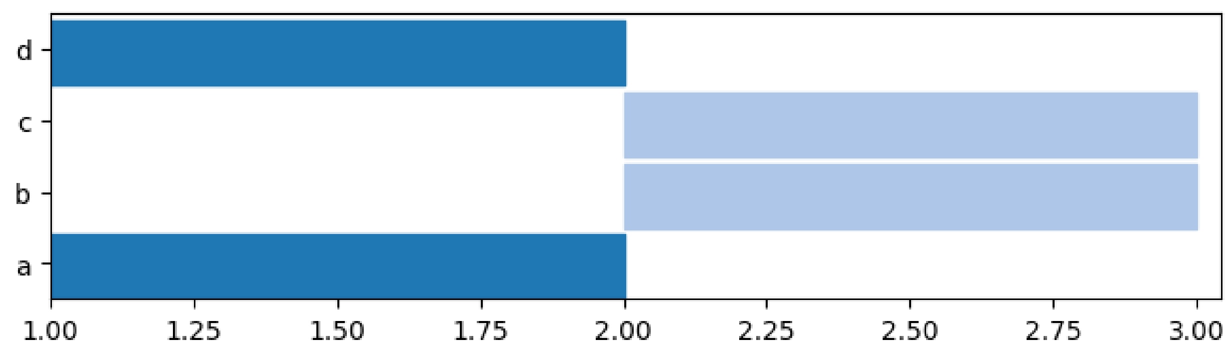
[13]:
```
converted_sn = com_ig.to_DynCommunitiesSN(slices=[(x,x+1) for x in range(1,4)])
print(converted_sn.communities())
print(com_sn.communities())
```

```
{'com1': {'a': [1], 'd': [1, 3]}, 'com2': {'b': [2, 3], 'c': [2, 3]}}
{'com1': {'a': [1], 'd': [1, 3]}, 'com2': {'c': [2, 3], 'b': [2, 3]}}
```

Slicing

Slicing part of networks can be useful, for instance to visualize only a fraction of a large dynamic partition

[14]:
```
sliced = com_ig.slice(start=1,end=3)
plot = tn.plot_longitudinal(communities=sliced,height=200)
```



[ ]:

2.3.4 Dynamic Communities detection and evaluation

If tnerwork library is not installed, you need to install it, for instance using the following command

[1]:
```
#%%capture #avoid printing output
#!pip install --upgrade git+https://github.com/Yquetzal/tnetwork.git
```

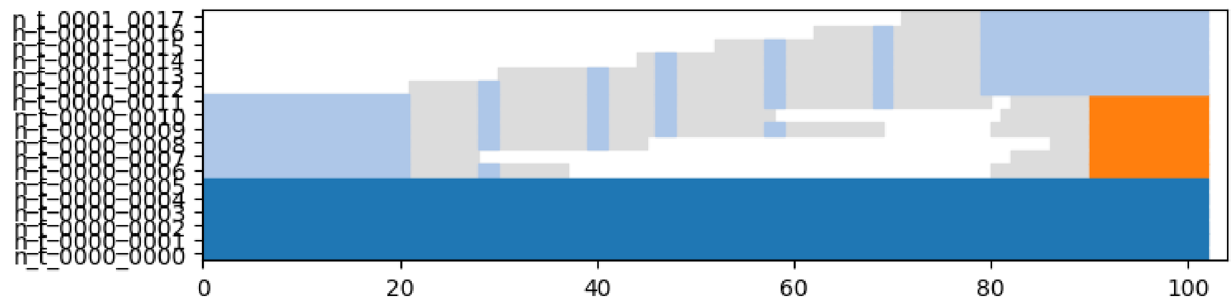
[2]:
```
%load_ext autoreload
%autoreload 2
import tnetwork as tn
import seaborn as sns
import pandas as pd
import networkx as nx
import numpy as np
```

Creating an example dynamic graph with changing community structure

We create a simple example of dynamic community evolution using the generator provided in the library. We generate a simple ship of Theseus scenario. Report to the corresponding tutorial to fully understand the generation part if needed.

[3]:
```
my_scenario = tn.ComScenario(alpha=0.8,random_noise=0.1)
[com1,com2] = my_scenario.INITIALIZE([6,6],["c1","c2"])
(com2,com3)=my_scenario.THESEUS(com2,delay=20)
my_scenario.CONTINUE(com3,delay=10)

#visualization
(generated_network_IG,generated_comunities_IG) = my_scenario.run()

plot = tn.plot_longitudinal(generated_network_IG,generated_comunities_IG,height=200)
generated_network_SN = generated_network_IG.to_DynGraphSN(slices=1)
generated_communities_SN = generated_comunities_IG.to_DynCommunitiesSN(slices=1)
```

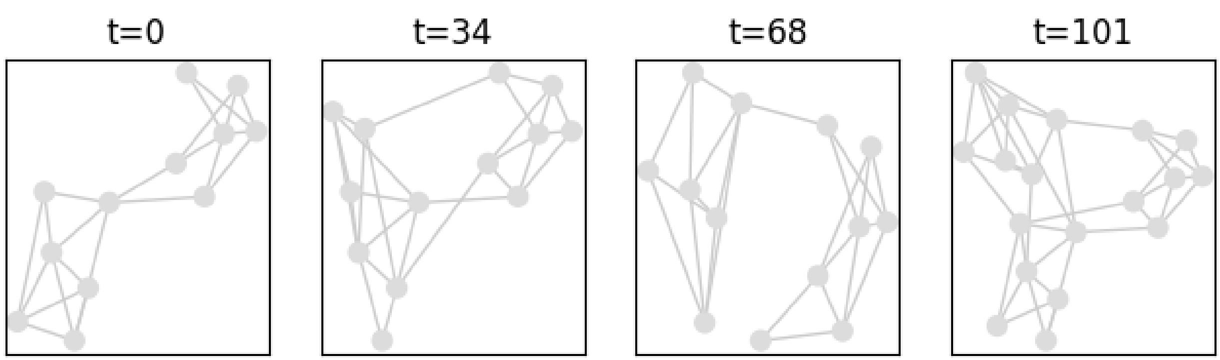
```
100% (8 of 8) |##########################| Elapsed Time: 0:00:00 ETA:  00:00:00/usr/
↪local/lib/python3.7/site-packages/numpy/core/numeric.py:2327: FutureWarning:␣
↪elementwise comparison failed; returning scalar instead, but in the future will␣
↪perform elementwise comparison
  return bool(asarray(a1 == a2).all())
```



Let's look at the graph at different stages. There are no communities.

```
[4]: last_time = generated_network_IG.end()
print(last_time)
times_to_plot = [0,int(last_time/3),int(last_time/3*2),last_time-1]
plot = tn.plot_as_graph(generated_network_IG,ts=times_to_plot,width=200,height=200)
```

```
102
```



Algorithms for community detection are located in the tnetwork.DCD package

```
[5]: import tnetwork.DCD as DCD
```

First algorithm: Iterative match

Iterative match consists in applying a static algorithm at each step and matching communities in successive snapshots if they are similar. Check the doc for more details.

Without particular parameters, it uses the louvain method and the jaccard coefficient.

```
[6]: com_iterative = DCD.iterative_match(generated_network_SN)
```

The static algorithm, the similarity function and the threashold to consider similar can be changed

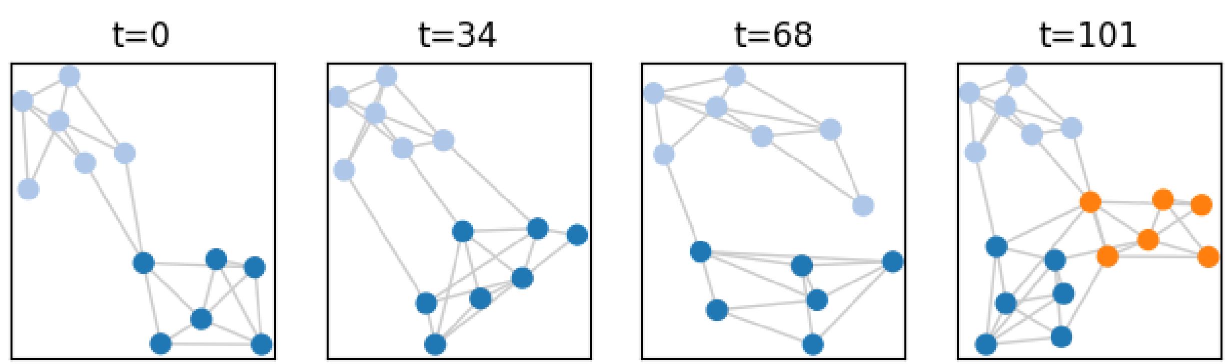
[7]:
```
custom_match_function = lambda x,y: len(x&y)/max(len(x),len(y))
com_custom = DCD.iterative_match(generated_network_SN,match_function=custom_match_
↪function,CDalgo=nx.community.greedy_modularity_communities,threshold=0.5)
```

Visualizing communities

One way to visualize the evolution of communities is to plot the graph at some snapshots. By calling the `plot_as_graph` function with several timestamps, we plot graphs at those timestamps while ensuring:

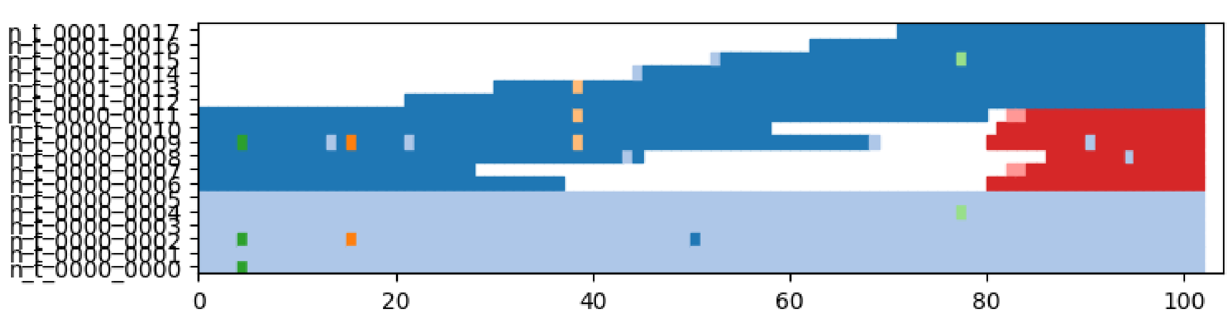
- That the position of nodes stay the same between snapshots
- That the same color in different plots means that nodes belong to the same dynamic communities

[8]:
```
last_time = generated_network_IG.end()
times_to_plot = [0,int(last_time/3),int(last_time/3*2),last_time-1]
plot = tn.plot_as_graph(generated_network_IG,com_iterative,ts=times_to_plot,auto_
↪show=True,width=200,height=200)
```



Another solution is to plot a longitudinal visualization: each horizontal line corresponds to a node, time is on the x axis, and colors correspond to communities. Grey means that a node corresponds to no community, white that the node is not present in the graph (or has no edges)

[9]:
```
to_plot = tn.plot_longitudinal(generated_network_SN,com_iterative,height=200)
```

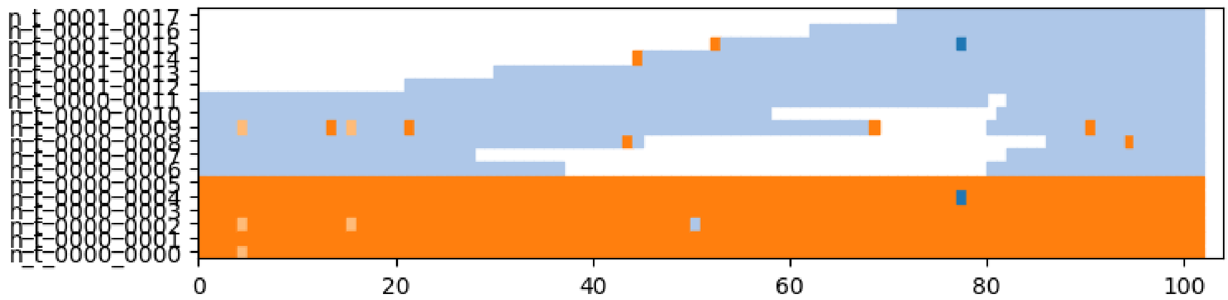


Survival Graph

This method matches communities not only between successive snaphsots, but between any snapshot, constituting a survival graph on which a community detection algorithm detects communities of communities => Dynamic communities

[10]:
```
com_survival = DCD.label_smoothing(generated_network_SN)
plot = tn.plot_longitudinal(generated_network_SN,com_survival,height=200)
```

```
starting label_smoothing method
```



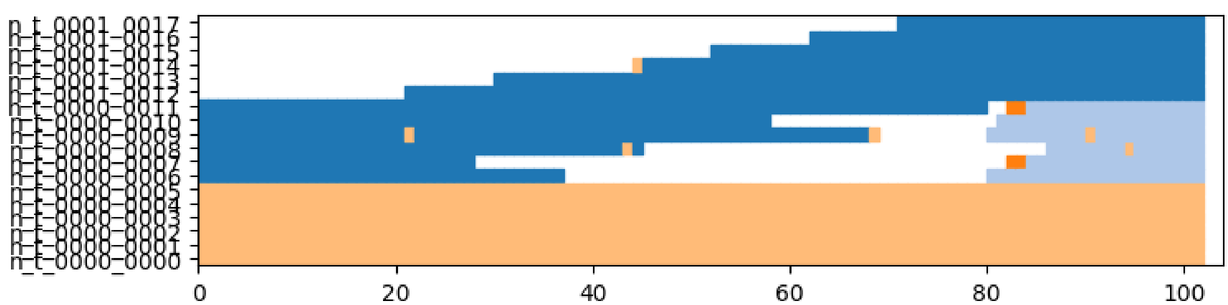
Smoothed louvain

The smoothed Louvain algorihm is very similar to the simple iterative match, at the difference that, at each step, it initializes the partition of the Louvain algorithm with the previous partition instead of having each node in its own community as in usual Louvain.

It has the same options as iterative match, since only the community detection process at each step changes, not the matching

[11]:
```
com_smoothed = DCD.smoothed_louvain(generated_network_SN)
plot = tn.plot_longitudinal(generated_network_SN,com_smoothed,height=200)
```

```
98% (100 of 102) |#################### | Elapsed Time: 0:00:00 ETA:   0:00:00
```

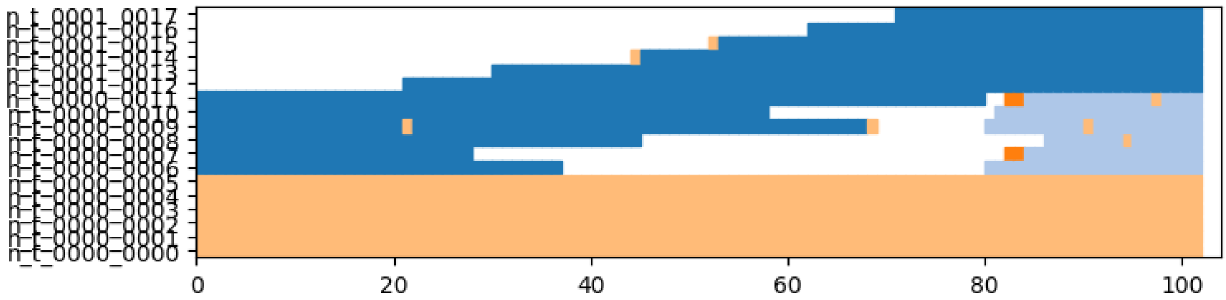


Smoothed graph

The smoothed-graph algorithm is similar to the previous ones, but the graph at each step is *smoothed* by the community structure found in the previous step. (An edge with a small weight is added between any pair of nodes that where in the same community previously. This weight is determined by a parameter `alpha`)

[12]:
```
com_smoothed_graph = DCD.smoothed_graph(generated_network_SN)
plot = tn.plot_longitudinal(generated_network_SN,com_smoothed_graph,height=200)
```

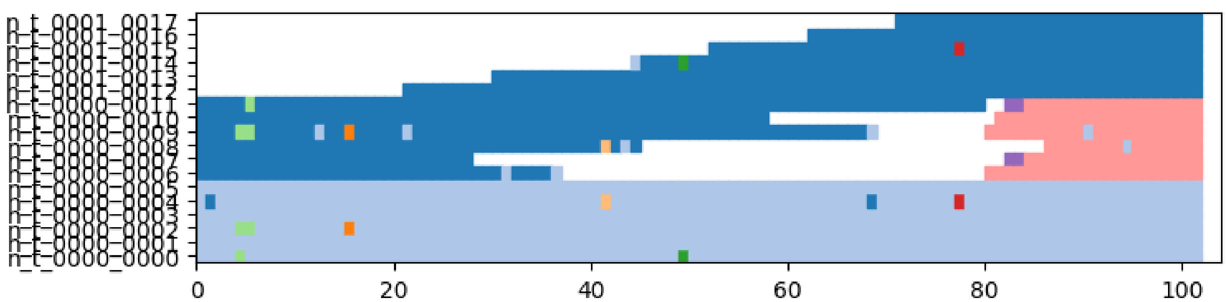
```
97% (99 of 102) |##################### | Elapsed Time: 0:00:00 ETA:   0:00:00
```



Matching with a custom function

The iterative match and survival graph methods can also be instantiated with any custom community detection algorithm at each step, and any matching function, as we can see below. The match function takes as input the list of nodes of both communities, while the community algorithm must follow the signature of networkx community detection algorithms

[13]:
```
custom_match_function = lambda x,y: len(x&y)/max(len(x),len(y))
com_custom2 = DCD.iterative_match(generated_network_SN,match_function=custom_match_
↪function,CDalgo=nx.community.greedy_modularity_communities)
plot = tn.plot_longitudinal(generated_network_SN,com_custom2,height=200)
```



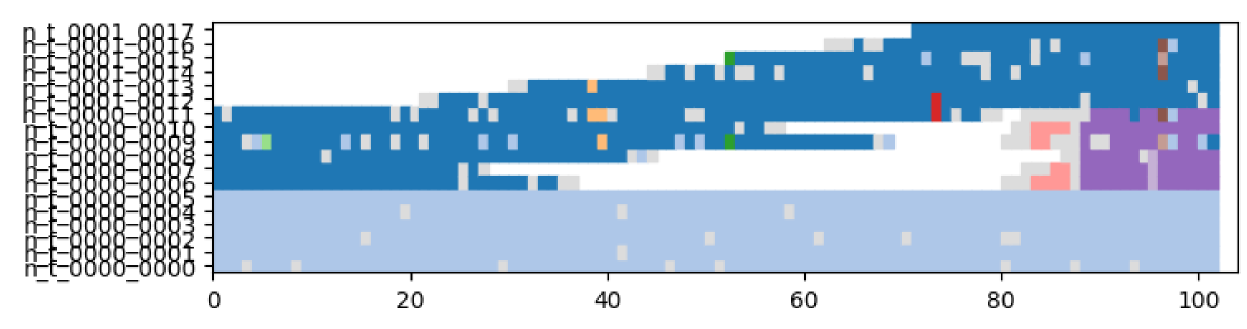
Another algoritm in python: CPM

CPM stands for Clique Percolation Method. An originality of this approach is that it yiealds overlapping communities.

Be careful, the visualization is not currently adapted to overlapping clusters. . .

[14]:
```
com_CPM = DCD.rollingCPM(generated_network_SN,k=3)
plot = tn.plot_longitudinal(generated_network_SN,com_CPM,height=200)
```

```
CD detection done 102
```

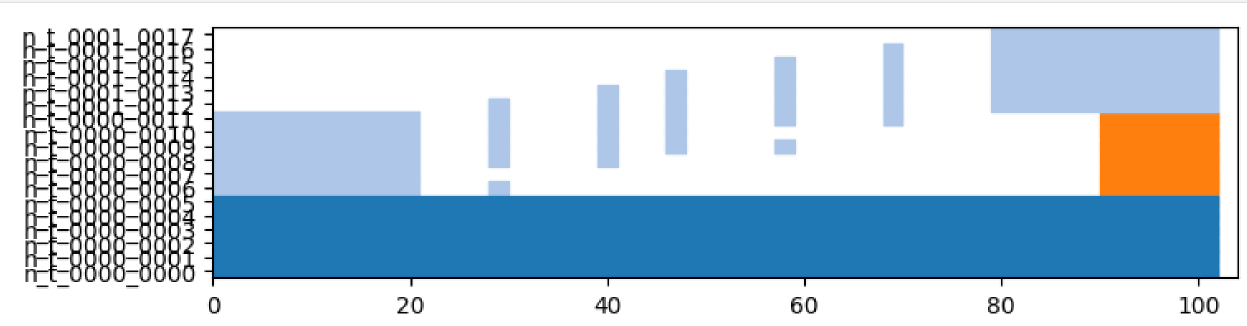


Dynamic partition evaluation

The goal of this section is to present the different types of dynamic community evalutation implemented in tnetwork.

For all evaluations below, no conclusion should be drawn about the quality of algorithms. . . .

[15]:
```
#Visualization
plot = tn.plot_longitudinal(communities=generated_comunities_IG,height=200,sn_
↪duration=1)
```



Quality at each step

The first type of evaluation we can do is simply to compute, at each type, a quality measure. By default, the method uses Modularity, but one can provide to the function its favorite quality function instead. It is the simplest adaptation of *internal evaluation*.

Note that * The result of an iterative approach is identical to the result of simply applying a static algorithm at each step * Smoothing therefore tends to lesser the scores. * The result migth or might not be computable at each step depending on the quality function used (e.g., modularity requires a complete partition of the networks to be computed)

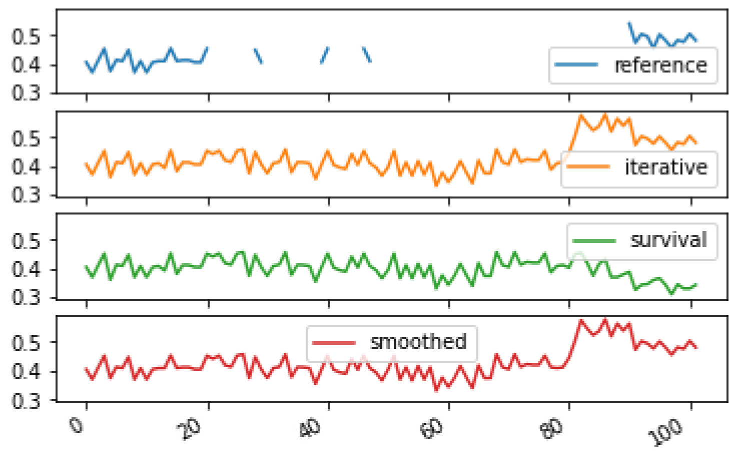
[16]:
```
quality_ref,sizes_ref = DCD.quality_at_each_step(generated_communities_SN,generated_
↪network_SN)
quality_iter,sizes_iter = DCD.quality_at_each_step(com_iterative,generated_network_SN)
quality_survival,sizes_survival = DCD.quality_at_each_step(com_survival,generated_
↪network_SN)
quality_smoothed,sizes_smoothed = DCD.quality_at_each_step(com_smoothed,generated_
↪network_SN)

df = pd.DataFrame({"reference":quality_ref,"iterative":quality_iter,"survival":
↪quality_survival,"smoothed":quality_smoothed})
```

(continues on next page)

(continued from previous page)

```
df.plot(subplots=True,sharey=True)
```

[16]:
```
array([<matplotlib.axes._subplots.AxesSubplot object at 0x11f1bd8d0>,
       <matplotlib.axes._subplots.AxesSubplot object at 0x11f5aa6d0>,
       <matplotlib.axes._subplots.AxesSubplot object at 0x11e993e10>,
       <matplotlib.axes._subplots.AxesSubplot object at 0x108343d10>],
      dtype=object)
```



Average values

One can of course compute average values over all steps. Be careful however when interpreting such values, as there are many potential biases: * Some scores (such as modularity) are not comparable between graphs of different sizes/density, so averaging values obtained on different timesteps might be incorrect * The *clarity* of the community structure might not be homogeneous, and your score might end up depending mostly on results on a specific period * Since the number of nodes change in every step, we have the choice of weighting the values by the size of the network * etc.

Since the process is the same for all later functions, we won't repeat it for the others in this tutorial

[17]:
```
print("iterative=", np.average(quality_iter),"weighted:", np.average(quality_iter,
↪weights=sizes_iter))
print("survival=", np.average(quality_survival),"weighted:", np.average(quality_
↪survival,weights=sizes_survival))
print("smoothed=", np.average(quality_smoothed),"weighted:", np.average(quality_
↪smoothed,weights=sizes_smoothed))
```

```
iterative= 0.4289862014179952 weighted: 0.4357461539951767
survival= 0.39927872978552464 weighted: 0.39689292217118277
smoothed= 0.42992554634769103 weighted: 0.4365993079467363
```

Similarity at each step

A second type of evaluation consists in adaptating *external evaluation*, i.e., comparison with a known reference truth.

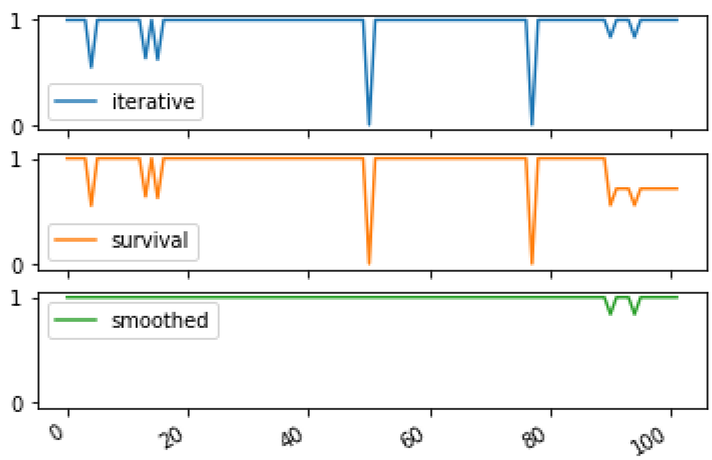
It simply computes at each step the similarity between the computed communities and the ground truth. By default, the function uses the Adjusted Mutual Information (AMI or aNMI), but again, any similarity measure can be provided to the function.

Note that, as for quality at each step, smoothing is not an advantage, community identities accross steps has no impact.

There is a subtility here: since, often, the dynamic ground truth might have some nodes without affiliations, we make the choice of comparing only what is known in the ground truth, i.e., if only 5 nodes out of 10 have a community in the ground truth at time t, the score of the proposed solution will depends only on those 5 nodes, and the affiliations of the 5 others is ignored

[18]:
```
quality_iter,sizes = DCD.similarity_at_each_step(generated_communities_SN,com_
↪iterative)
quality_survival,sizes = DCD.similarity_at_each_step(generated_communities_SN,com_
↪survival)
quality_smoothed,sizes = DCD.similarity_at_each_step(generated_communities_SN,com_
↪smoothed)

df = pd.DataFrame({"iterative":quality_iter,"survival":quality_survival,"smoothed":
↪quality_smoothed})
df.plot(subplots=True,sharey=True)
```

[18]:
```
array([<matplotlib.axes._subplots.AxesSubplot object at 0x11fb59290>,
       <matplotlib.axes._subplots.AxesSubplot object at 0x11f90ccd0>,
       <matplotlib.axes._subplots.AxesSubplot object at 0x11eb31c50>],
      dtype=object)
```



Smoothness Evaluation

We can evaluate the smoothness of a partition by comparing how the partition in each step is similar to the partition in the next. Again, any measure can be used, by default the overlapping NMI, because two adjacent partitions do not necessarily have the same nodes. * This evaluation is *internal*. * This time, it depends on the *labels* given to nodes accross steps, so a static algorithm applied at each step would have a score of zero. * The score does not depends at all on the quality of the solution, i.e., having all nodes in the same partition at every step would obtain a perfect score of 1

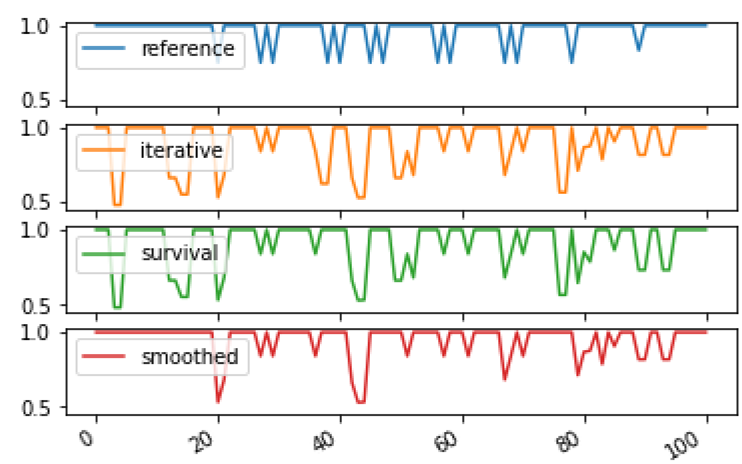
[19]:
```
quality_ref,sizes_ref = DCD.consecutive_sn_similarity(generated_communities_SN)
quality_iter,sizes_iter = DCD.consecutive_sn_similarity(com_iterative)
quality_survival,sizes_survival = DCD.consecutive_sn_similarity(com_survival)
quality_smoothed,sizes_smoothed = DCD.consecutive_sn_similarity(com_smoothed)
```

(continues on next page)

(continued from previous page)

```
df = pd.DataFrame({"reference":quality_ref,"iterative":quality_iter,"survival":
↪quality_survival,"smoothed":quality_smoothed})
df.plot(subplots=True,sharey=True)
```

[19]:
```
array([<matplotlib.axes._subplots.AxesSubplot object at 0x11f103850>,
       <matplotlib.axes._subplots.AxesSubplot object at 0x11c7af710>,
       <matplotlib.axes._subplots.AxesSubplot object at 0x11fc5c7d0>,
       <matplotlib.axes._subplots.AxesSubplot object at 0x11f46e610>],
      dtype=object)
```



Global scores

Another family of scores we can compute are not based on step by step computations, but rather compute directly a single score on whole communities

Longitudinal Similarity

This score is computed using a usual similarity measure, by default the AMI. But instead of computing the score for each step independently, it is computed once, consider each (node,time) pair as a data point (instead of each node in a static network). * The evaluation is *external*, it requires a (longitudinal) reference partition * It takes into account both the similarity at each step and the labels accros steps * Similar to step by step similarity, only (node,time) couples with a known affiliation in the reference partition are used, others are ignored

[20]:
```
quality_iter = DCD.longitudinal_similarity(generated_communities_SN,com_iterative)
quality_survival = DCD.longitudinal_similarity(generated_communities_SN,com_survival)
quality_smoothed = DCD.longitudinal_similarity(generated_communities_SN,com_smoothed)

print("iterative: ",quality_iter)
print("survival: ",quality_survival)
print("smoothed: ",quality_smoothed)
```

```
iterative:  0.9451292907933111
survival:  0.8234124633781458
smoothed:  0.9868504021347683
```

Global Smoothness

Trhee methods are proposed to evaluate the smoothness at the global level.

The first is the average value of partition smoothness as presented earlier, and is called `SM-P` for Partition Smoothness

The second one computes how many changes in affiliation there are, and the score `SM-N` (Node Smoothness) is 1/number of changes * It penalizes methods with many *glitches*, i.e., transient affiliation change. * It does not penalize long term changes

The third computes instead the entropy per node, and the score `SM-L` (Label smoothness) is 1/average node entropy. * It does not penalize much glitches * It advantages solutions in which nodes tend to belong to few communities

For all 3 scores, higher is better.

[21]:
```
print("iterative: SM-P" ,DCD.SM_P(com_iterative), "SM-N:",DCD.SM_N(com_iterative), "␣
↪SM-L:",DCD.SM_L(com_iterative))
print("survival: SM-P ",DCD.SM_P(com_survival), "SM-N:",DCD.SM_N(com_survival), " SM-
↪L:",DCD.SM_L(com_survival))
print("smoothed: SM-P:",DCD.SM_P(com_smoothed), "SM-N:",DCD.SM_N(com_smoothed), " SM-
↪L:",DCD.SM_L(com_smoothed))
```

```
iterative: SM-P 0.9001839896381273 SM-N: 0.023255813953488372  SM-L: 3.
↪6914110221883676
survival: SM-P  0.9026384453495243 SM-N: 0.03333333333333333  SM-L: 18.48733611718878
smoothed: SM-P: 0.9470754696907387 SM-N: 0.05555555555555555  SM-L: 4.416478672484498
```

[ ]:
```
```

2.3.5 Generation of dynamic networks with communities

Table of Contents

- *RESURGENCE*
- *Ship of Theseus*
- *CONTINUE*
- *Custom Event: ASSIGN*

4. *Generating random scenarios*
5. *Mixing parameters*

[1]:
```
#%%capture #avoid printing output
#!pip install --upgrade git+https://github.com/Yquetzal/tnetwork.git
```

[2]:
```
%load_ext autoreload
%autoreload 2

import tnetwork as tn
import numpy as np
```

Introduction: simple generation

The generation process works in 2 phases: 1. Define the scenario that you want 2. Run the generation

Everything is done on a community scenario `ComScenario` instance

[3]:
```
#First, we create an instance of community scenario
my_scenario = tn.ComScenario()
```

Initialization

We can define the original community structure. We set the size of communities and, optionnaly, their names. The function returns objects that represent those communities

[4]:
```
[com1,com2] = my_scenario.INITIALIZE([4,6],["com1","com2"])
```

As soon as we have declared those communities, we can check their number of nodes `n` and number of internal edges `m`. The number of edges is automatically determined by a density function that depends on the size of the community and a global parameter that can be specified when creating the scenario, more on that in the *mixing parameters* section

[5]:
```
print(com1)
print(com2)
```

```
(com1:n=4,m=5)
(com2:n=6,m=11)
```

Merge

Let's define a first operation on these communities. It will be a merge operation, using the function `MERGE`

[6]:
```
#We merge com1 and com2.
absorbing = my_scenario.MERGE([com1,com2],"merged")
```

Run

To better understand what is going on, let's run the generation, by calling the function `run`. This has two consequences: 1. It generates a network corresponding to the described community structure 2. It fixes the details of the number of steps required to do an operation. This is not known in advance, since it depends on a stochastic process

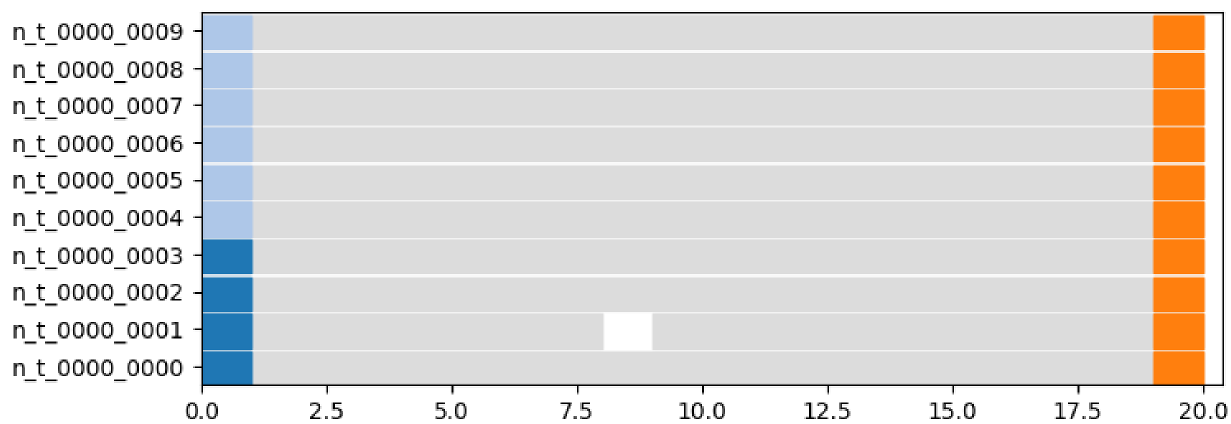
[7]:
```
(generated_network,generated_comunities) = my_scenario.run()
```

```
100% (1 of 1) |##########################| Elapsed Time: 0:00:00 ETA:  00:00:00
```

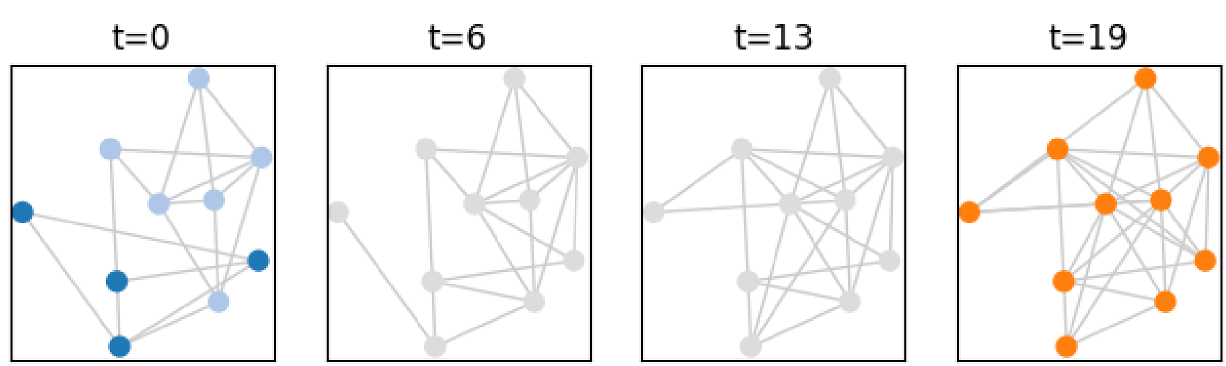
We can now plot the community structre and the state of the graphs at some times. We can observe that: since the merge is progressive, nodes belong to no community while the operation is in progress (grey color). We can also observe the topology of the graph evolving from two communities to one.

[8]:
```
plot = tn.plot_longitudinal(generated_network,generated_comunities,height=300)
```

```
/usr/local/lib/python3.7/site-packages/numpy/core/numeric.py:2327: FutureWarning:␣
↪elementwise comparison failed; returning scalar instead, but in the future will␣
↪perform elementwise comparison
  return bool(asarray(a1 == a2).all())
```



[9]:
```
last_time = generated_network.end()
times_to_plot = [0,int(last_time/3),int(last_time/3*2),last_time-1]
plot = tn.plot_as_graph(generated_network,generated_comunities,ts=times_to_plot,auto_
↪show=True,width=200,height=200)
```



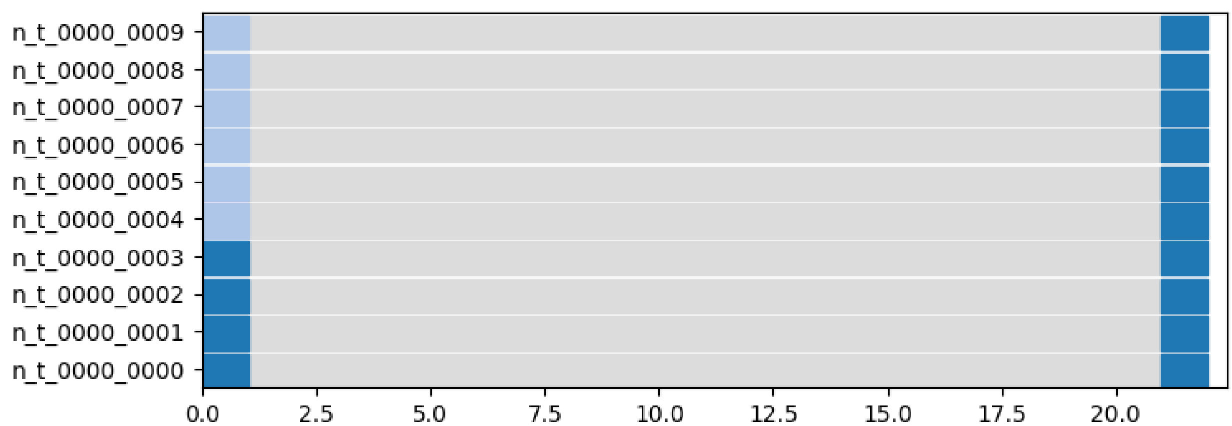
Conservation of identity of communities

Note that the label/name we give to communities is important, it corresponds to their *identity*, i.e., two communities with the same label have the same identity (=same community).

If we reuse the same scenario, only changing the label of the merged community from "merged" to "com1", we observe in the visualization that the community after the merge has now the same color (i.e., is "the same community") as one of the original ones.

[10]:
```
my_scenario = tn.ComScenario()
[com1,com2] = my_scenario.INITIALIZE([4,6],["com1","com2"])
absorbing = my_scenario.MERGE([com1,com2],"com1")
(generated_network,generated_comunities) = my_scenario.run()
plot = tn.plot_longitudinal(generated_network,generated_comunities,height=300)
```

```
100% (1 of 1) |#########################| Elapsed Time: 0:00:00 ETA:  00:00:00
```



Events chaining

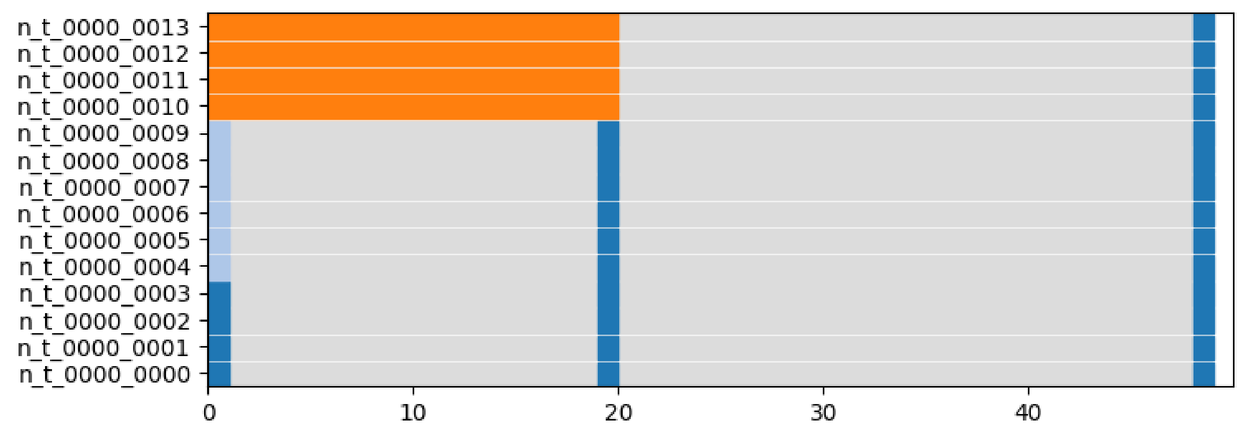
Several options are available to control the chaining of operations.

Natural chaining

First, each operation takes some communities as input. In order for the event to start, the communities required in input must be ready.

[11]:
```
my_scenario = tn.ComScenario()
[com1,com2,com3] = my_scenario.INITIALIZE([4,6,4],["c1","c2","c3"])
absorbing = my_scenario.MERGE([com1,com2],"c1")
absorbing = my_scenario.MERGE([absorbing,com3],"c1")

(generated_network,generated_comunities) = my_scenario.run()
plot = tn.plot_longitudinal(generated_network,generated_comunities,height=300)
```

```
100% (2 of 2) |#########################| Elapsed Time: 0:00:00 ETA:  00:00:00
```

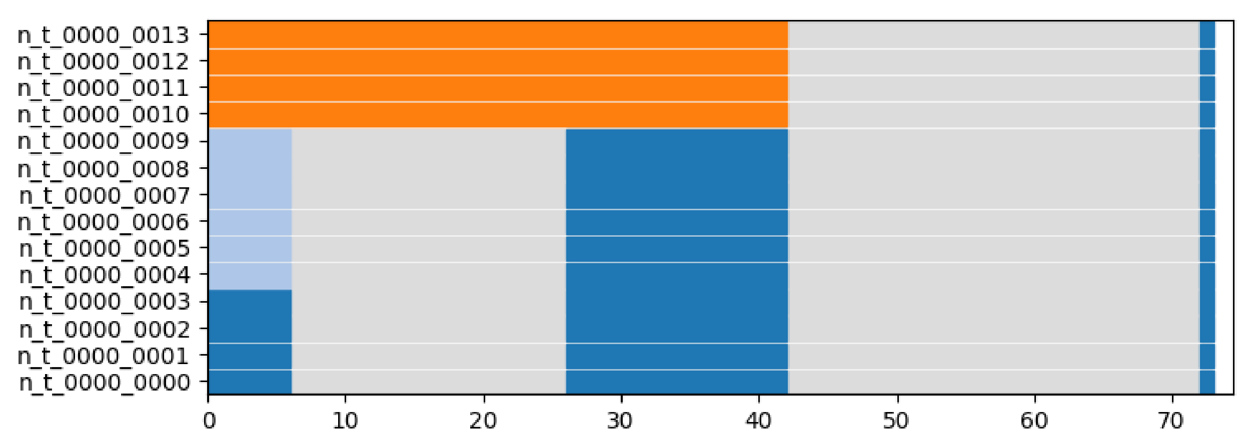


Fix delay

It is possible to explicitely require to wait for a given period before starting the event using the `delay` argument of any event

[12]:
```
my_scenario = tn.ComScenario()
[com1,com2,com3] = my_scenario.INITIALIZE([4,6,4],["c1","c2","c3"])
absorbing = my_scenario.MERGE([com1,com2],"c1",delay=5)
absorbing = my_scenario.MERGE([absorbing,com3],"c1",delay=15)

(generated_network,generated_comunities) = my_scenario.run()
plot = tn.plot_longitudinal(generated_network,generated_comunities,height=300)
```

```
100% (2 of 2) |########################| Elapsed Time: 0:00:00 ETA:  00:00:00
```



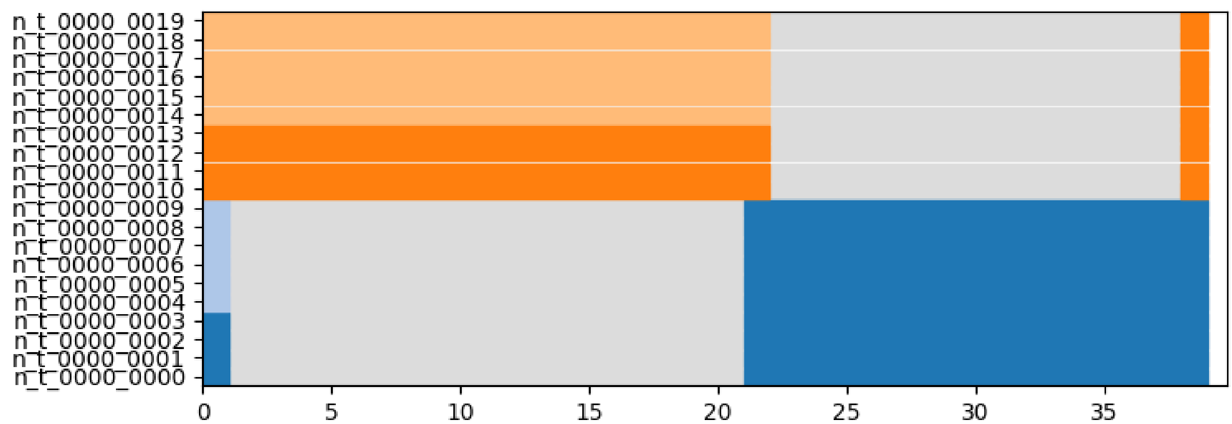
Triggers

One can also use triggers to define that an event can start only when another (unrelated) operations finished. This can be done using the keywork `triggers`.

In the following example, the second merge, completely unrelated to the first one, is triggered by its end

[13]:
```
my_scenario = tn.ComScenario()
[com1,com2,com3,com4] = my_scenario.INITIALIZE([4,6,4,6],["c1","c2","c3","c4"])
absorbing1 = my_scenario.MERGE([com1,com2],"c1")
absorbing2 = my_scenario.MERGE([com3,com4],"c3",triggers=[absorbing1])

(generated_network,generated_comunities) = my_scenario.run()
plot = tn.plot_longitudinal(generated_network,generated_comunities,height=300)
```

```
100% (2 of 2) |#########################| Elapsed Time: 0:00:00 ETA:  00:00:00
```



Events

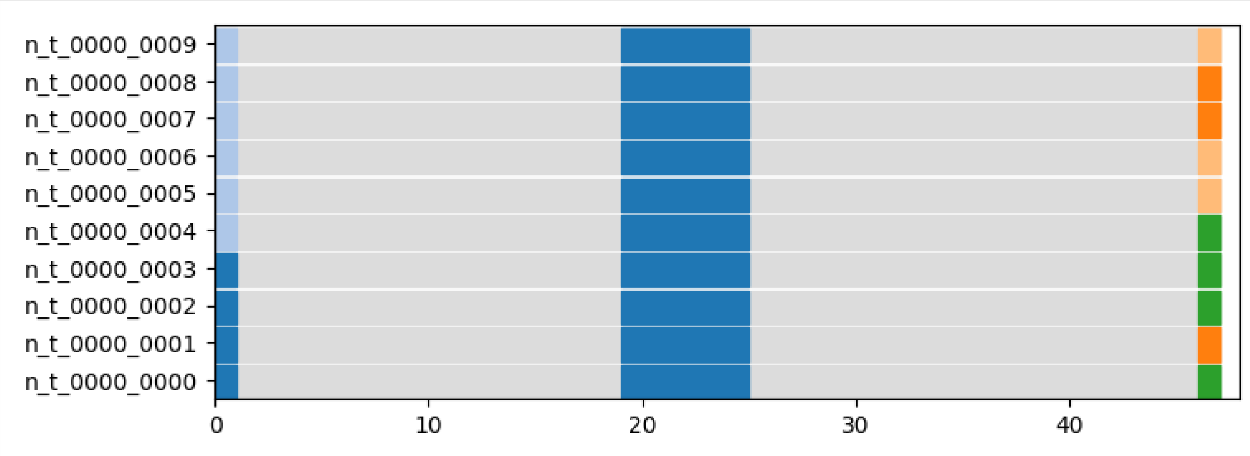
Let's now go through the different existing events

MERGE/SPLIT

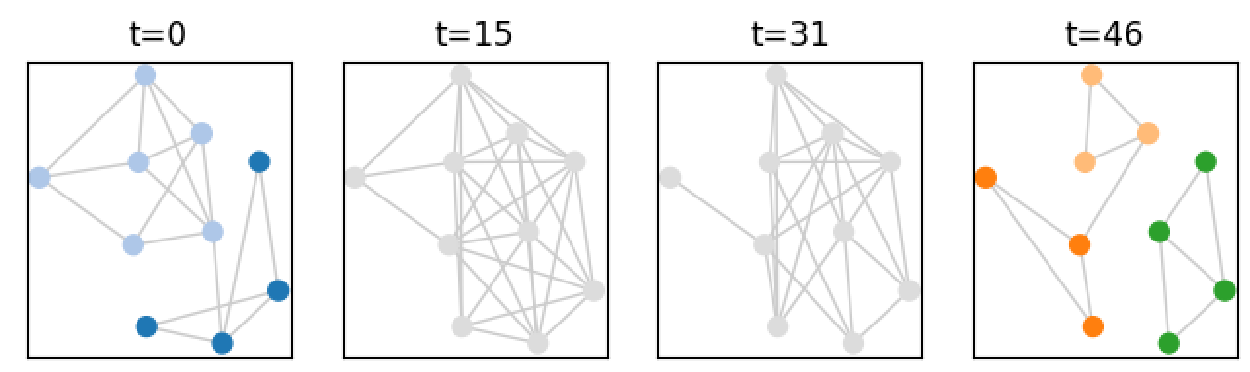
We have alredy seen the `MERGE` event, there is a symmetric `SPLIT` event.

[14]:
```
my_scenario = tn.ComScenario()
[com1,com2] = my_scenario.INITIALIZE([4,6],["c1","c2"])
merged = my_scenario.MERGE([com1,com2],"c1")
my_scenario.SPLIT(merged,["split1","split2","split3"],[3,3,4],delay=5)

(generated_network,generated_comunities) = my_scenario.run()
plot = tn.plot_longitudinal(generated_network,generated_comunities,height=300)
```

```
100% (2 of 2) |#########################| Elapsed Time: 0:00:00 ETA:  00:00:00
```



[15]:
```
last_time = generated_network.end()
times_to_plot = [0,int(last_time/3),int(last_time/3*2),last_time-1]
plot = tn.plot_as_graph(generated_network,generated_comunities,ts=times_to_plot,auto_
↪show=True,width=200,height=200)
```

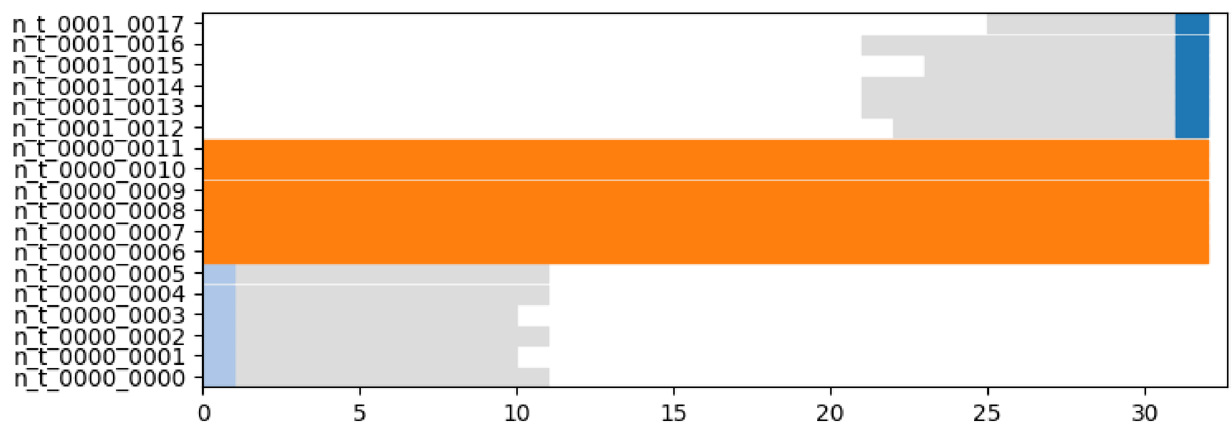


BIRTH/DEATH

Communities can appear and disappear. Note that communities appear progressively, edge by edge.

[16]:
```
my_scenario = tn.ComScenario()
[com1,com2] = my_scenario.INITIALIZE([6,6],["c1","c2"])
my_scenario.BIRTH(6,"born",delay=20)
my_scenario.DEATH(com1)

#visualization
(generated_network,generated_comunities) = my_scenario.run()
plot = tn.plot_longitudinal(generated_network,generated_comunities,height=300)
```

```
100% (2 of 2) |##########################| Elapsed Time: 0:00:00 ETA:  00:00:00
```



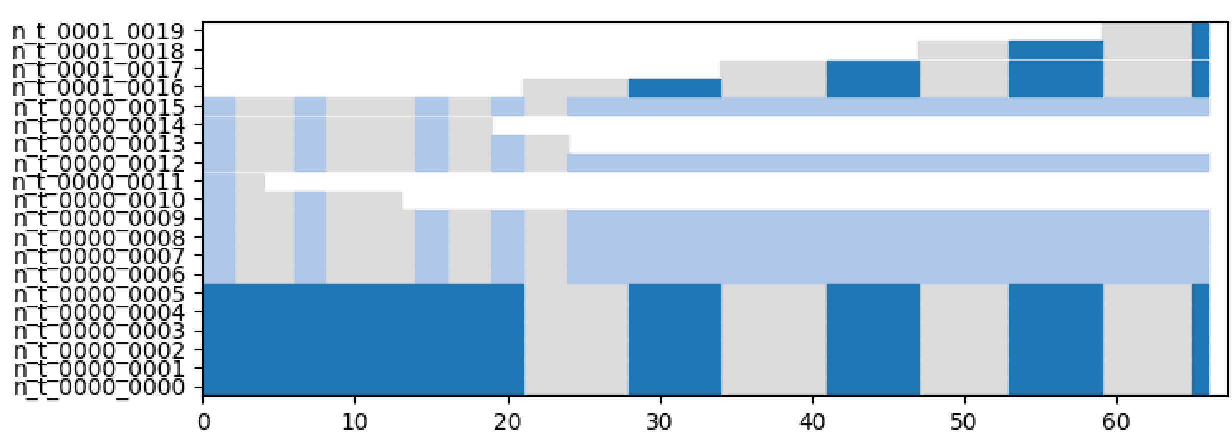
Iterative GROW/SHRINK

It is possible to make a community grow (creating new nodes) or shring (nodes disappear), one node after the other, node by node. It can be used to add/remove a single node too, of course.

A parameter allow to tune the time between each addition/removal

[17]:
```
my_scenario = tn.ComScenario()
[com1,com2] = my_scenario.INITIALIZE([6,10],["c1","c2"])
my_scenario.GROW_ITERATIVE(com1,nb_nodes2Add=4,wait_step=5,delay=20)
my_scenario.SHRINK_ITERATIVE(com2,nb_nodes2remove=4,wait_step=1)

#visualization
(generated_network,generated_comunities) = my_scenario.run()
plot = tn.plot_longitudinal(generated_network,generated_comunities,height=300)
```

```
100% (8 of 8) |##########################| Elapsed Time: 0:00:00 ETA:  00:00:00
```

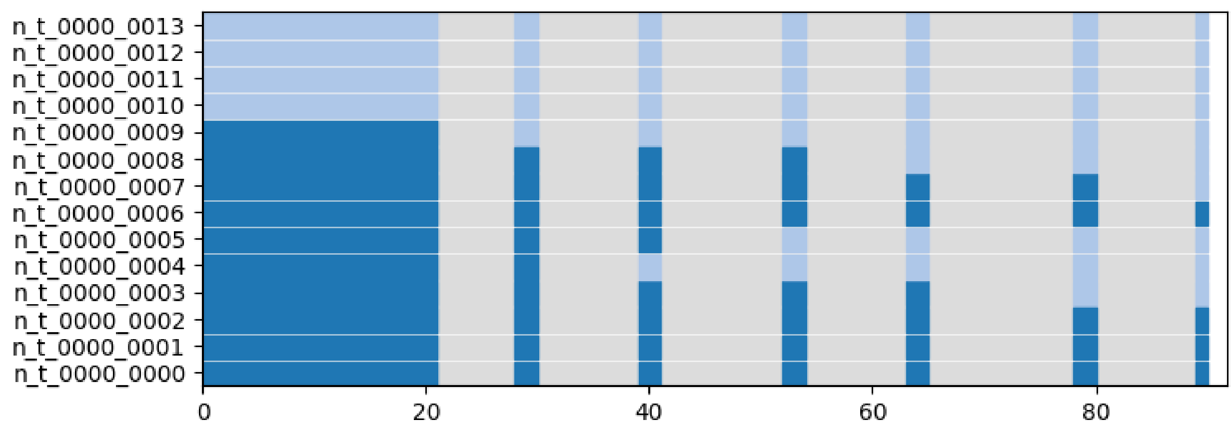


Iterative node MIGRATION

Most of the time, in the real world, when a community change size, it is not by integrating nodes newly created, but by taking nodes from existing communities. This is one this event corresponds to: nodes are moving from one community to another one, one after the other

[18]:
```
my_scenario = tn.ComScenario()
[com1,com2] = my_scenario.INITIALIZE([10,4],["c1","c2"])
my_scenario.MIGRATE_ITERATIVE(com1,com2,6,wait_step=1,delay=20)

#visualization
(generated_network,generated_comunities) = my_scenario.run()
plot = tn.plot_longitudinal(generated_network,generated_comunities,height=300)
```

```
100% (6 of 6) |#########################| Elapsed Time: 0:00:00 ETA:  00:00:00
```

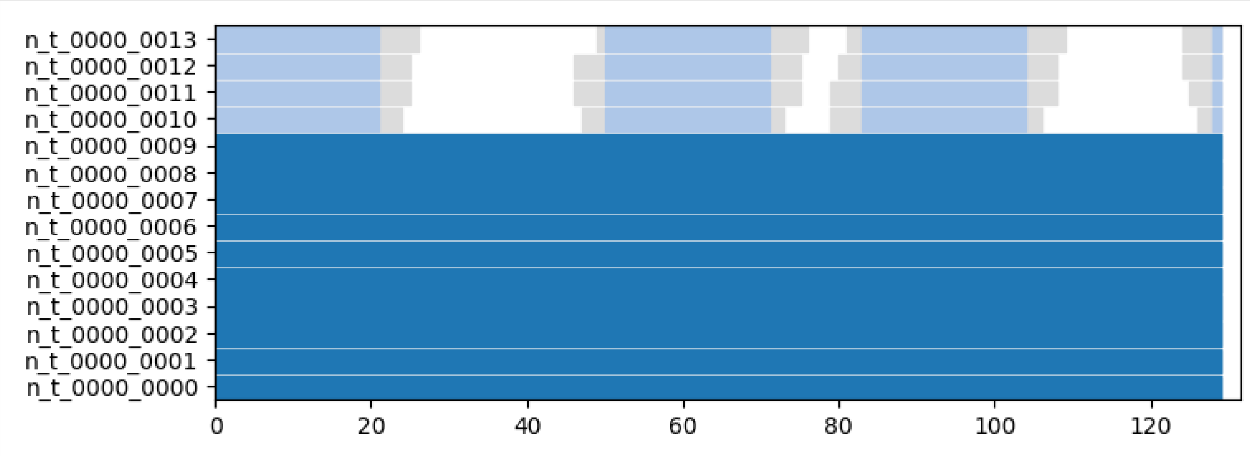


RESURGENCE

Resurgence is a type of event in which a community disappear for some time, and reappear later, identical to its state before the disappearance. Think of seasonal events for instance, with groups of people/animals/keywords observed together at regular periods.

[19]:
```
my_scenario = tn.ComScenario()
[com1,com2] = my_scenario.INITIALIZE([10,4],["c1","c2"])
com2 = my_scenario.RESURGENCE(com2,death_period=20,delay=20)
com2 = my_scenario.RESURGENCE(com2,death_period=3,delay=20)
my_scenario.RESURGENCE(com2,death_period=15,delay=20)



#visualization
(generated_network,generated_comunities) = my_scenario.run()
plot = tn.plot_longitudinal(generated_network,generated_comunities,height=300)
```

```
100% (6 of 6) |#########################| Elapsed Time: 0:00:00 ETA:  00:00:00
```



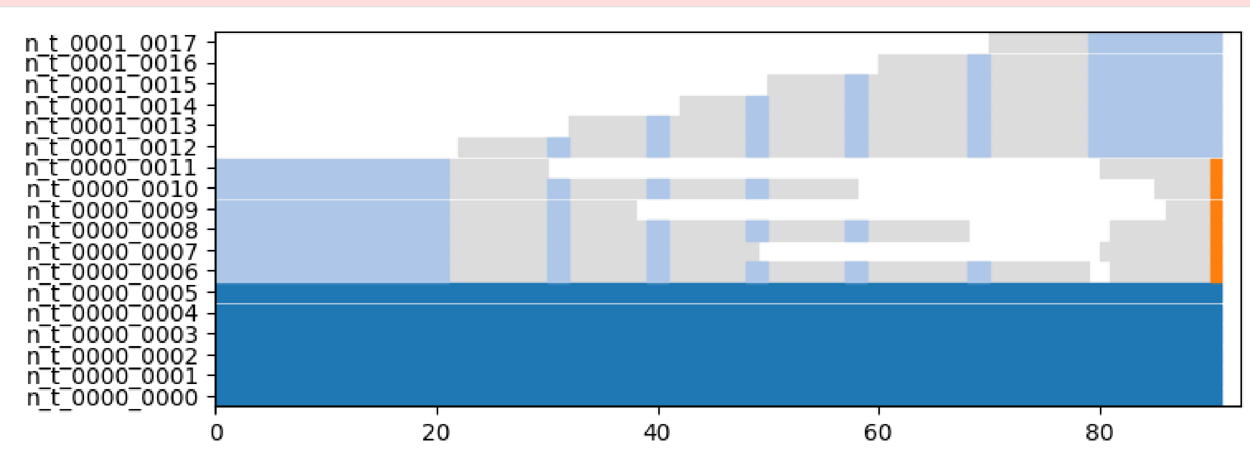
Ship of theseus

The ship of theseus is a typical example of the problem of community identity attribution: starting with a community A, all the nodes are replaced by new ones, one after the other, until none of the original remains. A new community B then appears with exactly the same nodes as the ones originally composing A. Which one is the *correct* A, the community currently labeled A but having no node in common with the original state of A, or the one labelled B ?

[20]:

```
my_scenario = tn.ComScenario()
[com1,com2] = my_scenario.INITIALIZE([6,6],["c1","c2"])
my_scenario.THESEUS(com2,delay=20)

#visualization
(generated_network,generated_comunities) = my_scenario.run()
plot = tn.plot_longitudinal(generated_network,generated_comunities,height=300)
```

```
100% (7 of 7) |#########################| Elapsed Time: 0:00:00 ETA:  00:00:00
```

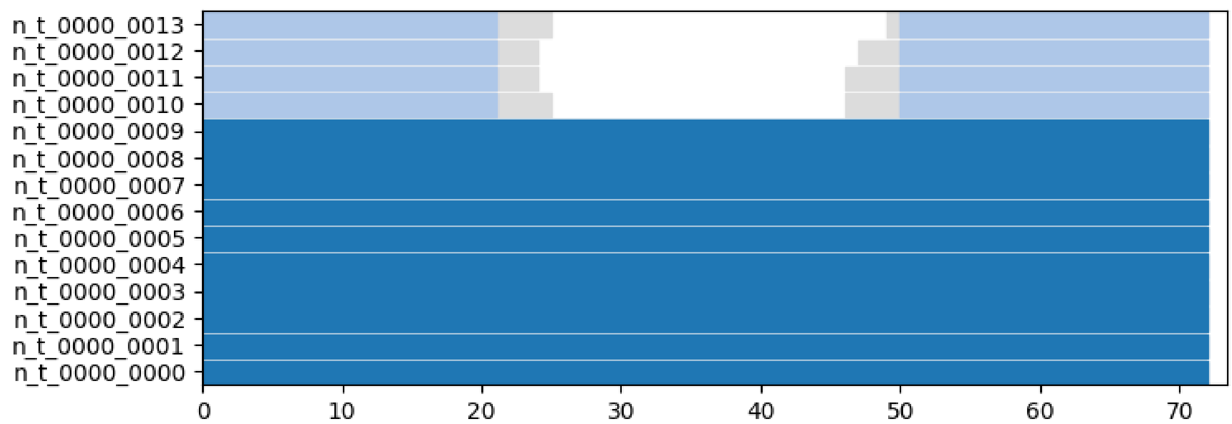


CONTINUE

The CONTINUE event allows to define a period without change for a community. It is mostly useful to add some period without any change at the end of the scenario.

[21]:
```
my_scenario = tn.ComScenario()
[com1,com2] = my_scenario.INITIALIZE([10,4],["c1","c2"])
com2 = my_scenario.RESURGENCE(com2,death_period=20,delay=20)
my_scenario.CONTINUE(com2,delay=20)



#visualization
(generated_network,generated_comunities) = my_scenario.run()
plot= tn.plot_longitudinal(generated_network,generated_comunities,height=300)
```

```
100% (3 of 3) |#########################| Elapsed Time: 0:00:00 ETA:  00:00:00
```



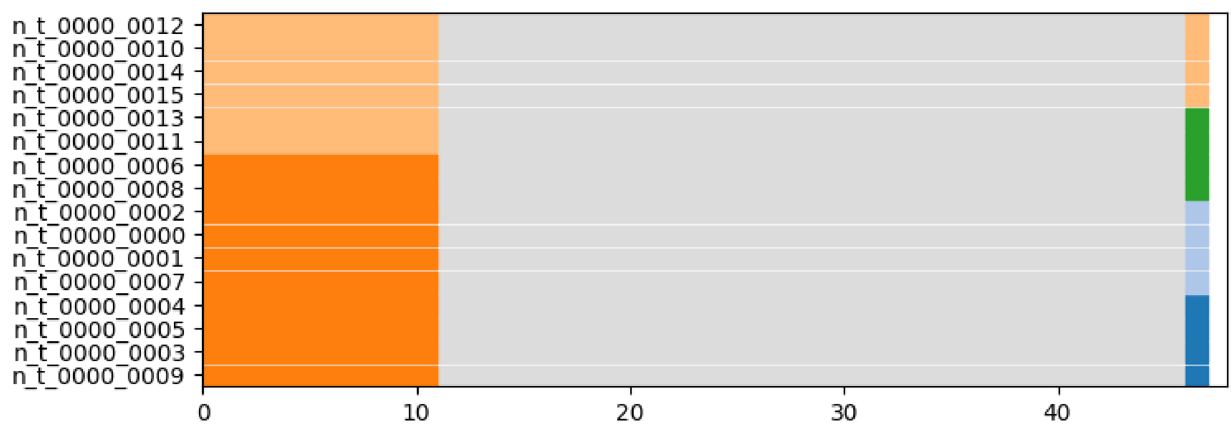
Custom event: ASSIGN

Most typical scenarios can be described by combining events described above. However, real community evolution might be even more complex than that. For instance, a community of 10 nodes might split in 2 communities of size 4, while 2 of its nodes merge with two nodes leaving another community to create a new community !

We can define any such scenario using the ASSIGN event. Note that in this case, we have to take care of a lower level and describe the event *node by node*

[22]:
```
my_scenario = tn.ComScenario()
[com1,com2] = my_scenario.INITIALIZE([10,6],["c1","c2"])
nodesC1 = list(com1.nodes())
nodesC2 = list(com2.nodes())
new_split = [nodesC1[:4],nodesC1[4:8],nodesC1[8:10]+nodesC2[:2],nodesC2[2:]]
my_scenario.ASSIGN(comsBefore=[com1,com2],comsAfter=["C1_split1","C1_split2","new_com
↪","c2"],splittingOut=new_split,delay=10)



#visualization
(generated_network,generated_comunities) = my_scenario.run()
plot = tn.plot_longitudinal(generated_network,generated_comunities,height=300,
↪nodes=nodesC1+nodesC2)
```

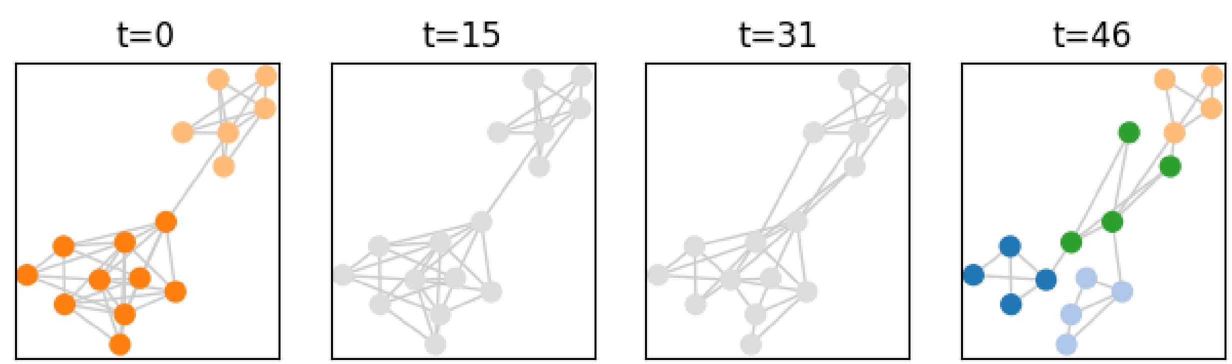
```
100% (1 of 1) |#########################| Elapsed Time: 0:00:00 ETA:  00:00:00
```



Let's check that the generated network structure do match the described community structure:

[23]:

```
last_time = generated_network.end()
times_to_plot = [0,int(last_time/3),int(last_time/3*2),last_time-1]
plot = tn.plot_as_graph(generated_network,generated_comunities,ts=times_to_plot,auto_
↪show=True,width=200,height=200)
```



Generating random scenarios

In what we have seen until now, the scenario was generated manually, by describing precisely the chaining of events.

In typical benchmarks, we want more flexibility, and generate several scenarios with random variations. This can easily been done by writing some code, as examplified below. Of course, all choices made have consequences, but the goal of this benchmark is to provide the atomic tools to provide good high level generators...

[24]:

```
def generate_graph(nb_com =6,min_size=4,max_size=15,operations=10,mu=0.1):
    print("generating graph with nb_com = ",nb_com)
    prog_scenario = tn.ComScenario(verbose=False,external_density_penalty=mu)
    all_communities = set(prog_scenario.INITIALIZE(np.random.randint(min_size,max_
↪size,size=nb_com)))

    for i in range(operations):
        [com1] = np.random.choice(list(all_communities),1,replace=False)
        all_communities.remove(com1)
```

(continues on next page)

(continued from previous page)

```
        if len(com1.nodes())<max_size and len(all_communities)>0: #merge
            [com2] = np.random.choice(list(all_communities),1,replace=False)
            largest_com = max([com1,com2],key=lambda x: len(x.nodes()))
            merged = prog_scenario.MERGE([com1,com2],largest_com.label(),delay=20)
            all_communities.remove(com2)
            all_communities.add(merged)
        else: #split
            smallest_size = int(len(com1.nodes())/3)
            (com2,com3) = prog_scenario.SPLIT(com1,[prog_scenario._get_new_ID("CUSTOM
↪"),com1.label()],[smallest_size,len(com1.nodes())-smallest_size],delay=20)
            all_communities|= set([com2,com3])
    (dyn_graph,dyn_com) = prog_scenario.run()


    return(dyn_graph,dyn_com)
```

[25]:

```
(generated_network,generated_comunities) = generate_graph(nb_com=6,max_size=10,
↪operations=10)
```

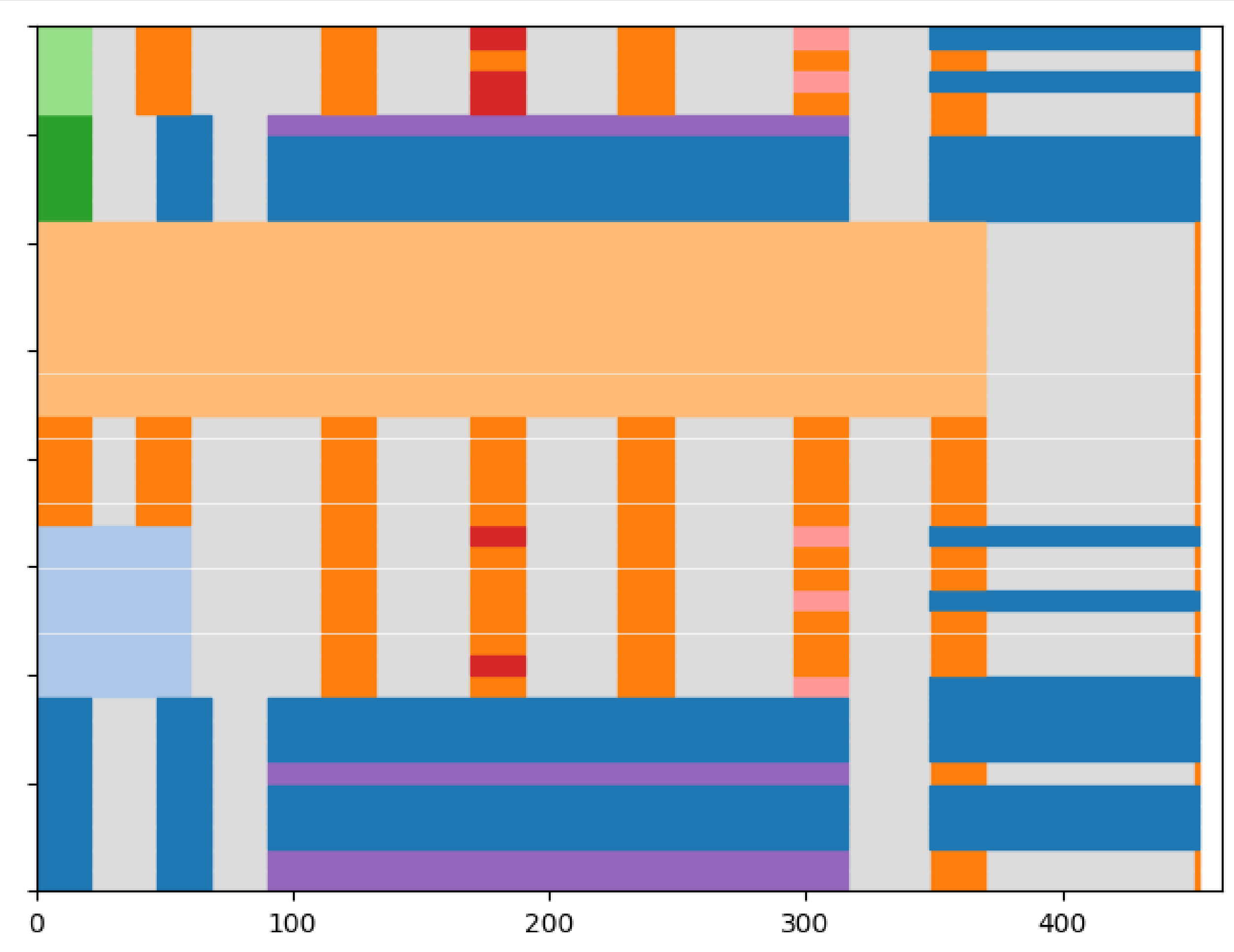
```
 70% (7 of 10) |################        | Elapsed Time: 0:00:00 ETA:   0:00:00
```

```
generating graph with nb_com =  6
```

```
100% (10 of 10) |########################| Elapsed Time: 0:00:00 ETA:  00:00:00
```

[26]:

```
#visualization
plot = tn.plot_longitudinal(generated_network,generated_comunities,height=600)
```



Mixing parameters Some parameters allow to tune how well defined is the community structure in term of network topology * `alpha` determines the internal density of communities. The average degree inside a community is approximately$ (n_{c}-1)^:nbsphinx-math:*alpha* '$ *with :math:'n_c* the number of nodes of community c. More precisely, the number of edges inside a community is equal to $d_c = \lceil \frac{n_c(n_c-1)^\alpha}{2} \rceil$. * `external_density_penalty`

corresponds to a penalty applied to the formula above for the density of the whole graph. The density among all nodes not in a community is defined as `external_density_penalty`*d_G. Beware, with small graphs, larger values often yield poor community structures. Note that edges added using this function are *stable*, i.e., if the community structure do not change, those nodes to not change either, contrary to the next option * `random_noise` corresponds to a different way to add randomness: this time, for each generated snapshot, a fraction of edges taken at random are rewired. It therefore adds randomness both inside an between communities. Unlike the previous one, choosing this parameter will lead to less edges inside communities than what has been set according to `alpha`.

We can illustrate this difference by generating a scenario without any community change and plotting the graph at some points.

First, all internal edges exist, no external edges exist

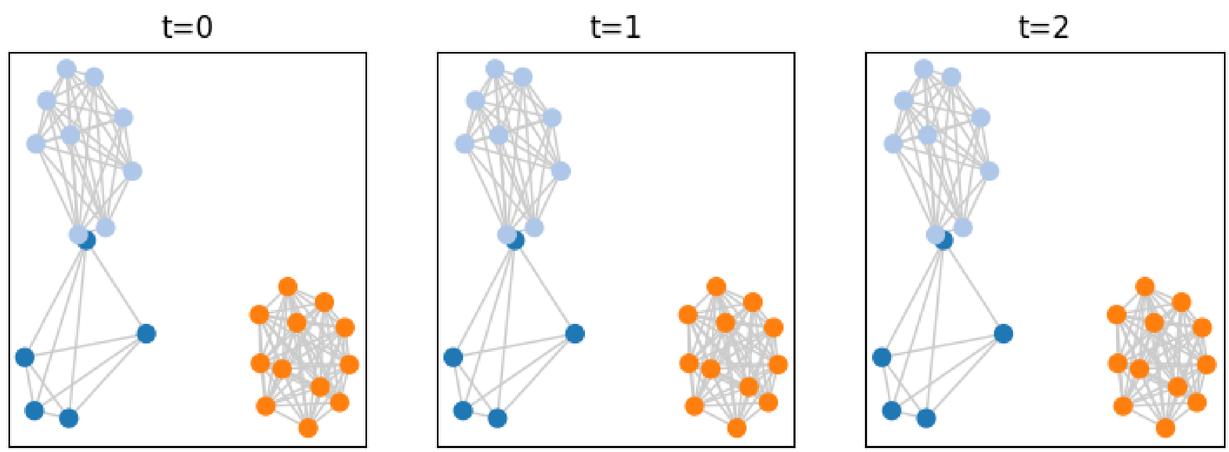
[27]:
```
my_scenario = tn.ComScenario(alpha=1,external_density_penalty=0,random_noise=0)
[com1,com2,com3] = my_scenario.INITIALIZE([5,9,12],["c1","c2","c3"])
my_scenario.CONTINUE(com1,delay=4)
(generated_network,generated_comunities) = my_scenario.run()
```

(continues on next page)

(continued from previous page)

```
times_to_plot = [0,1,2]
plot = tn.plot_as_graph(generated_network,generated_comunities,ts=times_to_plot,auto_
↪show=True,width=300,height=300,k=2.5,iterations=100)
```

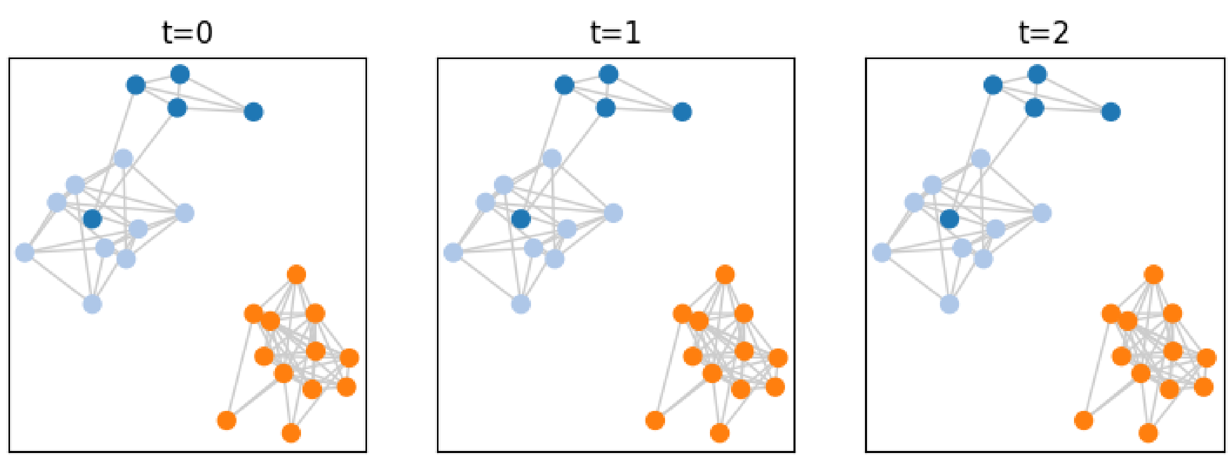
```
100% (1 of 1) |##########################| Elapsed Time: 0:00:00 ETA:  00:00:00
```



By decreasing `alpha`, communities become less dense.

[28]:
```
my_scenario = tn.ComScenario(alpha=0.8,external_density_penalty=0,random_noise=0)
[com1,com2,com3] = my_scenario.INITIALIZE([5,9,12],["c1","c2","c3"])
my_scenario.CONTINUE(com1,delay=4)
(generated_network,generated_comunities) = my_scenario.run()

times_to_plot = [0,1,2]
plot = tn.plot_as_graph(generated_network,generated_comunities,ts=times_to_plot,auto_
↪show=True,width=300,height=300,k=2.5,iterations=100)
```

```
100% (1 of 1) |##########################| Elapsed Time: 0:00:00 ETA:  00:00:00
```



By increasing external_density, some edges appear between communities. Note that, since the community structure do not evolves, the edges between communities do not change (see the article describing the benchmark for more details)

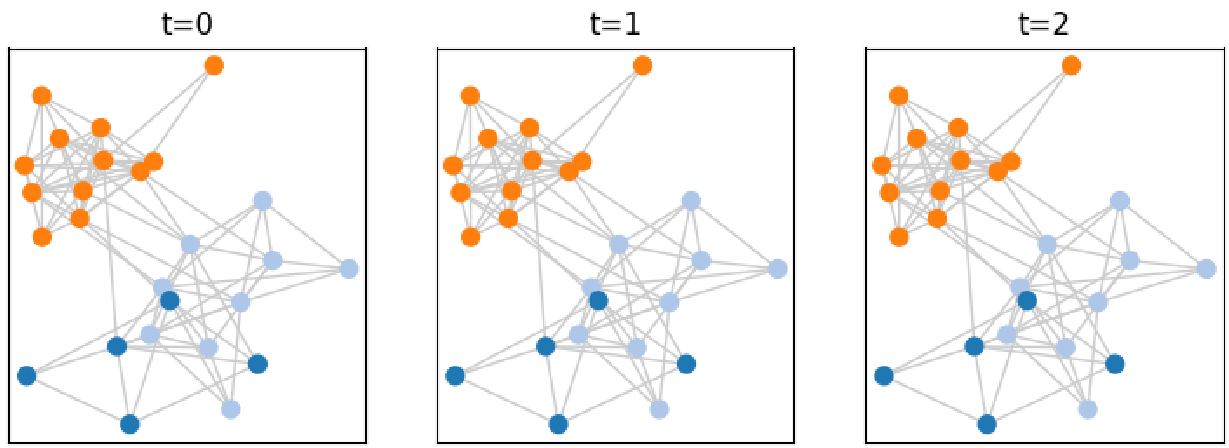
[29]:
```
my_scenario = tn.ComScenario(alpha=0.8,external_density_penalty=0.1,random_noise=0)
```

(continues on next page)

(continued from previous page)

```
[com1,com2,com3] = my_scenario.INITIALIZE([5,9,12],["c1","c2","c3"])
my_scenario.CONTINUE(com1,delay=4)
(generated_network,generated_comunities) = my_scenario.run()

times_to_plot = [0,1,2]
plot = tn.plot_as_graph(generated_network,generated_comunities,ts=times_to_plot,auto_
↪show=True,width=300,height=300,k=2.5,iterations=100)
```

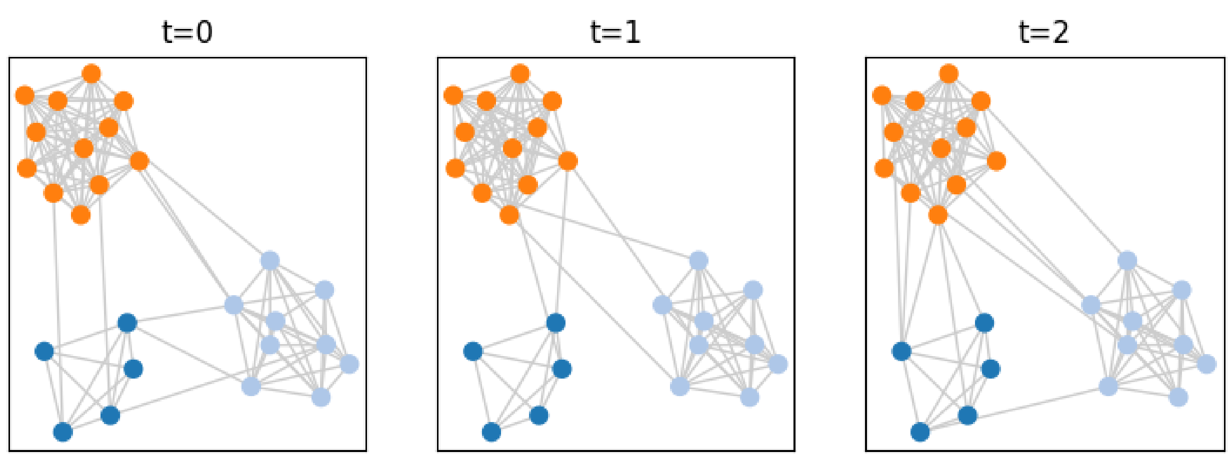
```
100% (1 of 1) |#########################| Elapsed Time: 0:00:00 ETA:  00:00:00
```



Instead, if we increase the `random_noise`, edges modifications are present but they differ from one snaphsots to the next, despite the community structure being unchanged

[30]:
```
my_scenario = tn.ComScenario(alpha=1,external_density_penalty=0,random_noise=0.1)
[com1,com2,com3] = my_scenario.INITIALIZE([5,9,12],["c1","c2","c3"])
my_scenario.CONTINUE(com1,delay=4)
(generated_network,generated_comunities) = my_scenario.run()

times_to_plot = [0,1,2]
plot = tn.plot_as_graph(generated_network,generated_comunities,ts=times_to_plot,auto_
↪show=True,width=300,height=300,k=2.5,iterations=100)
```

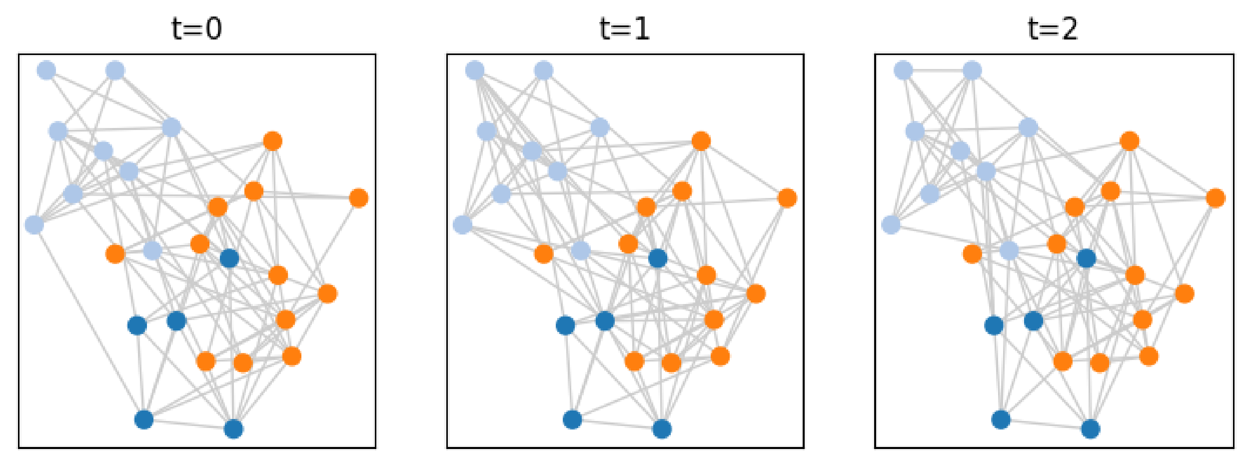
```
100% (1 of 1) |#########################| Elapsed Time: 0:00:00 ETA:  00:00:00
```



We can set all three parameters, but be careful when interpreting the results ! The community structure might quickly degrade

[31]:
```
my_scenario = tn.ComScenario(alpha=0.8,external_density_penalty=0.2,random_noise=0.2)
[com1,com2,com3] = my_scenario.INITIALIZE([5,9,12],["c1","c2","c3"])
my_scenario.CONTINUE(com1,delay=4)
(generated_network,generated_comunities) = my_scenario.run()

times_to_plot = [0,1,2]
plot = tn.plot_as_graph(generated_network,generated_comunities,ts=times_to_plot,auto_
↪show=True,width=300,height=300,k=2.5,iterations=100)
```

```
100% (1 of 1) |##########################| Elapsed Time: 0:00:00 ETA:  00:00:00
```



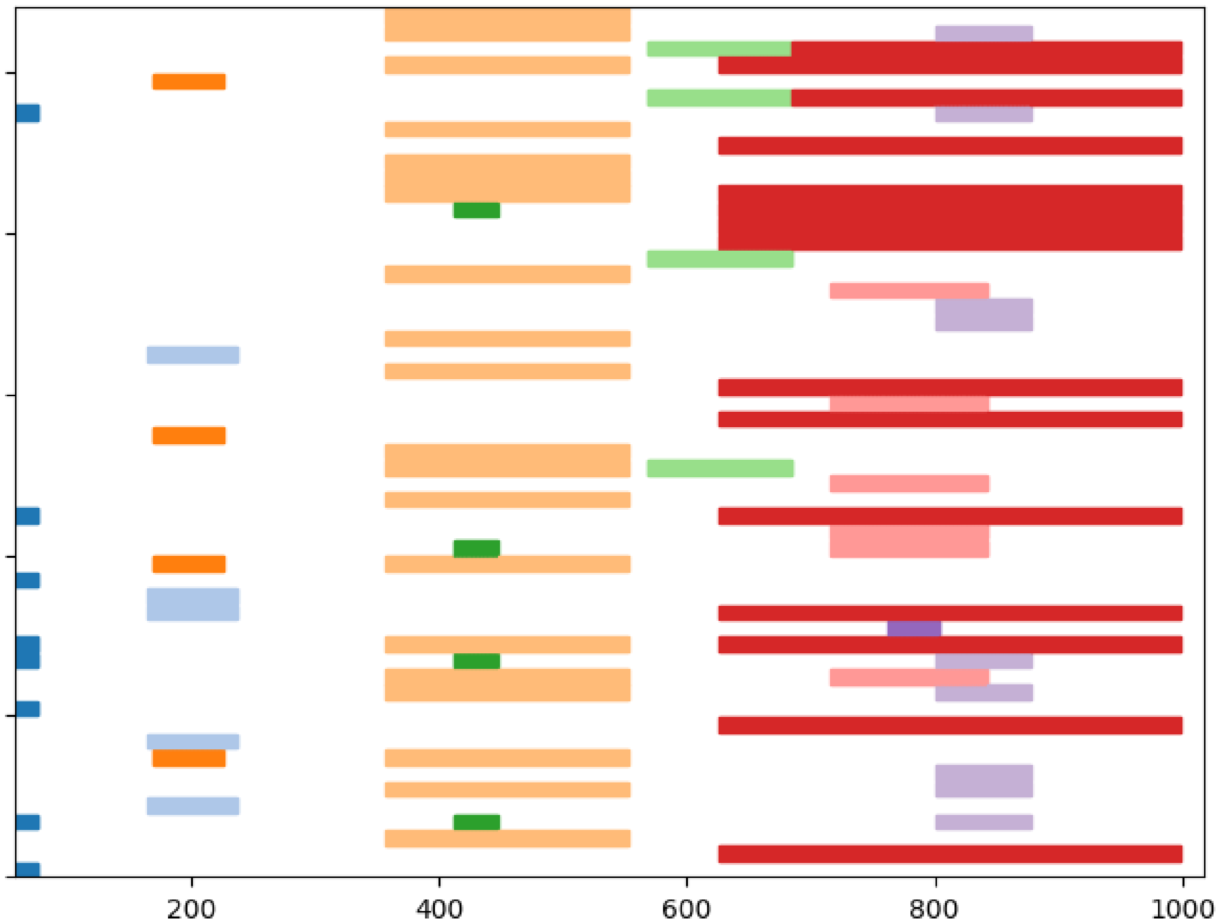
Benchmark for Multiple Temporal Scales

This benchmark allows to generate temporal networks as described in `Detecting Stable Communities in Link Streams at Multiple Temporal Scales. Boudebza, S., Cazabet, R., Nouali, O., & Azouaou, F. (2019).`.

To sum up the method, *stable* communities are generated (i.e., no node change). These communities exist for some periods, but have different *temporal scales*, i.e., some of them have a high frequency of edges (their edges appear at every step) while others have a lower frequency (i.e., each edge appear only every t steps). To simplify, communities are complete cliques.(but for the low frequency ones, we might observe only a small fraction of their edges in every step)

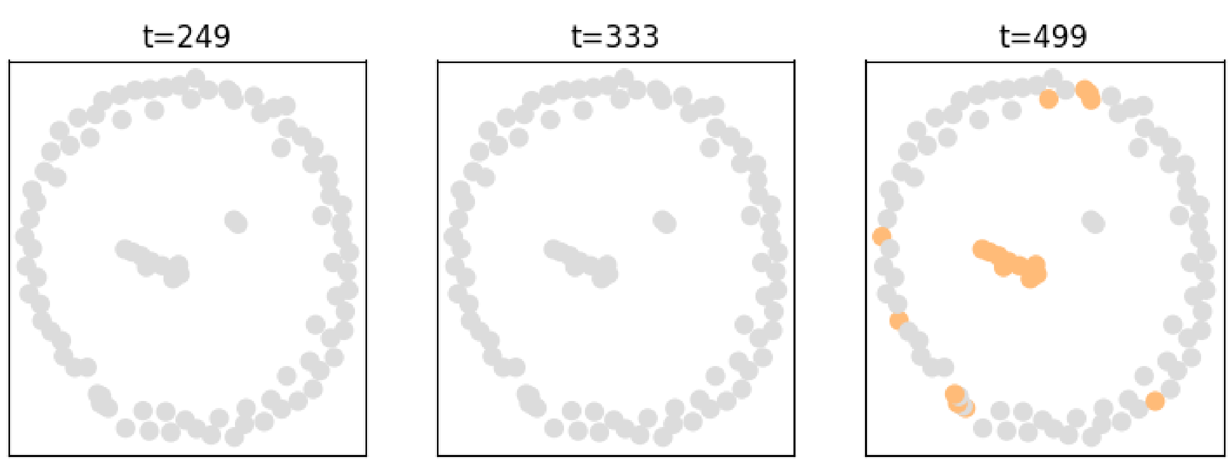
The basic parameters are the number of steps, number of nodes and number of communities. There are other parameters allowing to modify the random noise, the maximal size of communities and the maximal duration of communities, that are by default assigned with values scaled according to the other parameters. Check documentation for details.

[32]:
```
(generated_network,generated_comunities) = tn.generate_multi_temporal_scale(nb_
↪steps=1000,nb_nodes=100,nb_com=10)
plot = tn.plot_longitudinal(communities=generated_comunities,sn_duration=1)
```



We can observe that communities are not well defined on a given particular snapshot

[33]:
```
last_time = generated_network.end()
times_to_plot = [int(last_time/4),int(last_time/3),int(last_time/2)]
plot = tn.plot_as_graph(generated_network,generated_comunities,ts=times_to_plot,
↪width=300,height=300)
```



2.3.6 Reproducing results of the benchmark article

This notebook allows to reproduce results of the article: `Evaluating Community Detection Algorithms for Progressively Evolving Graphs`

[1]:
```
#If you have not installed tnetwork yet, you need to install it first, for instance␣
↪with this line

#!pip install --upgrade tnetwork==1.1
```

[1]:
```
import tnetwork as tn
import numpy as np
import seaborn as sns
import pandas as pd
import seaborn as sns
from tnetwork.experiments.experiments import *
import matplotlib.pyplot as plt
import datetime
from tnetwork import ComScenario
```

We start by defined the list of methods to test. In order to be able to execute the code online, we removed DYNAMO and transveral_network approaches that require to run locally (use of JAVA/Matlab)

[2]:
```
elapsed_time=True
def iterative(x, elapsed_time=elapsed_time):
    return tn.DCD.iterative_match(x, elapsed_time=elapsed_time)
def smoothed_louvain(x, elapsed_time=True):
    return tn.DCD.smoothed_louvain(x, elapsed_time=elapsed_time)
def smoothed_graph(x, elapsed_time=True):
    return tn.DCD.smoothed_graph(x, elapsed_time=elapsed_time, alpha=0.9)
#def label_smoothing(x, elapsed_time=True):
#    return tn.DCD.label_smoothing(x, elapsed_time=elapsed_time)

#def DYNAMO(x, elapsed_time=True):
#    return tn.DCD.externals.dynamo(x, elapsed_time=elapsed_time,timeout=100)
#def transversal_network(x, elapsed_time=True):
#    return tn.DCD.externals.transversal_network_mucha_original(x, elapsed_
↪time=elapsed_time,om=0.5, matlab_session=eng)

methods_to_test = { "smoothed-graph":smoothed_graph,
                   "implicit-global": smoothed_louvain,
                   "no-smoothing":iterative,
                   #"label-smoothing":label_smoothing
                  }
```

Qualitative analysis

We define the custom scenario on which to make experiments, following the paper.

Definition of the custom scenario

The function is part of `tnetwork` library, but we reproduce it here as a code example

[3]:
```
def generate_toy_random_network(**kwargs):
    """
    Generate a small, toy dynamic graph

    Generate a toy dynamic graph with evolving communities, following scenario␣
↪described in XXX
    Optional parameters are the same as those passed to the ComScenario class to␣
↪generate custom scenarios

    :return: pair, (dynamic graph, dynamic reference partition) (as snapshots)
    """
    my_scenario = ComScenario(**kwargs)

    # Initialization with 4 communities of different sizes
    [A, B, C, T] = my_scenario.INITIALIZE([5, 8, 20, 8],
                                                        ["A", "B", "C", "T"])
    # Create a theseus ship after 20 steps
    (T,U)=my_scenario.THESEUS(T, delay=20)

    # Merge two of the original communities after 30 steps
    B = my_scenario.MERGE([A, B], B.label(), delay=30)

    # Split a community of size 20 in 2 communities of size 15 and 5
    (C, C1) = my_scenario.SPLIT(C, ["C", "C1"], [15, 5], delay=75)

    # Split again the largest one, 40 steps after the end of the first split
    (C1, C2) = my_scenario.SPLIT(C, ["C", "C2"], [10, 5], delay=40)

    # Merge the smallest community created by the split, and the one created by the␣
↪first merge
    my_scenario.MERGE([C2, B], B.label(), delay=20)

    # Make a new community appear with 5 nodes, disappear and reappear twice, grow by␣
↪5 nodes and disappear
    R = my_scenario.BIRTH(5, t=25, label="R")
    R = my_scenario.RESURGENCE(R, delay=10)
    R = my_scenario.RESURGENCE(R, delay=10)
    R = my_scenario.RESURGENCE(R, delay=10)

    # Make the resurgent community grow by 5 nodes 4 timesteps after being ready
    R = my_scenario.GROW_ITERATIVE(R, 5, delay=4)

    # Kill the community grown above, 10 steps after the end of the addition of the␣
↪last node
    my_scenario.DEATH(R, delay=10)

    (dyn_graph, dyn_com) = my_scenario.run()
    dyn_graph_sn = dyn_graph.to_DynGraphSN(slices=1)
    GT_as_sn = dyn_com.to_DynCommunitiesSN(slices=1)
    return dyn_graph_sn, GT_as_sn
```

Generation of the two flavors, Sharp and Blurred

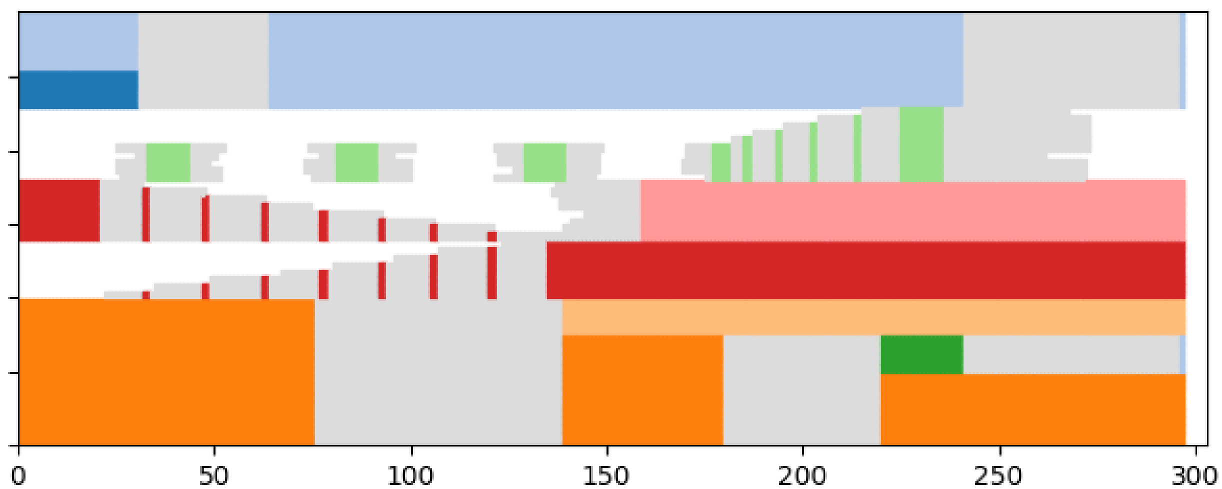
[4]:
```
dyn_graph_sharp,dyn_com_sharp= tn.generate_toy_random_network(random_noise=0.01,␣
↪external_density_penalty=0.05,alpha=0.9)
dyn_graph_blurred,dyn_com_blurred= tn.generate_toy_random_network(random_noise=0.01,␣
↪external_density_penalty=0.25,alpha=0.8)
```

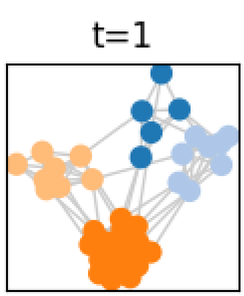
```
100% (26 of 26) |########################| Elapsed Time: 0:00:00 ETA:  00:00:00
```

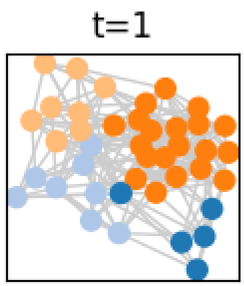
Plotting the ground truth

[6]:
```
node_order = dyn_com_sharp.automatic_node_order()
p = tn.plot_longitudinal(dyn_graph_sharp,dyn_com_sharp,nodes=node_order,height=300)
plt.show(p)
p = tn.plot_as_graph(dyn_graph_sharp,dyn_com_sharp,ts=1, height=150,width=150)
plt.show(p)
p = tn.plot_as_graph(dyn_graph_blurred,dyn_com_blurred,ts=1, height=150,width=150)
```

```
/usr/local/lib/python3.7/site-packages/numpy/core/numeric.py:2327: FutureWarning:␣
↪elementwise comparison failed; returning scalar instead, but in the future will␣
↪perform elementwise comparison
  return bool(asarray(a1 == a2).all())
```







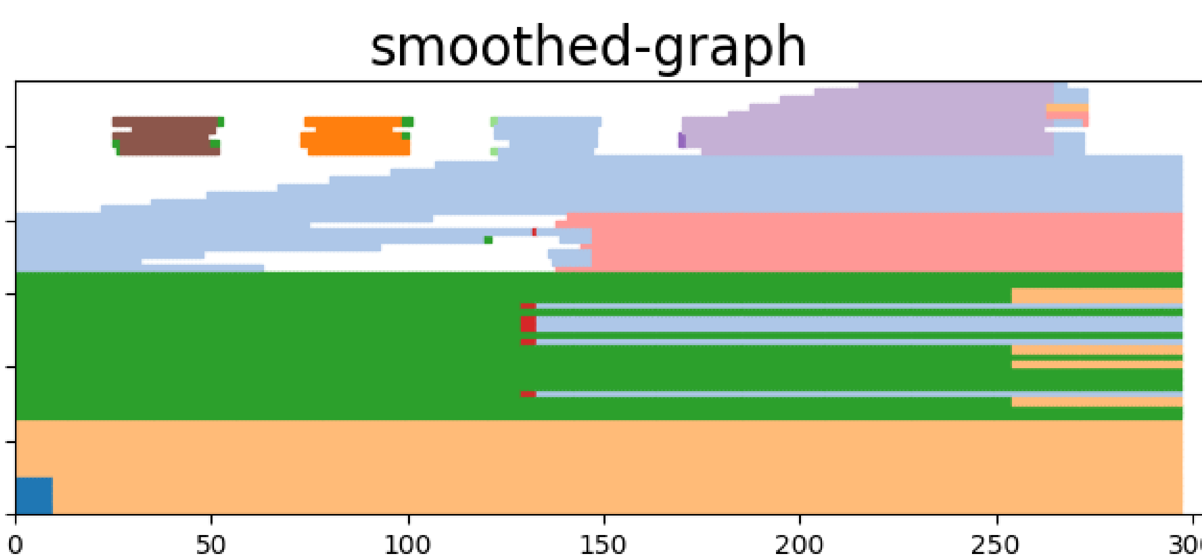
Run all algorithms

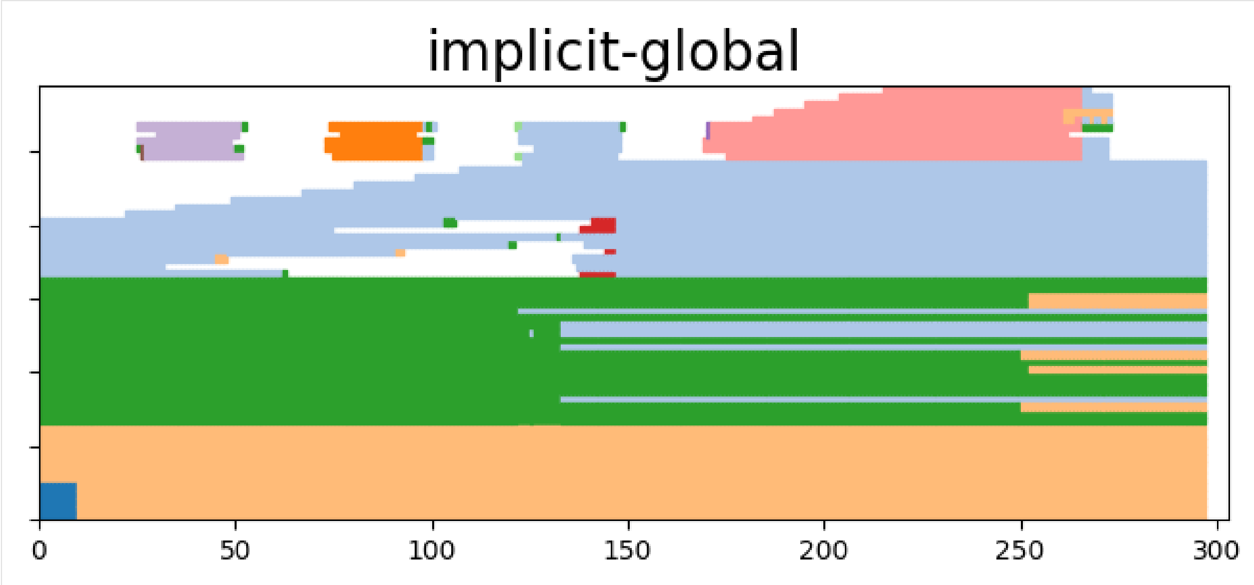
We use a function of tnetwork which takes a graph and a list of methods and return the communities. We then plot the results. We do it only for the sharp scenario in this example

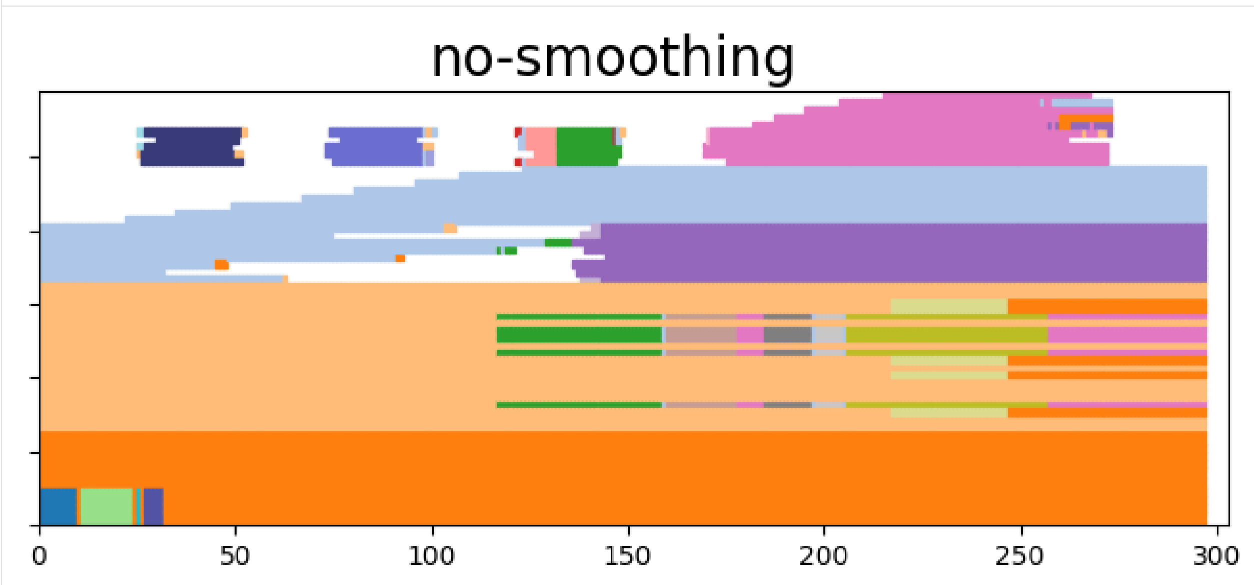
[7]:
```
coms_sharp = tn.run_algos_on_graph(methods_to_test,dyn_graph_sharp)
```

```
N/A% (0 of 297) |                               | Elapsed Time: 0:00:00 ETA:  --:--:--
starting smoothed_graph
N/A% (0 of 297) |                               | Elapsed Time: 0:00:00 ETA:  --:--:--
starting smoothed_louvain
N/A% (0 of 297) |                               | Elapsed Time: 0:00:00 ETA:  --:--:--
starting no_smoothing
 96% (286 of 297) |#####################  | Elapsed Time: 0:00:01 ETA:   0:00:00
```

[8]:
```
for name,(communities,time) in coms_sharp.items():
    to_plot = tn.plot_longitudinal(communities=communities,height=300)
    to_plot.suptitle(name, fontsize=20)
    plt.show(to_plot)
```







Quantitative analysis

Computing community qualities

The first test consists in computing scores when varying mu and keeping all other parameters constant. In order to run it quickly online, we choose only 3 values of mu and run only 1 iteration for each.

We use a function of `tnetwork` which, given a set of parameters, generate networks according to the generator described in the paper and compute all scores for them

Be careful, it takes a few minutes

[9]:
```
#mus = [0,0.05]+[0.1,0.15,0.2]+[0.3,0.4,0.5]
mus = [0,0.15,0.3]
```

(continues on next page)

(continued from previous page)

```
df_stats = tn.DCD_benchmark(methods_to_test,mus,iterations=1)
```

```
mu:  0
iteration:  0
generating graph with nb_com =  10
```

```
N/A% (0 of 1565) |                       | Elapsed Time: 0:00:00 ETA:  --:--:--
```

```
subset length: None
starting smoothed_graph
```

```
N/A% (0 of 1565) |                       | Elapsed Time: 0:00:00 ETA:  --:--:--
```

```
starting smoothed_louvain
```

```
N/A% (0 of 1565) |                       | Elapsed Time: 0:00:00 ETA:  --:--:--
```

```
starting no_smoothing
```

```
 99% (1563 of 1565) |################### | Elapsed Time: 0:00:24 ETA:   0:00:00
```

```
mu:  0.15
iteration:  0
generating graph with nb_com =  10
```

```
N/A% (0 of 821) |                        | Elapsed Time: 0:00:00 ETA:  --:--:--
```

```
subset length: None
starting smoothed_graph
```

```
N/A% (0 of 821) |                        | Elapsed Time: 0:00:00 ETA:  --:--:--
```

```
starting smoothed_louvain
```

```
N/A% (0 of 821) |                        | Elapsed Time: 0:00:00 ETA:  --:--:--
```

```
starting no_smoothing
```

```
 99% (816 of 821) |##################### | Elapsed Time: 0:00:10 ETA:   0:00:00
```

```
mu:  0.3
iteration:  0
generating graph with nb_com =  10
```

```
N/A% (0 of 776) |                        | Elapsed Time: 0:00:00 ETA:  --:--:--
```

```
subset length: None
starting smoothed_graph
```

```
N/A% (0 of 776) |                        | Elapsed Time: 0:00:00 ETA:  --:--:--
```

```
starting smoothed_louvain
```

```
N/A% (0 of 776) |                        | Elapsed Time: 0:00:00 ETA:  --:--:--
```

```
starting no_smoothing
```

```
 99% (773 of 776) |##################### | Elapsed Time: 0:00:15 ETA:   0:00:00
```

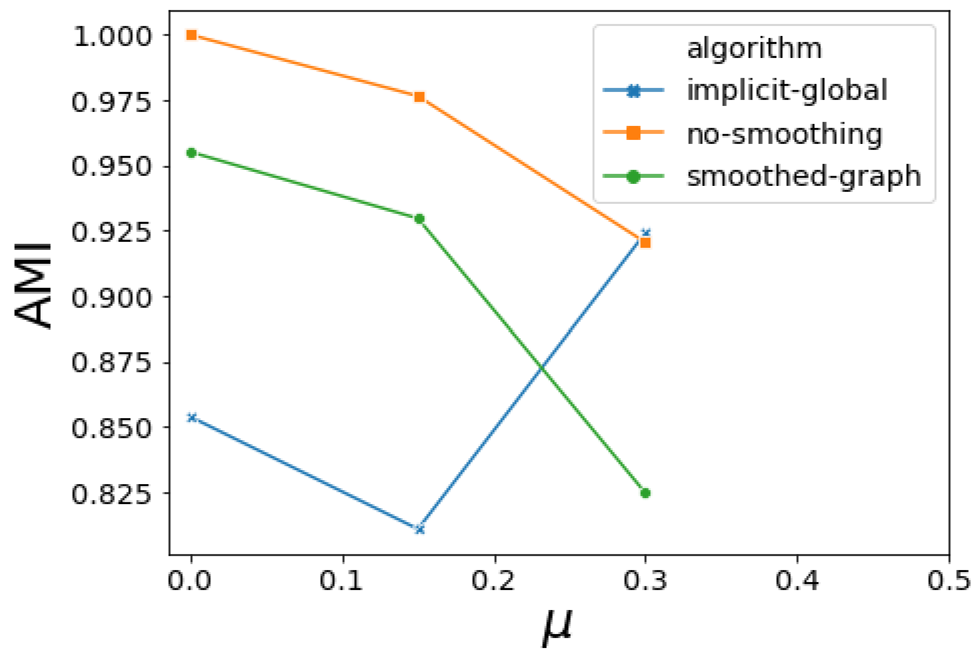
```
Compute stats
```

Visualize results

First with the longitudinal plots

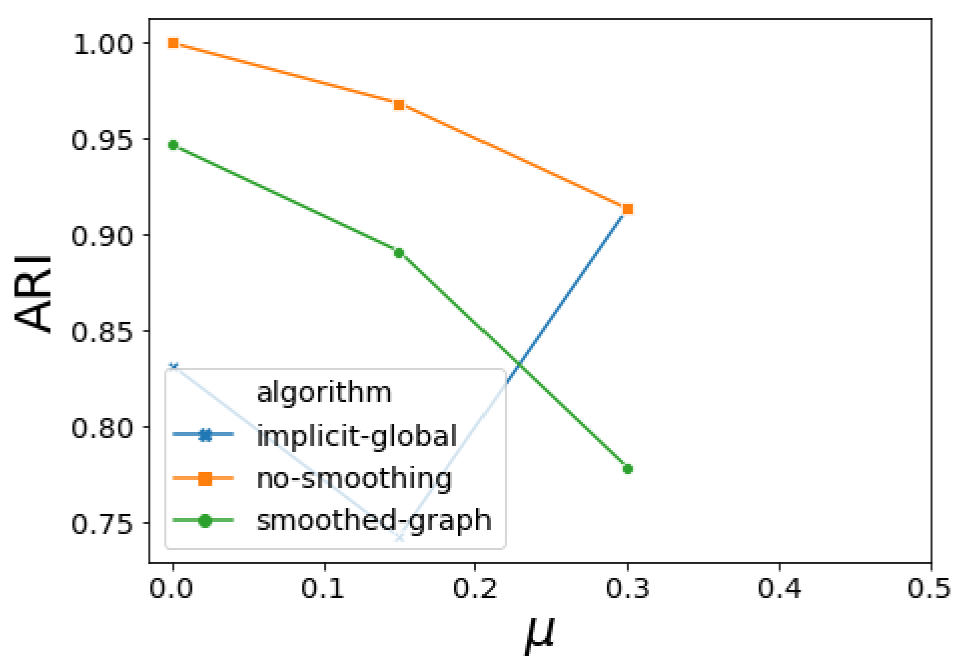
[10]:
```
import matplotlib.pyplot as plt
import matplotlib.pylab as pylab

params = {'legend.fontsize': 'x-large',
          'figure.figsize': (15, 5),
         'axes.labelsize': 'x-large',
         'axes.titlesize':'x-large',
         'xtick.labelsize':'x-large',
         'ytick.labelsize':'x-large'}
pylab.rcParams.update(params)

for carac in ["AMI","ARI","Q","LAMI","LARI","SM-P","SM-N","SM-L"]:
    plt.clf()

    sorted_methods_names = sorted(list(set(df_stats["algorithm"])))


    fig, ax = plt.subplots(figsize=(7, 5))
    ax = sns.lineplot(x="mu", y=carac, ax=ax,hue="algorithm",hue_order=sorted_methods_
↪names,style="algorithm",legend="full",data=df_stats,dashes=False,markers=True,err_
↪kws={"alpha":0.05})#,err_style="bars")
    ax.set_xlabel("$\mu$", fontsize=25)
    ax.set_ylabel(carac, fontsize=25)
    ax.set_xticks(np.arange(0.0, 0.51, 0.1))
    ax.ticklabel_format(axis="y",scilimits=(-1,1),style="sci")


    handles,labels = ax.get_legend_handles_labels()
    figlegend = pylab.figure(figsize=(4,3))
    figlegend.legend(handles,labels,loc="center")
    #ax.get_legend().remove()
    plt.show(fig)
#plt.show(figlegend)
```

```
<Figure size 1080x360 with 0 Axes>
```



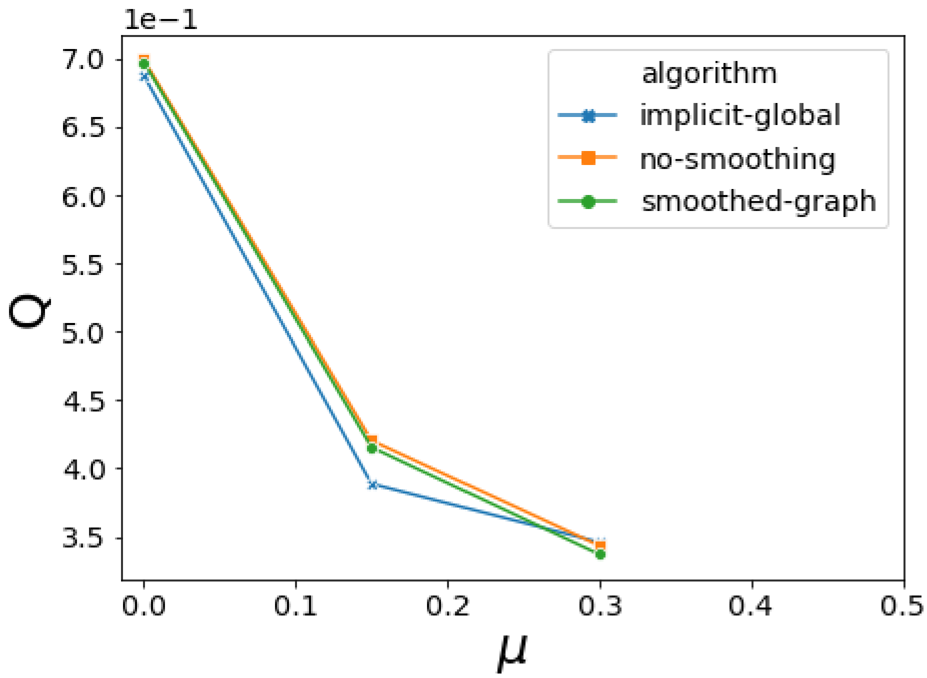

```
<Figure size 288x216 with 0 Axes>
```

```
<Figure size 1080x360 with 0 Axes>
```



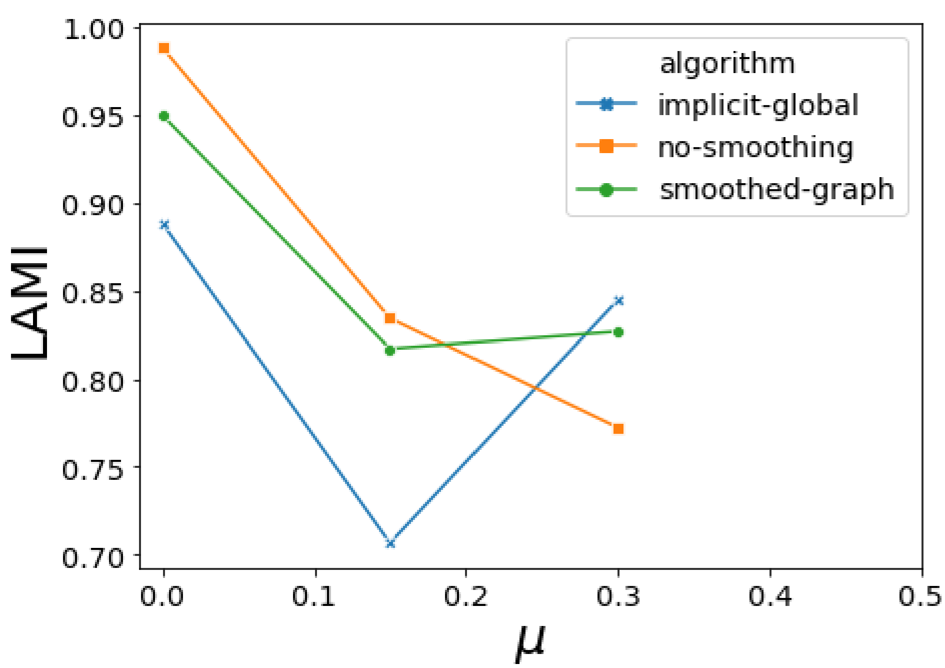

```
<Figure size 288x216 with 0 Axes>
```

```
<Figure size 1080x360 with 0 Axes>
```



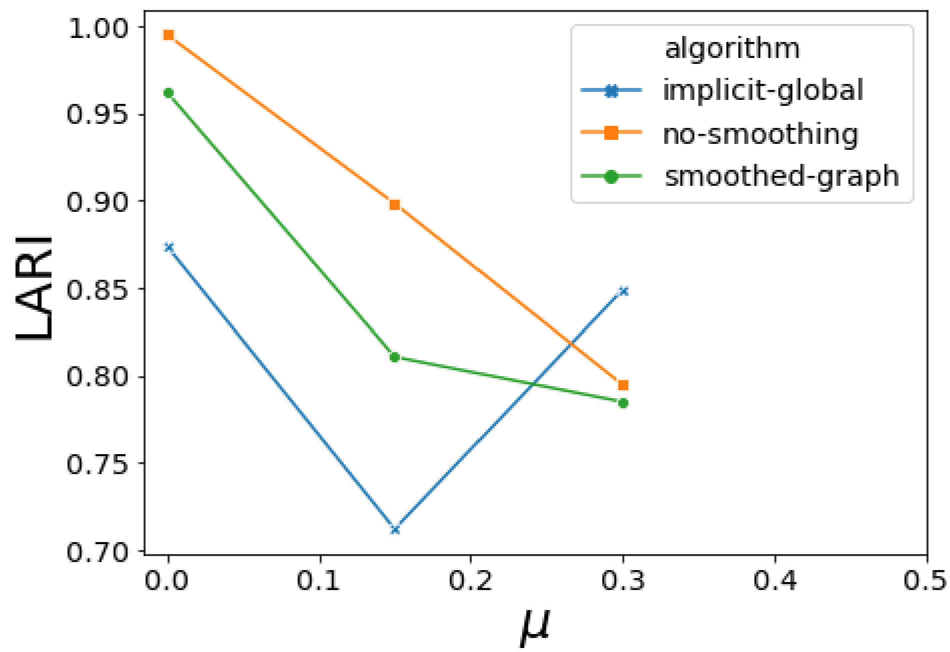

```
<Figure size 288x216 with 0 Axes>
```

```
<Figure size 1080x360 with 0 Axes>
```



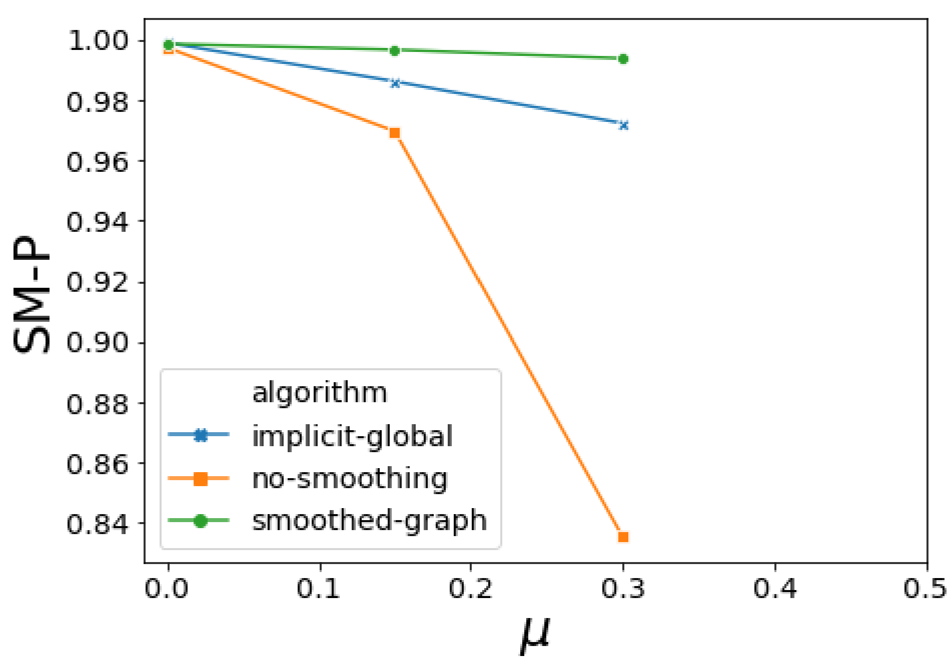

```
<Figure size 288x216 with 0 Axes>
```

```
<Figure size 1080x360 with 0 Axes>
```



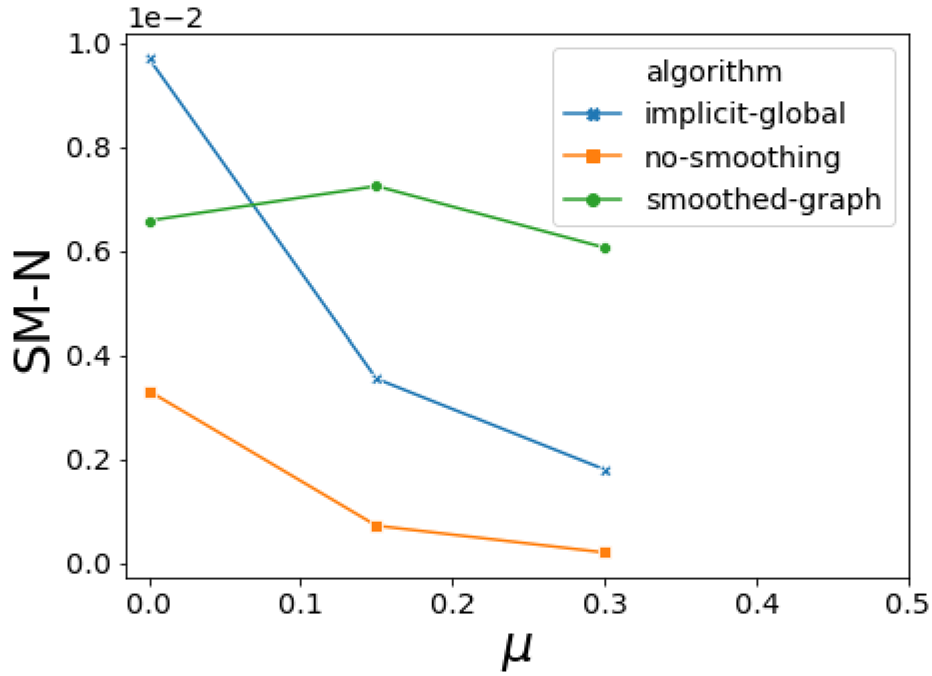

```
<Figure size 288x216 with 0 Axes>
```

```
<Figure size 1080x360 with 0 Axes>
```



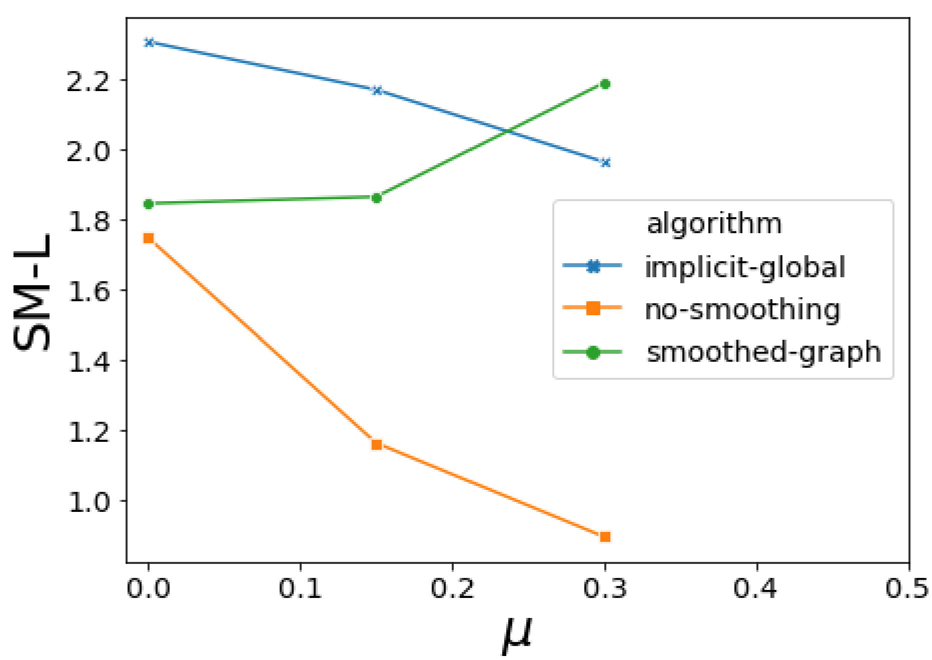

```
<Figure size 288x216 with 0 Axes>
```

```
<Figure size 1080x360 with 0 Axes>
```




```
<Figure size 288x216 with 0 Axes>
```

```
<Figure size 1080x360 with 0 Axes>
```




```
<Figure size 288x216 with 0 Axes>
```

Then visualize using a spider web plot

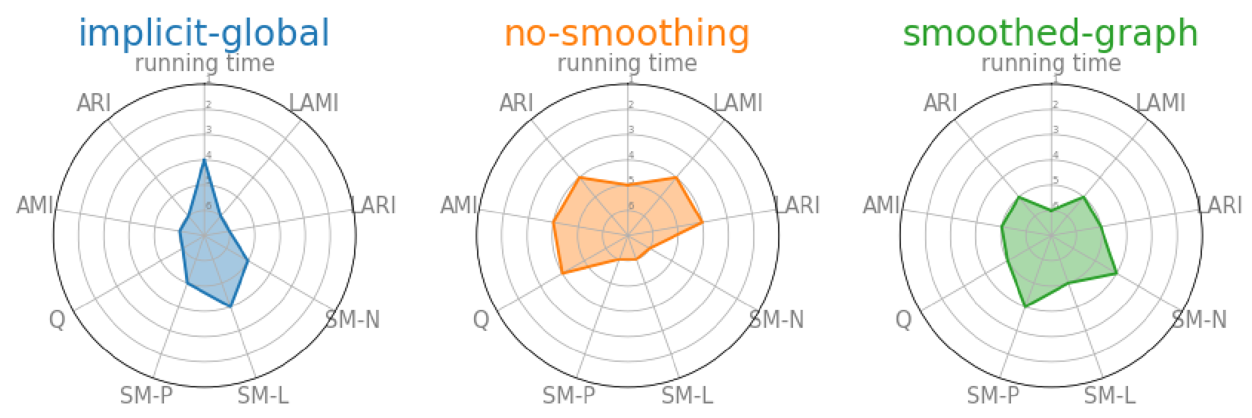
[11]:
```
df_stats = df_stats.drop([0])
```

[12]:
```
df = df_stats[df_stats["mu"]==0.15].groupby('algorithm', as_index=False).mean()
df['LAMI'] = df['LAMI'].rank(ascending=True)
df['LARI'] = df['LARI'].rank(ascending=True)
df['SM-N'] = df['SM-N'].rank(ascending=True)
df['SM-L'] = df['SM-L'].rank(ascending=True)
df['SM-P'] = df['SM-P'].rank(ascending=True)
df['Q'] = df['Q'].rank(ascending=True)
df['AMI'] = df['AMI'].rank(ascending=True)
df['ARI'] = df['ARI'].rank(ascending=True)
df['running time'] = df['running time'].rank(ascending=False)


df = df.drop(columns=[ "mu", "iteration", "# nodes","# steps", "#coms"])

# ------- PART 1: Define a function that do a plot for one line of the dataset!

pi=3.14159
def make_spider( row, title, color):

    # number of variable
    categories=list(df)[1:]
    N = len(categories)

    # What will be the angle of each axis in the plot? (we divide the plot / number␣
↪of variable)
    angles = [n / float(N) * 2 * pi for n in range(N)]
    angles += angles[:1]

    # Initialise the spider plot
    ax = plt.subplot(2,3,row+1, polar=True, )

    # If you want the first axis to be on top:
    ax.set_theta_offset(pi / 2)
    ax.set_theta_direction(-1)

    # Draw one axe per variable + add labels labels yet
    plt.xticks(angles[:-1], categories, color='grey', size=15)

    # Draw ylabels
    ax.set_rlabel_position(0)
    plt.yticks([1,2,3,4,5,6], ["6","5","4","3","2","1"], color="grey", size=7)
    plt.ylim(0,6)

    # Ind1
    values=df.loc[row].drop('algorithm').values.flatten().tolist()
    values += values[:1]
    ax.plot(angles, values, color=color, linewidth=2, linestyle='solid')
    ax.fill(angles, values, color=color, alpha=0.4)

    # Add a title
    plt.title(title, size=25, color=color, y=1.1)

    # ------- PART 2: Apply to all individuals
```

(continues on next page)

(continued from previous page)

```
    # initialize the figure
my_dpi=70
plt.figure(figsize=(1000/my_dpi, 700/my_dpi), dpi=my_dpi)

# Create a color palette:
#my_palette = plt.cm.get_cmap("Set2", len(df.index))
my_pallete = sns.color_palette()
# Loop to plot
for row in range(0, len(df.index)):
    make_spider( row=row, title=df['algorithm'][row], color=my_pallete[row])

plt.subplots_adjust(wspace=0.4)
```



Evaluate scalability

First by varying the number of steps

Again, we do it only for a few values for the sake of example

[13]:
```
#steps= [100,200,400,800,1200,1600,2000]
steps= [100,200,400]

df_stats = tn.DCD_benchmark(methods_to_test,mus=[0.2],nb_coms=[10],subsets=steps,
↪iterations=1,operations=40)
```

```
mu:  0.2
iteration:  0
generating graph with nb_com =  10
N/A% (0 of 100) |                        | Elapsed Time: 0:00:00 ETA:  --:--:--
subset length: 100
starting smoothed_graph
N/A% (0 of 100) |                        | Elapsed Time: 0:00:00 ETA:  --:--:--
starting smoothed_louvain
N/A% (0 of 100) |                        | Elapsed Time: 0:00:00 ETA:  --:--:--
starting no_smoothing
N/A% (0 of 200) |                        | Elapsed Time: 0:00:00 ETA:  --:--:--
```

```
subset length: 200
starting smoothed_graph
```

```
N/A% (0 of 200) |                        | Elapsed Time: 0:00:00 ETA:  --:--:--
```

```
starting smoothed_louvain
```

```
N/A% (0 of 200) |                        | Elapsed Time: 0:00:00 ETA:  --:--:--
```

```
starting no_smoothing
```

```
N/A% (0 of 400) |                        | Elapsed Time: 0:00:00 ETA:  --:--:--
```

```
subset length: 400
starting smoothed_graph
```

```
N/A% (0 of 400) |                        | Elapsed Time: 0:00:00 ETA:  --:--:--
```

```
starting smoothed_louvain
```

```
N/A% (0 of 400) |                        | Elapsed Time: 0:00:00 ETA:  --:--:--
```

```
starting no_smoothing
```

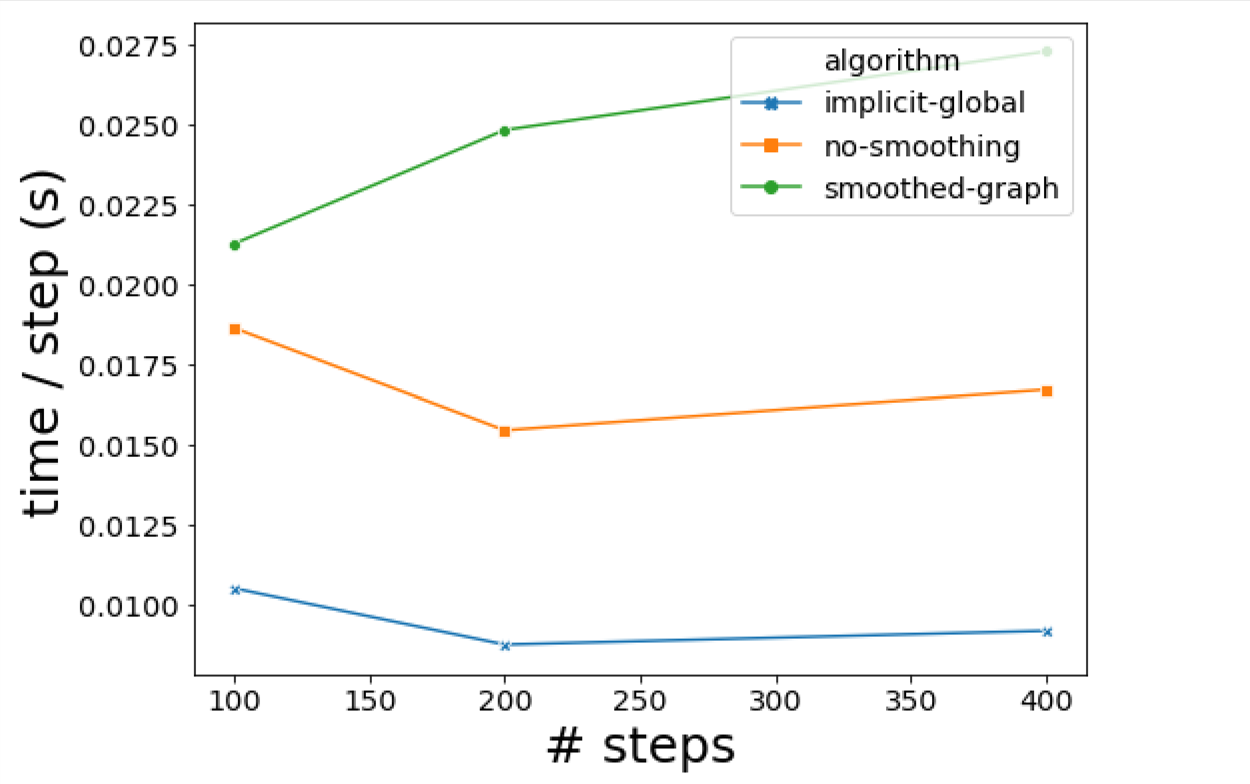
```
 98% (395 of 400) |##################### | Elapsed Time: 0:00:06 ETA:   0:00:00
```

```
Compute stats
```

[14]:

```
import matplotlib.pyplot as plt
import matplotlib.pylab as pylab

df_stats["running time"] = df_stats["running time"]/df_stats["# steps"]
df_stats = df_stats[df_stats["# steps"]>50]

params = {'legend.fontsize': 'x-large',
          'figure.figsize': (15, 5),
         'axes.labelsize': 'x-large',
         'axes.titlesize':'x-large',
         'xtick.labelsize':'x-large',
         'ytick.labelsize':'x-large'}
pylab.rcParams.update(params)

for carac in ["running time"]:
    plt.clf()

    fig, ax = plt.subplots(figsize=(8, 6))
    sorted_methods_names = sorted(list(set(df_stats["algorithm"])))
    ax = sns.lineplot(x="# steps", y=carac, ax=ax,hue="algorithm",hue_order=sorted_
↪methods_names,style="algorithm",legend="full",data=df_stats,dashes=False,
↪markers=True,err_kws={"alpha":0.05})#,err_style="bars")
    ax.set_xlabel("# steps", fontsize=25)
    ax.set_ylabel(" time / step (s)", fontsize=25)
```

```
<Figure size 1080x360 with 0 Axes>
```



Secondly by varying the number of nodes

[15]:
```
#nb_coms = [10,25,50,75,100]
nb_coms = [10,15,20]

df_stats = tn.DCD_benchmark(methods_to_test, mus=[0.2],nb_coms=nb_coms,subsets=[50],
↪iterations=1,operations=5)
```

```
mu:  0.2
iteration:  0
generating graph with nb_com =  10
```

```
N/A% (0 of 50) |                        | Elapsed Time: 0:00:00 ETA:  --:--:--
```

```
subset length: 50
starting smoothed_graph
```

```
N/A% (0 of 50) |                        | Elapsed Time: 0:00:00 ETA:  --:--:--
```

```
starting smoothed_louvain
```

```
N/A% (0 of 50) |                        | Elapsed Time: 0:00:00 ETA:  --:--:--
```

```
starting no_smoothing
```

```
 96% (48 of 50) |###################### | Elapsed Time: 0:00:00 ETA:   0:00:00
```

```
generating graph with nb_com =  15
```

```
N/A% (0 of 50) |                        | Elapsed Time: 0:00:00 ETA:  --:--:--
```

```
subset length: 50
starting smoothed_graph
```

```
N/A% (0 of 50) |                          | Elapsed Time: 0:00:00 ETA:  --:--:--
```

```
starting smoothed_louvain
```

```
N/A% (0 of 50) |                          | Elapsed Time: 0:00:00 ETA:  --:--:--
```

```
starting no_smoothing
```

```
 94% (47 of 50) |######################   | Elapsed Time: 0:00:01 ETA:   0:00:00
```

```
generating graph with nb_com =  20
```

```
N/A% (0 of 50) |                          | Elapsed Time: 0:00:00 ETA:  --:--:--
```

```
subset length: 50
starting smoothed_graph
```

```
N/A% (0 of 50) |                          | Elapsed Time: 0:00:00 ETA:  --:--:--
```

```
starting smoothed_louvain
```

```
N/A% (0 of 50) |                          | Elapsed Time: 0:00:00 ETA:  --:--:--
```

```
starting no_smoothing
```

```
 98% (49 of 50) |####################### | Elapsed Time: 0:00:01 ETA:   0:00:00
```

```
Compute stats
```

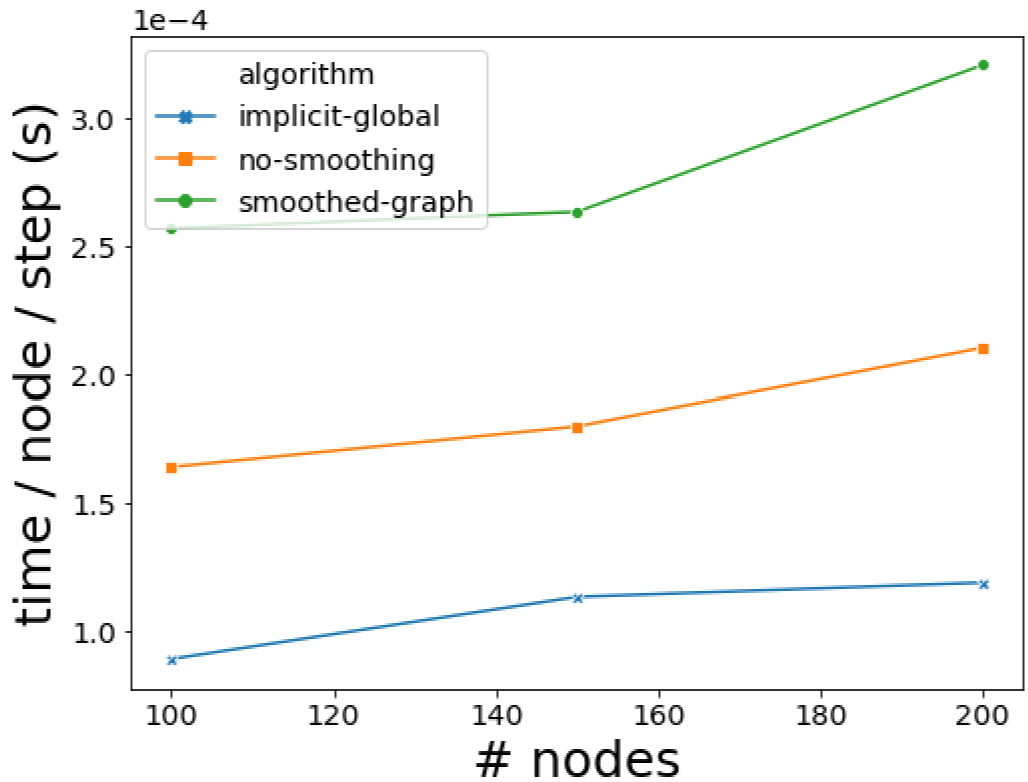
[16]:

```
import matplotlib.pyplot as plt
import matplotlib.pylab as pylab

df_stats["#coms"] = df_stats["#coms"]*10
df_stats["running time"] = df_stats["running time"]/(df_stats["# nodes"])/df_stats["#␣
↪steps"]



#df_stats["#coms"] = df_stats["#coms"]*10
params = {'legend.fontsize': 'x-large',
          'figure.figsize': (15, 5),
         'axes.labelsize': 'x-large',
         'axes.titlesize':'x-large',
         'xtick.labelsize':'x-large',
         'ytick.labelsize':'x-large'}
pylab.rcParams.update(params)

for carac in ["running time"]:
    plt.clf()

    fig, ax = plt.subplots(figsize=(8, 6))
    sorted_methods_names = sorted(list(set(df_stats["algorithm"])))


    ax = sns.lineplot(x="#coms", y=carac, ax=ax,hue="algorithm",style="algorithm",hue_
↪order=sorted_methods_names,legend="full",data=df_stats,dashes=False,markers=True,
↪err_kws={"alpha":0.05})#,err_style="bars")
    ax.set_xlabel("# nodes", fontsize=25)
    ax.set_ylabel("time / node / step (s)", fontsize=25)
    ax.ticklabel_format(axis="y",scilimits=(-1,1),style="sci")
```

(continues on next page)

(continued from previous page)

```
<Figure size 1080x360 with 0 Axes>
```



[ ]:

2.3.7 Reproducing results of the graph encoding article

This notebook allows to reproduce results of the article: `Data compression to choose a proper dynamic network representation`

[67]:
```
#If you have not installed tnetwork yet, you need to install it first, for instance␣
↪with this line

#!pip install --upgrade tnetwork==1.1
```

[4]:
```
import tnetwork as tn
import pandas as pd
import seaborn as sns
import numpy as np
import networkx as nx
import matplotlib.pyplot as plt
```

```
The autoreload extension is already loaded. To reload it, use:
  %reload_ext autoreload
```

We first define a function which, given a dynamic graph and a series of periods of aggregations, returns the encoding length according to the 4 encoding strategies for each dynamic graph produced by the periods of aggregation.

Note that the code of the encoding computation itself is available as part of the tnetwork library, and can be found there: `https://github.com/Yquetzal/tnetwork/blob/master/tnetwork/dyn_graph/encodings.py`

[5]:
```
# First, we define the functions we want to use to compute encodings
def score_sn_m(g_sn,g_ig):
    return(tn.code_length_SN_M(g_sn))
def score_sn_e(g_sn,g_ig):
    return(tn.code_length_SN_E(g_sn))
def score_ig(g_sn,g_ig):
    return(tn.code_length_IG(g_ig))
def score_ls(g_sn,g_ig):
    return tn.code_length_LS(g_sn)

functions = [score_ls,score_sn_m,score_ig,score_sn_e]
# We also specify the corresponding names to plot on the figures
names= ["$LS$","$SN_M$","$IG$","$SN_E$"]
```

[9]:
```
def compute_stats(ps,tts):
    """
    :param ps: original graph in snpashot format
    :param tts: list of length of sliding windows to test
    """
    sn1 = []
    sn2 = []
    ls = []
    ig=[]
    updates=[]

    scores = []

    for tt in tts:
        print("====",tt," ====")
        ps_tt=ps.aggregate_sliding_window(tt,weighted=False)
        ps_ig = ps_tt.to_DynGraphIG()
        scores.append([tt]+[f(ps_tt,ps_ig) for f in functions])



    df = pd.DataFrame.from_records(scores,columns=["tts"]+names)
    return df
```

Real graphs

First, we compute encoding lenght with a real graph. We choose tts to go from 20s (the actual collection frequency) to a period as long as the whole dataset.

We show here a single example as any other network can be treated the same way. Results for graphs used in the paper are available at the end of this notebook.

[11]:
```
h = 3600
d=h*24
tts=[5*d,4*d,2*d,d,h*12,h*6,h*4,h*2,h,60*30,60*15,60*5,60*2,60,20]
SP2012 = compute_stats(tn.graph_socioPatterns2012(format=tn.DynGraphSN),tts)
```

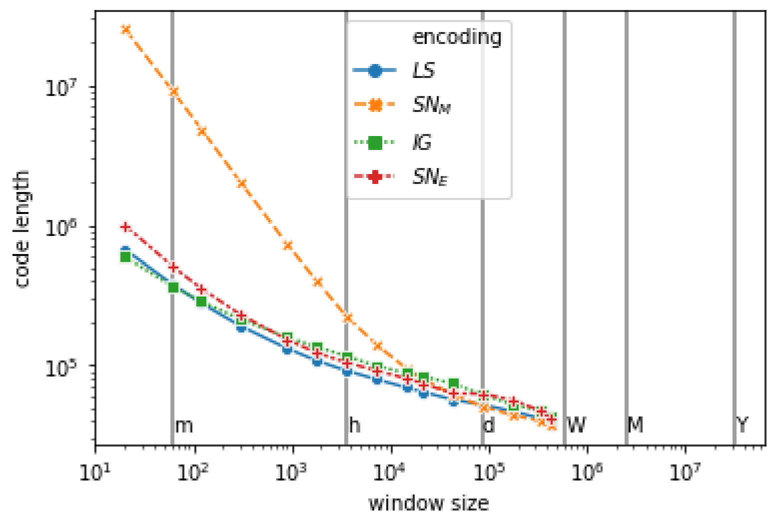
```
graph will be loaded as:  <class 'tnetwork.dyn_graph.dyn_graph_sn.DynGraphSN'>
==== 432000  ====
==== 345600  ====
==== 172800  ====
==== 86400  ====
==== 43200  ====
==== 21600  ====
==== 14400  ====
==== 7200  ====
==== 3600  ====
==== 1800  ====
==== 900  ====
==== 300  ====
==== 120  ====
==== 60  ====
==== 20  ====
```

To improve readability of the plots, we create a function to add vertical lines on human-intepretable periods

[12]:
```
def print_lines(long):
    plt.axvline(60,color="grey",zorder=1)
    plt.axvline(3600,color="grey",zorder=1)
    plt.axvline(3600*24,color="grey",zorder=1)
    plt.axvline(3600*24*7,color="grey",zorder=1)
    plt.axvline(3600*24*30,color="grey",zorder=1)
    plt.axvline(3600*24*365,color="grey",zorder=1)



    y0=min(long["value"])*0.9
    plt.text(60,y0,'m',rotation=0)
    plt.text(3600,y0,'h',rotation=0)
    plt.text(3600*24,y0,'d',rotation=0)
    plt.text(3600*24*7,y0,'W',rotation=0)
    plt.text(3600*24*30,y0,'M',rotation=0)
    plt.text(3600*24*365,y0,'Y',rotation=0)
```

Finally, we plot the result

[47]:
```
long = pd.melt(SP2012,id_vars=['tts'],value_vars=names)
long["value"]=long["value"]
long["encoding"]=long["variable"]
ax = sns.lineplot(x="tts",y="value",data=long,hue="encoding",markers=True,style=
↪"encoding")
ax.set_xscale('log')
ax.set_yscale('log')
ax.set_xlabel("window size")
ax.set_ylabel("code length")
print_lines(long)
plt.savefig('encoding/SP2012.pdf')
```



Synthetic graphs

[16]:
```
nb_nodes = 100
nb_edges = 640
nb_steps = 64
```

Stable

[19]:
```
tts=[32,16,8,4,2,1]

aGraph = nx.generators.gnm_random_graph(nb_nodes,nb_edges)
dynnet = tn.DynGraphSN([aGraph]*nb_steps)

df_stable = compute_stats(dynnet,tts)
```

```
==== 32  ====
==== 16  ====
==== 8  ====
==== 4  ====
==== 2  ====
==== 1  ====
```

Independent snapshots, dense

[27]:
```
tts=[32,16,8,4,2,1]

independent = [nx.generators.gnm_random_graph(nb_nodes,nb_edges) for i in range(nb_
↪steps)]
dynnet = tn.DynGraphSN(independent)

df_ind_dense = compute_stats(dynnet,tts)
```

```
==== 32  ====
==== 16  ====
==== 8  ====
==== 4  ====
==== 2  ====
==== 1  ====
```

Independent snapshots, sparse

[29]:
```
tts=[32,16,8,4,2,1]

independent = [nx.generators.gnm_random_graph(nb_nodes,nb_edges/nb_steps) for i in␣
↪range(nb_steps)]
dynnet = tn.DynGraphSN(independent)

df_ind_sparse = compute_stats(dynnet,tts)
```

```
==== 32  ====
==== 16  ====
==== 8  ====
==== 4  ====
==== 2  ====
==== 1  ====
```

Progressively evolving Graph (PEG) benchmark

[35]:
```
tts=[512,256,128,64,32,16,8,4,2,1]
dynnet,_ = tn.generate_simple_random_graph()

df_bench = compute_stats(dynnet.to_DynGraphSN(1),tts)
```

```
generating graph with nb_com =  10
```

```
100% (20 of 20) |########################| Elapsed Time: 0:00:04 ETA:  00:00:00
```

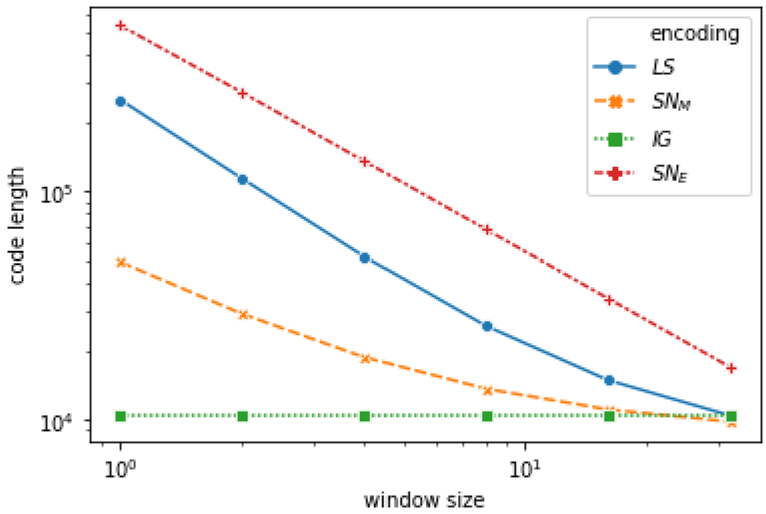
```
==== 512  ====
==== 256  ====
==== 128  ====
==== 64  ====
==== 32  ====
==== 16  ====
==== 8  ====
==== 4  ====
==== 2  ====
==== 1  ====
```

[42]:
```
long = pd.melt(df_stable,id_vars=['tts'],value_vars=names)
long["value"]=long["value"]
long["encoding"]=long["variable"]
ax = sns.lineplot(x="tts",y="value",data=long,hue="encoding",markers=True,style=
↪"encoding")
ax.set_xscale('log')
ax.set_yscale('log')
ax.set_xlabel("window size")
```

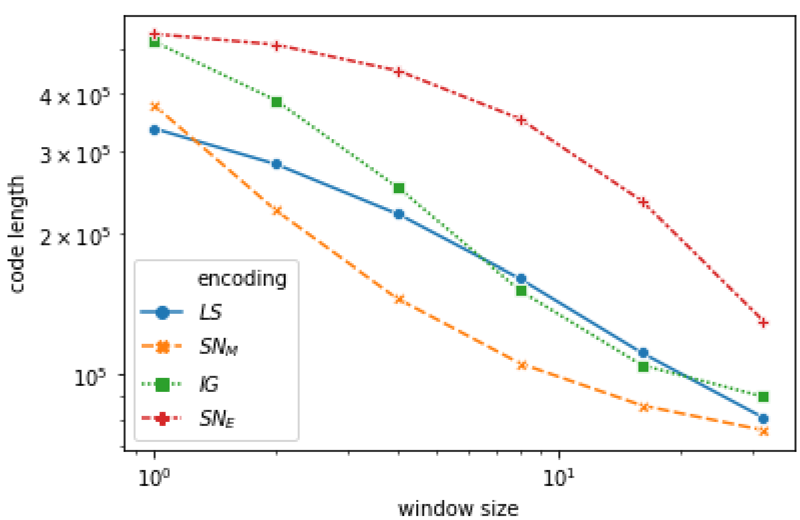
(continues on next page)

(continued from previous page)

```
ax.set_ylabel("code length")
plt.savefig('encoding/stable.pdf')
```



[43]:
```
long = pd.melt(df_ind_dense,id_vars=['tts'],value_vars=names)
long["value"]=long["value"]
long["encoding"]=long["variable"]
ax = sns.lineplot(x="tts",y="value",data=long,hue="encoding",markers=True,style=
↪"encoding")
ax.set_xscale('log')
ax.set_yscale('log')
ax.set_xlabel("window size")
ax.set_ylabel("code length")
plt.savefig('encoding/independent_dense.pdf')
```

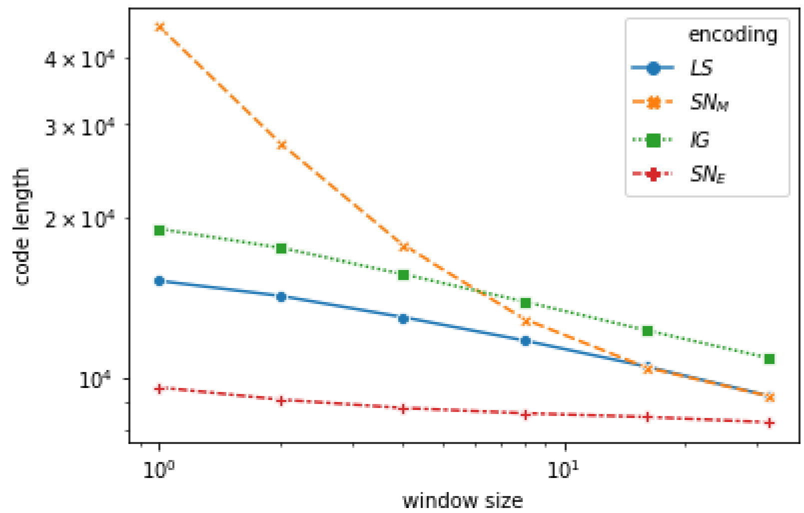


[44]:
```
long = pd.melt(df_ind_sparse,id_vars=['tts'],value_vars=names)
long["value"]=long["value"]
long["encoding"]=long["variable"]
ax = sns.lineplot(x="tts",y="value",data=long,hue="encoding",markers=True,style=
↪"encoding")
```

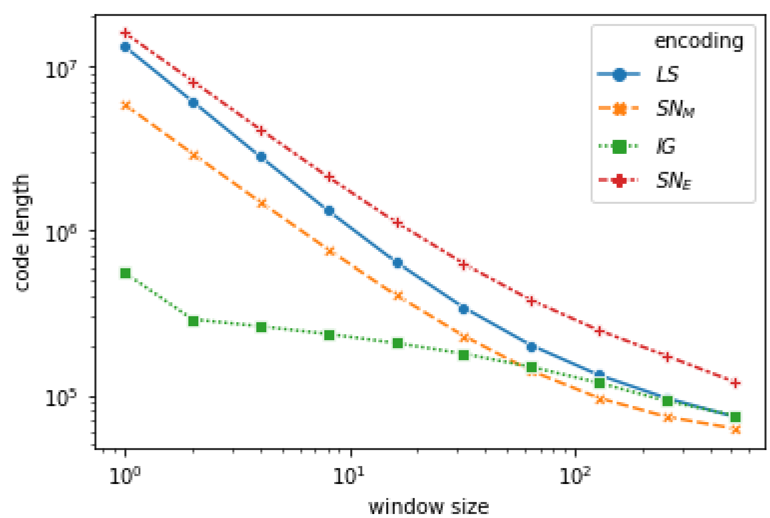
(continues on next page)

(continued from previous page)

```
ax.set_xscale('log')
ax.set_yscale('log')
ax.set_xlabel("window size")
ax.set_ylabel("code length")
plt.savefig('encoding/independent_sparse.pdf')
```



[45]:
```
long = pd.melt(df_bench,id_vars=['tts'],value_vars=names)
long["value"]=long["value"]
long["encoding"]=long["variable"]
ax = sns.lineplot(x="tts",y="value",data=long,hue="encoding",markers=True,style=
↪"encoding")
ax.set_xscale('log')
ax.set_yscale('log')
ax.set_xlabel("window size")
ax.set_ylabel("code length")
plt.savefig('encoding/bench.pdf')
```



Experiments with other real networks

[49]:
```
tts=[2*d,d,h*12,h*6,h*4,h*2,h,60*30,60*15,60*5,60*2,60,20]
SP_hospital = compute_stats(tn.graph_socioPatterns_Hospital(format=tn.DynGraphSN),tts)
```

```
graph will be loaded as:  <class 'tnetwork.dyn_graph.dyn_graph_sn.DynGraphSN'>
==== 432000  ====
==== 345600  ====
==== 172800  ====
==== 86400  ====
==== 43200  ====
==== 21600  ====
==== 14400  ====
==== 7200  ====
==== 3600  ====
==== 1800  ====
==== 900  ====
==== 300  ====
==== 120  ====
==== 60  ====
==== 20  ====
```

[60]:
```
tts=[2*d,d,h*12,h*6,h*4,h*2,h,60*30,60*15,60*5,60*2,60,20]
SP_PS = compute_stats(tn.graph_socioPatterns_Primary_School(format=tn.DynGraphSN),tts)
```

```
graph will be loaded as:  <class 'tnetwork.dyn_graph.dyn_graph_sn.DynGraphSN'>
==== 172800  ====
==== 86400  ====
==== 43200  ====
==== 21600  ====
==== 14400  ====
==== 7200  ====
==== 3600  ====
==== 1800  ====
==== 900  ====
==== 300  ====
==== 120  ====
==== 60  ====
==== 20  ====
```

[61]:
```
tts=[250,100,50,30,15,10,7,5,4,3,2,1]
GOT = compute_stats(tn.graph_GOT(),tts)
```

```
==== 250  ====
==== 100  ====
==== 50  ====
==== 30  ====
==== 15  ====
==== 10  ====
==== 7  ====
==== 5  ====
==== 4  ====
==== 3  ====
==== 2  ====
==== 1  ====
```

[55]:
```
h = 3600
```

(continues on next page)

(continued from previous page)

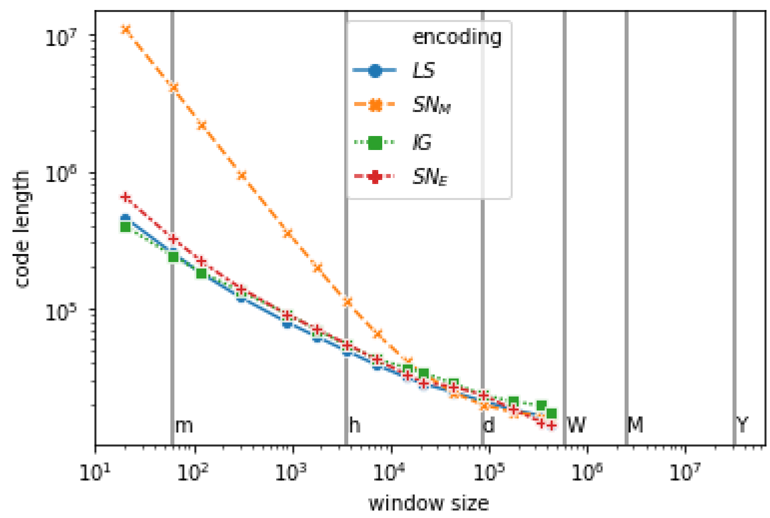
```
d=h*24
tts=[d*365,d*30,d*7,d,h,60]

location ="ia-enron-employees/"
ENRON = compute_stats(
    tn.read_interactions(location+"ia-enron-employees.edges",format=tn.DynGraphSN,sep=
↪" ",columns=["n1","n2","?","time"])
    ,tts)
```

```
graph will be loaded as:  <class 'tnetwork.dyn_graph.dyn_graph_sn.DynGraphSN'>
==== 31536000  ====
==== 2592000  ====
==== 604800  ====
==== 86400  ====
==== 3600  ====
==== 60  ====
```

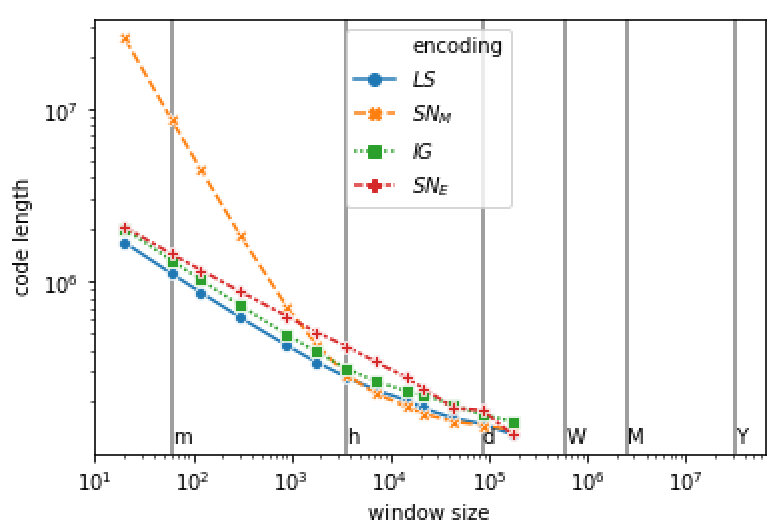
[57]:
```
location = "mammalia-primate-association/mammalia-primate-association.edges"
largeG = tn.read_interactions(location,sep=" ",columns=["n1","n2","__","time"])
tts=[10,5,2,1]
primate = compute_stats(largeG,tts)
```

```
nb_interactions: 1340 nb_unique_Edges: 280 nb_time: 19 nb_nodes: 25
nb intervals:  827
sn_m : 8001.270089029274
ls : 9482.202038052455
ig : 10816.05127727374
sn_e : 12702.711746563129
graph will be loaded as:  <class 'tnetwork.dyn_graph.dyn_graph_sn.DynGraphSN'>
==== 10  ====
==== 5  ====
==== 2  ====
==== 1  ====
```

[58]:
```
long = pd.melt(SP_hospital,id_vars=['tts'],value_vars=names)
long["value"]=long["value"]
long["encoding"]=long["variable"]
ax = sns.lineplot(x="tts",y="value",data=long,hue="encoding",markers=True,style=
↪"encoding")
ax.set_xscale('log')
ax.set_yscale('log')
ax.set_xlabel("window size")
ax.set_ylabel("code length")
print_lines(long)
plt.savefig('encoding/hospital.pdf')
```



[63]:
```
long = pd.melt(SP_PS,id_vars=['tts'],value_vars=names)
long["value"]=long["value"]
long["encoding"]=long["variable"]
ax = sns.lineplot(x="tts",y="value",data=long,hue="encoding",markers=True,style=
↪"encoding")
ax.set_xscale('log')
ax.set_yscale('log')
ax.set_xlabel("window size")
ax.set_ylabel("code length")
print_lines(long)
plt.savefig('encoding/PS.pdf')
```

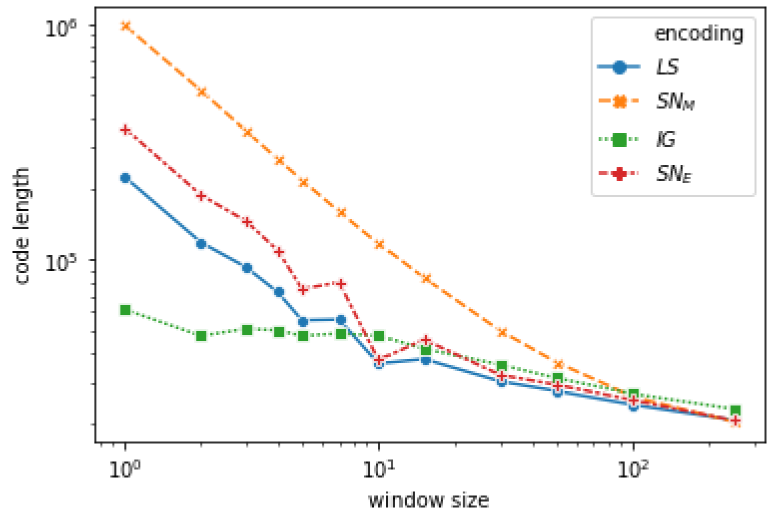


[64]:
```
long = pd.melt(GOT,id_vars=['tts'],value_vars=names)
long["value"]=long["value"]
long["encoding"]=long["variable"]
ax = sns.lineplot(x="tts",y="value",data=long,hue="encoding",markers=True,style=
↪"encoding")
ax.set_xscale('log')
ax.set_yscale('log')
```

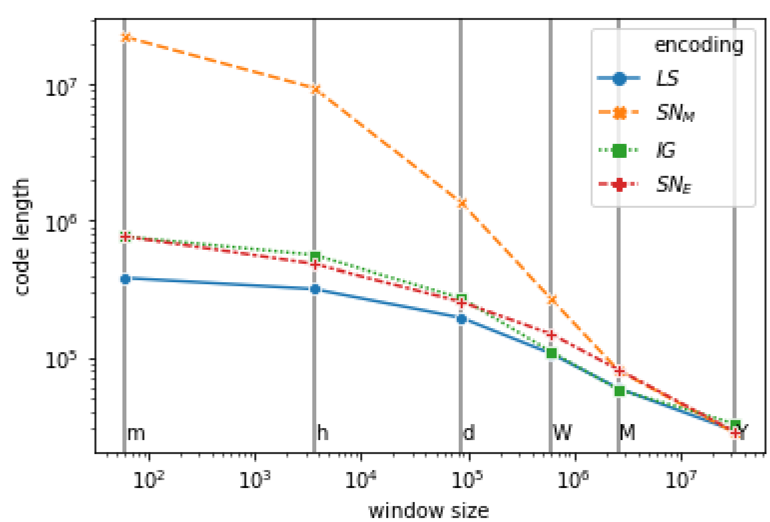
(continues on next page)

(continued from previous page)

```
ax.set_xlabel("window size")
ax.set_ylabel("code length")
plt.savefig('encoding/GOT.pdf')
```



[65]:
```
long = pd.melt(ENRON,id_vars=['tts'],value_vars=names)
long["value"]=long["value"]
long["encoding"]=long["variable"]
ax = sns.lineplot(x="tts",y="value",data=long,hue="encoding",markers=True,style=
↪"encoding")
ax.set_xscale('log')
ax.set_yscale('log')
ax.set_xlabel("window size")
ax.set_ylabel("code length")
print_lines(long)
plt.savefig('encoding/ENRON.pdf')
```

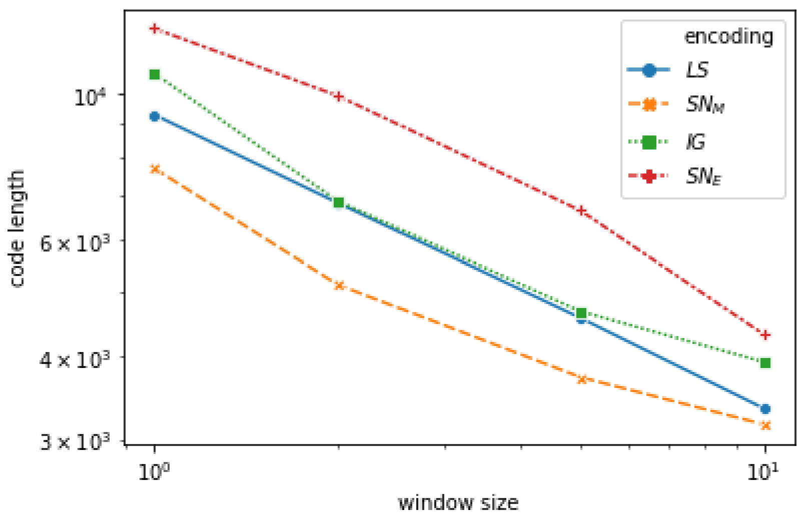


[66]:
```
long = pd.melt(primate,id_vars=['tts'],value_vars=names)
long["value"]=long["value"]
```

(continues on next page)

(continued from previous page)

```
long["encoding"]=long["variable"]
ax = sns.lineplot(x="tts",y="value",data=long,hue="encoding",markers=True,style=
↪"encoding")
ax.set_xscale('log')
ax.set_yscale('log')
ax.set_xlabel("window size")
ax.set_ylabel("code length")
plt.savefig('encoding/primates.pdf')
```




```
[ ]:
```

2.4 Documentation

2.4.1 Dynamic Network Classes

A simple demo of usage can be found here.

Introduction

Dynamic graphs can be represented as:

- Sequences of snapshots
- Interval Graphs
- Link streams

Each representation has strengths and weaknesses. The representation to use depends on

1. Algorithms we wish to use
2. Information we need to access to efficiently
3. Properties of the network to represent.

In summary, the properties of each representation are the following:

Sequences of snapshots

Time is discrete. Interactions are ponctual.

Most appropriate if there are a few timesteps (<50?), or if you need to access efficiently the network at a given time.

Inefficient to access the list of all interactions of a particular node/edge.

Interval Graph

Time is continuous. Interactions have a duration.

Most appropriate when observed relations last a consequent time relatively to the whole period of study, i.e., if the original data is continuous or if it is discrete but an edge observed at time t tends to be also present from t to t+n, with n large.

Efficient to access all the interactions of a node or a pair of nodes, but not to access all interactions at a particular time.

Link Streams:

Time is continuous. Interactions are ponctual.

Most appropriate when interactions are rare compared to the frequency of observation. For instance, an email dataset in which each emails timestamp is at the level of the second.

Efficient to access all the interactions of a node or a pair of nodes, but not to access all interactions at a particular time.

Automatic model selection

As introduced in *Data compression to choose a proper dynamic network representation* (TBP), the library propose to choose automatically the representation when provided with a file containing interactions as triplets <Time, Node1,Node2>. The method is based on the most efficient data compression. Check the *Read/Write* section to know more.

Shared methods

All representation share a set of common fonctions to access and modify them. Note that the implementation of those methods vary.

Those methods are:

`start`()	First valid date of the data
`end`()	Last valid date of the data
`summary`()	Print a summary of the graph
`add_node_presence`(node, time)	Add presence of a node
`add_nodes_presence_from`(nodes, times)	Add nodes at times
`add_interaction`(u, v, time)	Add an interaction at a time
`add_interactions_from`(nodePairs, times)	Add interactions at times
`remove_node_presence`(node, time)	Remove a node presence
`remove_interaction`(u, v, time)	Remove an interaction at a time
`remove_interactions_from`(nodePairs, times)	Remove interactions at times
`edge_presence`([nbunch])	Return presence time of edges

Continued on next page

Table 1 – continued from previous page

interactions()	Return all interactions as a set
change_times()	Return all times with interactions/change
graph_at_time(t)	Return graph at a time
cumulated_graph([times])	Return the cumulated graph over a period
slice(start, end)	Return a slice of the temporal network
aggregate_sliding_window([bin_size, shift, …])	Aggregate using sliding windows
frequency(value)	Set and/or return graph frequency
write_interactions(filename)	Export custom format with only interactions

tnetwork.dyn_graph.dyn_graph.DynGraph.start

DynGraph.**start**()

First valid date of the data

tnetwork.dyn_graph.dyn_graph.DynGraph.end

DynGraph.**end**()

Last valid date of the data

tnetwork.dyn_graph.dyn_graph.DynGraph.summary

DynGraph.**summary**()

Print a summary of the graph

tnetwork.dyn_graph.dyn_graph.DynGraph.add_node_presence

DynGraph.**add_node_presence**(*node*, *time*)

Add presence of a node

tnetwork.dyn_graph.dyn_graph.DynGraph.add_nodes_presence_from

DynGraph.**add_nodes_presence_from**(*nodes*, *times*)

Add nodes at times

tnetwork.dyn_graph.dyn_graph.DynGraph.add_interaction

DynGraph.**add_interaction**(*u*, *v*, *time*)

Add an interaction at a time

tnetwork.dyn_graph.dyn_graph.DynGraph.add_interactions_from

DynGraph.**add_interactions_from**(*nodePairs*, *times*)

Add interactions at times

tnetwork.dyn_graph.dyn_graph.DynGraph.remove_node_presence

DynGraph.**remove_node_presence**(*node*, *time*)
Remove a node presence

tnetwork.dyn_graph.dyn_graph.DynGraph.remove_interaction

DynGraph.**remove_interaction**(*u*, *v*, *time*)
Remove an interaction at a time

tnetwork.dyn_graph.dyn_graph.DynGraph.remove_interactions_from

DynGraph.**remove_interactions_from**(*nodePairs*, *times*)
Remove interactions at times

tnetwork.dyn_graph.dyn_graph.DynGraph.edge_presence

DynGraph.**edge_presence**(*nbunch=None*)
Return presence time of edges

tnetwork.dyn_graph.dyn_graph.DynGraph.interactions

DynGraph.**interactions**()
Return all interactions as a set

Returns a set of pairs ((n1,n2),time)

tnetwork.dyn_graph.dyn_graph.DynGraph.change_times

DynGraph.**change_times**()
Return all times with interactions/change

tnetwork.dyn_graph.dyn_graph.DynGraph.graph_at_time

DynGraph.**graph_at_time**(*t*)
Return graph at a time

tnetwork.dyn_graph.dyn_graph.DynGraph.cumulated_graph

DynGraph.**cumulated_graph**(*times=None*)
Return the cumulated graph over a period

tnetwork.dyn_graph.dyn_graph.DynGraph.slice

`DynGraph.`**`slice`**(*start*, *end*)

Return a slice of the temporal network

Parameters

- **`start`** – start of the slice
- **`end`** – end of the slice

Returns

tnetwork.dyn_graph.dyn_graph.DynGraph.aggregate_sliding_window

`DynGraph.`**`aggregate_sliding_window`**(*bin_size=None*, *shift=None*, *t_start=None*, *t_end=None*, *weighted=True*)

Aggregate using sliding windows

tnetwork.dyn_graph.dyn_graph.DynGraph.frequency

`DynGraph.`**`frequency`**(*value: int = None*)

Set and/or return graph frequency

The frequency of a dynamic network is the smallest possible difference between two consecutive observations. Note that if for some reason you really need continuous value, you can set the frequency to -1, but you will need to set explicitely the temporality every time it is needed for a computation (conversion between formats, visualization, etc)

Parameters **`value`** – if None, the frequency is not changed. If -1, time is considered continuous.

Returns current frequency value

tnetwork.dyn_graph.dyn_graph.DynGraph.write_interactions

`DynGraph.`**`write_interactions`**(*filename*)

Export custom format with only interactions

Sequences of snapshots

`class` `tnetwork.`**`DynGraphSN`**(*data=None*, *frequency=1*)

A class to represent dynamic graphs as snapshot sequence.

Each snapshot is represented as a networkx graph, and is associated to a time step identifier. The time step can be an position in the sequence (1,2,3,. . .) or an arbitrary temporal indicator (year, timestamp. . .).

Snpashots are ordered according to their time step identifier using a sorted dictionary (SortedDict).

Adding and removing nodes and edges

`DynGraphSN.__init__`([data, frequency])	Instanciate a new graph, with or without initial data
`DynGraphSN.add_node_presence`(n, time)	Add presence for a node at a time

Continued on next page

Table 2 – continued from previous page

DynGraphSN.add_nodes_presence_from(nodes, times)	Add nodes for times
DynGraphSN.add_interaction(u, v, time)	Add a single interaction at a single time step.
DynGraphSN.add_interactions_from(nodePairs, …)	Add interactions between the provided node pairs for the provided times.
DynGraphSN.remove_node_presence(n, time)	Remove presence for a node at a time
DynGraphSN.remove_interaction(u, v, time)	Remove a single interaction at a single time step.
DynGraphSN.remove_interactions_from(…)	Remove interactions between the provided node pairs for the provided times.
DynGraphSN.add_snapshot([t, graphSN])	Add a snapshot for a time step t
DynGraphSN.remove_snapshot(t)	Remove a snapshot
DynGraphSN.discard_empty_snapshots()	Discard snapshots with no edges

tnetwork.DynGraphSN.__init__

DynGraphSN.**__init__**(*data=None*, *frequency=1*)

Instanciate a new graph, with or without initial data

Parameters

- **data** – can be a dictionary {time step:graph} or a list of graph, in which sase time steps are integers starting at 0
- **frequency** – minimal time difference between two observations. Default: 1

tnetwork.DynGraphSN.add_node_presence

DynGraphSN.**add_node_presence**(*n*, *time*)

Add presence for a node at a time

Parameters

- **n** – node
- **time** – a snapshot time

tnetwork.DynGraphSN.add_nodes_presence_from

DynGraphSN.**add_nodes_presence_from**(*nodes*, *times*)

Add nodes for times

For each node in nodes, add it for each time in times.

Parameters

- **nodes** – list of nodes, or a single node
- **times** – list of times of same length as node, or a single time

tnetwork.DynGraphSN.add_interaction

DynGraphSN.**add_interaction**(*u*, *v*, *time*)

Add a single interaction at a single time step.

Parameters

- **u** – first node
- **v** – second node
- **time** – time step identifier

tnetwork.DynGraphSN.add_interactions_from

DynGraphSN.**add_interactions_from**(*nodePairs*, *times*)

Add interactions between the provided node pairs for the provided times.

Add each provided nodePair at each provided time

Parameters

- **nodePairs** – list of pairs of nodes, or a single pair of nodes as a tuple or set
- **times** – list of times as integer or a single integer

tnetwork.DynGraphSN.remove_node_presence

DynGraphSN.**remove_node_presence**(*n*, *time*)

Remove presence for a node at a time

Parameters

- **n** – node
- **time** – a snapshot time

tnetwork.DynGraphSN.remove_interaction

DynGraphSN.**remove_interaction**(*u*, *v*, *time*)

Remove a single interaction at a single time step.

Note: it does not remove the node

Parameters

- **u** – first node
- **v** – second node
- **time** – time step identifier

tnetwork.DynGraphSN.remove_interactions_from

DynGraphSN.**remove_interactions_from**(*nodePairs*, *times*)

Remove interactions between the provided node pairs for the provided times.

If one of the two parameters is a single element, will remove the node pair at all provided time steps, or all the node pairs at the provided time step.

Parameters

- **nodePairs** – list of pairs of nodes, or a single pair of nodes
- **times** – list of times for this node, or a single time

Returns

tnetwork.DynGraphSN.add_snapshot

`DynGraphSN.add_snapshot(t=None, graphSN=None)`

Add a snapshot for a time step t

Parameters

- **t** – the time step identifier. If none, the last one + 1
- **graphSN** – the graph to add (networkx object), if None, add an empty snapshot

tnetwork.DynGraphSN.remove_snapshot

`DynGraphSN.remove_snapshot(t)`

Remove a snapshot

Parameters **t** – the time at which to remove a snapshot

Returns

tnetwork.DynGraphSN.discard_empty_snapshots

`DynGraphSN.discard_empty_snapshots()`

Discard snapshots with no edges

Accessing the graph

DynGraphSN.summary()	Print a summary of the graph
DynGraphSN.snapshots([t])	Return all snapshots or a particular one
DynGraphSN.node_presence([nodes])	Presence time of nodes
DynGraphSN.edge_presence([edges])	Presence time of edges
DynGraphSN.graph_at_time(t)	Return the graph as it is at time t
DynGraphSN.snapshots_timesteps()	Return the list of time steps
DynGraphSN.last_snapshot()	Return the last snapshot
DynGraphSN.start()	Time of the first snapshot
DynGraphSN.end()	Time of the last snapshot
DynGraphSN.change_times()	Times of non-empty snapshots
DynGraphSN.frequency(value)	Set and/or return graph frequency

tnetwork.DynGraphSN.summary

`DynGraphSN.summary()`

Print a summary of the graph

tnetwork.DynGraphSN.snapshots

`DynGraphSN.snapshots(t=None)`

Return all snapshots or a particular one

Default: return a sorted dictionary, key: the time information, value: a networkx graph. If t is provided, return graph at that particular time

Parameters `t` – the time of the snapshot to return

Returns

tnetwork.DynGraphSN.node_presence

`DynGraphSN.`**`node_presence`**`(`*nodes=None*`)`

Presence time of nodes

Several usages:

- If nodes==None (default), return a dict for each note, its existing times
- If nodes is a single node, return the interval of presence of this node
- If nodes is a set of nodes, return interval of presence of those nodes as a dictionary

Parameters `nodes` – list of ndoes

Returns a dictionary, key:node, value: list of time steps

tnetwork.DynGraphSN.edge_presence

`DynGraphSN.`**`edge_presence`**`(`*edges=None*`)`

Presence time of edges

Several usages:

- If edges==None (default), return a dict for each edge, its existing times
- If edges is a set of edges, return interval of presence of those edges as a dictionary

Parameters `edges` – list of edges

Returns a dictionary, key:edge(pair), value: list of time steps

tnetwork.DynGraphSN.graph_at_time

`DynGraphSN.`**`graph_at_time`**`(`*t*`)`

Return the graph as it is at time t

Parameters `t` – a time step identifier

Returns the graph as a networkx graph

tnetwork.DynGraphSN.snapshots_timesteps

`DynGraphSN.`**`snapshots_timesteps`**`()`

Return the list of time steps

Returns list of time steps

tnetwork.DynGraphSN.last_snapshot

`DynGraphSN.last_snapshot()`

Return the last snapshot

Returns the last snapshot as a networkx graph

tnetwork.DynGraphSN.start

`DynGraphSN.start()`

Time of the first snapshot

Returns

tnetwork.DynGraphSN.end

`DynGraphSN.end()`

Time of the last snapshot

Returns

tnetwork.DynGraphSN.change_times

`DynGraphSN.change_times()`

Times of non-empty snapshots

Returns list of times

tnetwork.DynGraphSN.frequency

`DynGraphSN.frequency(value: int = None)`

Set and/or return graph frequency

The frequency of a dynamic network is the smallest possible difference between two consecutive observations. Note that if for some reason you really need continuous value, you can set the frequency to -1, but you will need to set explicitely the temporality every time it is needed for a computation (conversion between formats, visualization, etc)

Parameters **value** – if None, the frequency is not changed. If -1, time is considered continuous.

Returns current frequency value

Conversion to different formats

DynGraphSN.to_DynGraphIG()	Convert the graph into a DynGraph_IG.
DynGraphSN.to_DynGraphLS()	Convert to a linkstream
DynGraphSN.to_tensor([always_all_nodes])	Return a tensor representation

tnetwork.DynGraphSN.to_DynGraphIG

DynGraphSN.**to_DynGraphIG**()

Convert the graph into a DynGraph_IG.

##Can be optimized !

Returns

tnetwork.DynGraphSN.to_DynGraphLS

DynGraphSN.**to_DynGraphLS**()

Convert to a linkstream

Currently, conserve only edges :return:

tnetwork.DynGraphSN.to_tensor

DynGraphSN.**to_tensor**(*always_all_nodes=True*)

Return a tensor representation

Compute the list of matrices corresponding to each graph, with nodes ordered in a same order And the dic of nodes corresponding and the list for each sn of nodes :param always_all_nodes: if True, even if a node is not active during a snapshot, it is included in the matrix :return: 3 elements:(A,B,C) A: list of numpy matrices, B: a bidictionary {node name:node order in the matrix}, C: active node at each step, as a list of list of nodes

Aggregation

DynGraphSN.cumulated_graph([times])	Compute the cumulated graph.
DynGraphSN.slice(start, end)	Keep only the selected period
DynGraphSN.aggregate_sliding_window([…])	Return a new dynamic graph without modifying the original one, aggregated using sliding windows of the desired size.
DynGraphSN.aggregate_time_period(period[, …])	Aggregate graph by time period (day, year, …)

tnetwork.DynGraphSN.cumulated_graph

DynGraphSN.**cumulated_graph**(*times=None*)

Compute the cumulated graph.

Return a networkx graph corresponding to the cumulated graph of the given period (whole graph by default)

Parameters **times** – list/set of time steps ID of snapshots to cumulate. Default (None) means all snapshots

Returns a networkx (weighted) graph

tnetwork.DynGraphSN.slice

`DynGraphSN.`**`slice`**(*start*, *end*)

Keep only the selected period

Parameters

- **`start`** – time of the beginning of the slice
- **`end`** – time of the end of the slice

tnetwork.DynGraphSN.aggregate_sliding_window

`DynGraphSN.`**`aggregate_sliding_window`**(*bin_size=None*, *shift=None*, *t_start=None*, *t_end=None*, *weighted=True*)

Return a new dynamic graph without modifying the original one, aggregated using sliding windows of the desired size. If Shift is not provided or equal to bin_size, windows are non overlapping. If no parameter is provided, creates a single graph aggregating the whole period. Yielded graphs are weighted (weight: number of apparition of edges during the period)

Parameters

- **`bin_size`** – desired size of bins, in the internal time unit (not necessarily equals to the number of snapshot_affiliations)
- **`shift`** – time distance (shift) between the start of two successive bins, in the internal time unit (not necessarily number of sn)
- **`t_start`** – time step to start the binning (default: first)
- **`t_end`** – time step (not included) to stop the binning (default: last)

Returns a DynGraph_SN object

tnetwork.DynGraphSN.aggregate_time_period

`DynGraphSN.`**`aggregate_time_period`**(*period*, *step_to_datetime=<built-in method utcfromtimestamp of type object>*)

Aggregate graph by time period (day, year, . . .)

Return a new dynamic graph without modifying the original one, aggregated such as all Yielded graphs are weighted (weight: number of apparition of edges during the period)

Parameters

- **`period`** – either a string (minute,hour,day,week,month,year) or a function returning the timestamp truncated to the start of the desired period
- **`step_to_datetime`** – function to convert time step to a datetime object, default is utfromtimestamp

Returns a DynGraph_SN object

Other graph operations

DynGraphSN.apply_nx_function(function[, ...])	Apply a networkx function to each snapshot and return the list of result.
DynGraphSN.code_length([as_matrix, as_edgelist])	
DynGraphSN.write_interactions(filename)	Write interactions in a file

tnetwork.DynGraphSN.apply_nx_function

DynGraphSN.**apply_nx_function**(*function*, *start=None*, *stop=None*, ***kwargs*)

Apply a networkx function to each snapshot and return the list of result. Parameters of the function to apply can be passed as parameter to this function. example >>> dg = DynGraphSN.graph_socioPatterns2012() >>> dg.apply_nx_function(nx.nodes) >>> dg.apply_nx_function(nx.Graph.add_node,node_for_adding="nodeTest")

Parameters **function** – the networkx function

Returns the list of results for each snapshot

tnetwork.DynGraphSN.code_length

DynGraphSN.**code_length**(*as_matrix=True*, *as_edgelist=True*)

tnetwork.DynGraphSN.write_interactions

DynGraphSN.**write_interactions**(*filename*)

Write interactions in a file

Write in corresponding json format

Parameters **filename** –

Returns

Interval graphs

class tnetwork.**DynGraphIG**(*edges=None*, *nodes=None*, *start=None*, *end=None*, *frequency=1*)

A class to represent dynamic graphs as interval graphs.

It is represented using a networkx Graph, using an attribute ("t") for each node and each edge representing its periods of presence. The representation is done using the class Intervals (tnetwork.utils.intervals) Time steps are represented by integers, that can correspond to an arbitrary scale (1,2,3,...) or to timestamps in order to represent dates.

Examples

Adding and removing nodes and edges

DynGraphIG.__init__([edges, nodes, start, ...])	Instanciate a dynamic graph

Continued on next page

Table 7 – continued from previous page

DynGraphIG.add_node_presence(n, time)	Add presence for a node for a period
DynGraphIG.add_nodes_presence_from(nodes, times)	Add interactions between provided pairs for the provided periods
DynGraphIG.add_interaction(u, v, time)	Add an interaction between nodes u and v at time time
DynGraphIG.add_interactions_from(nodePairs, ...)	Add interactions between provided pairs for the provided periods
DynGraphIG.remove_node_presence(node, time)	Remove node and its interactions over the period
DynGraphIG.remove_interaction(u, v, time)	Remove an interaction between nodes u and v at time time
DynGraphIG.remove_interactions_from(...)	Remove interactions between provided pairs for the provided periods

tnetwork.DynGraphIG.__init__

DynGraphIG.**__init__**(*edges=None*, *nodes=None*, *start=None*, *end=None*, *frequency=1*)

Instanciate a dynamic graph

A start end end dates can be used to give a "duration" to the graph independently from its nodes and edges (for instance, to study activity during a whole year, the graph might start on January 1st at 00:00 while the first recorded activity occurs in the afternoon or on another day)

Parameters

- **start** – set a start time, by default will be the first time of the added affiliations
- **end** – set an end time, by default will be the last time of the added affiliations
- **edges** – data to initialize the dynamic graph, dictionary {(n1,n2):time}. Keys are edges, time is Intervals object
- **nodes** – data to initialize the dynamic graph, dictionary {n:time}. Keys are ndoes, time is Intervals object

tnetwork.DynGraphIG.add_node_presence

DynGraphIG.**add_node_presence**(*n*, *time*)

Add presence for a node for a period

Parameters

- **n** – node
- **time** – a period, couple (start, stop) or an interval

tnetwork.DynGraphIG.add_nodes_presence_from

DynGraphIG.**add_nodes_presence_from**(*nodes*, *times*)

Add interactions between provided pairs for the provided periods

Parameters

- **nodes** – list of nodes, or a single node
- **times** – list of times defined as couple (start, stop) , of same length as node, or a single time

tnetwork.DynGraphIG.add_interaction

DynGraphIG.**add_interaction**(*u*, *v*, *time*)
Add an interaction between nodes u and v at time time

Parameters

- **u** – first node
- **b** – second node
- **time** – pair (start,end) or Intervals

Returns

tnetwork.DynGraphIG.add_interactions_from

DynGraphIG.**add_interactions_from**(*nodePairs*, *times*)
Add interactions between provided pairs for the provided periods

Add each provided nodePair at each provided time

param nodePairs list of pairs of nodes, or a single pair of nodes as a tuple or set

param times a single time or a list of times, as pair (start,end) or an Interval Object

tnetwork.DynGraphIG.remove_node_presence

DynGraphIG.**remove_node_presence**(*node*, *time*)
Remove node and its interactions over the period

Parameters

- **node** – node to remove
- **time** – a period, couple (start, stop) or an interval

tnetwork.DynGraphIG.remove_interaction

DynGraphIG.**remove_interaction**(*u*, *v*, *time*)
Remove an interaction between nodes u and v at time time

Parameters

- **u** – first node
- **v** – second node
- **time** – pair (start,end)

Returns

tnetwork.DynGraphIG.remove_interactions_from

DynGraphIG.**remove_interactions_from**(*nodePairs*, *times*)
Remove interactions between provided pairs for the provided periods

Parameters

- **nodePairs** – a list of node pairs
- **times** – a pair of time step of the form (start,stop), or a list of pair of time step of same length as nodePairs

Accessing the graph

DynGraphIG.summary()	Print a summary of the graph
DynGraphIG.node_presence([nodes])	Presence period of nodes
DynGraphIG.edge_presence([edges, as_intervals])	Return the periods of interactions for each pair of nodes with at least an interaction
DynGraphIG.graph_at_time(t)	Graph as it is at time t
DynGraphIG.interactions()	Return all interactions as a set
DynGraphIG.interactions_intervals([edges])	Return the periods of interactions for each pair of nodes with at least an interaction
DynGraphIG.change_times()	List of all times with a node/edge change
DynGraphIG.start()	First valid date of the data
DynGraphIG.end()	Last valid date of the data

tnetwork.DynGraphIG.summary

DynGraphIG.**summary**()

Print a summary of the graph

tnetwork.DynGraphIG.node_presence

DynGraphIG.**node_presence**(*nodes=None*)

Presence period of nodes

Several usages:

- If nodes==None (default), return a dict for each node, its existing times
- If nodes is a single node, return the interval of presence of this node
- If nodes is a set of nodes, return interval of presence of those nodes as a dictionary

Parameters **nodes** –

Returns dictionary, for each node, its presence Intervals, or single Interval for single node

tnetwork.DynGraphIG.edge_presence

DynGraphIG.**edge_presence**(*edges=None*, *as_intervals=False*)

Return the periods of interactions for each pair of nodes with at least an interaction

Parameters **edges** – the list of edges to get interactions for, all by default

Returns dictionary, keys : pair of nodes, values : an interval object

tnetwork.DynGraphIG.graph_at_time

`DynGraphIG.graph_at_time(t: int)` → <Mock id='139756918186256'>

Graph as it is at time t

Return a networkx graph corresponding to the graphs as it is at time t, i.e., edges and nodes present at that time

Parameters `t` – timestep

Returns a networkx Graph

tnetwork.DynGraphIG.interactions

`DynGraphIG.interactions()`

Return all interactions as a set

Returns a set of pairs ((n1,n2),time)

tnetwork.DynGraphIG.interactions_intervals

`DynGraphIG.interactions_intervals(edges=None)`

Return the periods of interactions for each pair of nodes with at least an interaction

Parameters `edges` – the list of edges to get interactions for, all by default

Returns dictionary, keys : pair of nodes, values : an interval object

tnetwork.DynGraphIG.change_times

`DynGraphIG.change_times()` → [<class 'int'>]

List of all times with a node/edge change

Return the list of all times at which a change (new edge, end of edge, node appear/disappear) occurs :return: list of int

tnetwork.DynGraphIG.start

`DynGraphIG.start()`

First valid date of the data

tnetwork.DynGraphIG.end

`DynGraphIG.end()`

Last valid date of the data

Conversion to different formats

DynGraphIG.to_DynGraphSN([slices, discard_empty])	Convert to a snapshot representation.

tnetwork.DynGraphIG.to_DynGraphSN

DynGraphIG.**to_DynGraphSN**(*slices=None*, *discard_empty=True*)

Convert to a snapshot representation.

Parameters **slices** – can be one of

- None, snapshot_affiliations are created such as a new snapshot is created at every node/edge change,
- an integer, snapshot_affiliations are created using a sliding window
- a list of periods, represented as pairs (start, end), each period yielding a snapshot

Parameters **discard_empty** – if True, the returned dynamic network won't have empty snapshots

Returns a dynamic graph represented as snapshot_affiliations, the weight of nodes/edges correspond to their presence time during the snapshot

Aggregation

DynGraphIG.cumulated_graph([times])	Compute the cumulated graph.
DynGraphIG.slice(start, end)	Keep only the selected period
DynGraphIG.code_length()	
DynGraphIG.write_interactions(filename)	Write a file with interactions

tnetwork.DynGraphIG.cumulated_graph

DynGraphIG.**cumulated_graph**(*times=None*)

Compute the cumulated graph.

Return a networkx graph corresponding to the cumulated graph of the given period (whole graph by default)

Parameters **times** – Intervals object or list of pairs (start, end)

Returns a networkx (weighted) graph

tnetwork.DynGraphIG.slice

DynGraphIG.**slice**(*start*, *end*)

Keep only the selected period

Parameters

- **start** – time of the beginning of the slice
- **end** – time of the end of the slice

tnetwork.DynGraphIG.code_length

DynGraphIG.**code_length**()

tnetwork.DynGraphIG.write_interactions

DynGraphIG.**write_interactions**(*filename*)
Write a file with interactions

Write interactions in the corresponding json format

Parameters **filename** –

Returns

Link Streams

class tnetwork.**DynGraphLS**(*edges=None*, *nodes=None*, *frequency=1*, *start=None*, *end=None*)

A class to represent dynamic graphs as link streams.

It is represented using a networkx Graph, using an attribute ("t") for each node and each edge representing its time of presence. The representation is done using a list of integer.

Adding and removing nodes and edges

DynGraphLS.__init__([edges, nodes, ...])	Instanciate a dynamic graph
DynGraphLS.start()	First valid date of the data
DynGraphLS.end()	Last valid date of the data
DynGraphLS.add_interaction(u, v, time)	Add an interaction between nodes u and v at time time
DynGraphLS.add_interactions_from(nodePairs, ...)	Add interactions between the provided node pairs for the provided times.
DynGraphLS.add_node_presence(n, time)	Add presence for a node for a period
DynGraphLS.add_nodes_presence_from(nodes, times)	Add interactions between provided pairs for the provided periods
DynGraphLS.remove_node_presence(node, time)	Remove node and its interactions over the period
DynGraphLS.remove_interaction(u, v, time)	Remove an interaction between nodes u and v at time time
DynGraphLS.remove_interactions_from(...)	Remove interactions between provided pairs for the provided periods

tnetwork.DynGraphLS.__init__

DynGraphLS.**__init__**(*edges=None*, *nodes=None*, *frequency=1*, *start=None*, *end=None*)
Instanciate a dynamic graph

A start and end dates can be used to give a "duration" to the graph independently from its nodes and edges (for instance, to study activity during a whole year, the graph might start on January 1st at 00:00 while the first recorded activity occurs in the afternoon or on another day)

Parameters

- **start** – set a start time, by default will be the first added time
- **end** – set an end time, by default will be the last added time
- **frequency** – minimal time difference between two observations. Default: 1

- **edges** – data to initialize the dynamic graph, dictionary {(n1,n2):[int]}. Keys are edges, time is ordered list of int
- **nodes** – data to initialize the dynamic graph, dictionary {n:time}. Keys are nodes, time is Intervals object (see interval graph)

tnetwork.DynGraphLS.start

`DynGraphLS.start()`

First valid date of the data

tnetwork.DynGraphLS.end

`DynGraphLS.end()`

Last valid date of the data

tnetwork.DynGraphLS.add_interaction

`DynGraphLS.add_interaction(u, v, time)`

Add an interaction between nodes u and v at time time

Parameters

- **u** – first node
- **b** – second node
- **time** – integer or list of integers

Returns

tnetwork.DynGraphLS.add_interactions_from

`DynGraphLS.add_interactions_from(nodePairs, times)`

Add interactions between the provided node pairs for the provided times.

Add each provided nodePair at each provided time

Parameters

- **nodePairs** – list of pairs of nodes, or a single pair of nodes as a tuple or set
- **times** – list of times as integer or a single integer

tnetwork.DynGraphLS.add_node_presence

`DynGraphLS.add_node_presence(n, time)`

Add presence for a node for a period

Parameters

- **n** – node
- **time** – a period, couple (start, stop) or an interval

tnetwork.DynGraphLS.add_nodes_presence_from

DynGraphLS.**add_nodes_presence_from**(*nodes*, *times*)
Add interactions between provided pairs for the provided periods

Parameters

- **nodes** – list of nodes, or a single node
- **times** – list of times or a single time (integer)

tnetwork.DynGraphLS.remove_node_presence

DynGraphLS.**remove_node_presence**(*node*, *time*)
Remove node and its interactions over the period

Parameters

- **node** – node to remove
- **time** – a period, couple (start, stop) or an interval

tnetwork.DynGraphLS.remove_interaction

DynGraphLS.**remove_interaction**(*u*, *v*, *time*)
Remove an interaction between nodes u and v at time time

Parameters

- **u** – first node
- **v** – second node
- **time** – integer

Returns

tnetwork.DynGraphLS.remove_interactions_from

DynGraphLS.**remove_interactions_from**(*nodePairs*, *times*)
Remove interactions between provided pairs for the provided periods

Parameters

- **nodePairs** – a node pair, or a list of node pairs
- **times** – a list of integer (applied to all pairs) or a list of lsit of integer (one per nodePairs)

Accessing the graph

DynGraphLS.summary()	Print a summary of the graph
DynGraphLS.interactions()	Return all interactions as a set
DynGraphLS.node_presence([nodes])	Presence period of nodes
DynGraphLS.edge_presence([edges])	Return the periods of interactions for each pair of nodes with at least an interaction

Continued on next page

Table 12 – continued from previous page

DynGraphLS.graph_at_time(t)	Graph as it is at time t
DynGraphLS.change_times()	List of all times with a node/edge change

tnetwork.DynGraphLS.summary

DynGraphLS.**summary**()

Print a summary of the graph

tnetwork.DynGraphLS.interactions

DynGraphLS.**interactions**()

Return all interactions as a set

Returns a set of pairs ((n1,n2),time)

tnetwork.DynGraphLS.node_presence

DynGraphLS.**node_presence**(*nodes=None*)

Presence period of nodes

Several usages:

- If nodes==None (default), return a dict for each node, its existing times
- If nodes is a single node, return the interval of presence of this node
- If nodes is a set of nodes, return interval of presence of those nodes as a dictionary

Parameters **nodes** –

Returns dictionary, for each node, its presence Intervals, or single Interval for single node

tnetwork.DynGraphLS.edge_presence

DynGraphLS.**edge_presence**(*edges=None*)

Return the periods of interactions for each pair of nodes with at least an interaction

Parameters **edges** – the list of edges to get interactions for, all by default

Returns dictionary, keys : pair of nodes, values : list of integer

tnetwork.DynGraphLS.graph_at_time

DynGraphLS.**graph_at_time**(*t: int*) → <Mock id=’139757002396304’>

Graph as it is at time t

Return a networkx graph corresponding to the graphs as it is at time t, i.e., edges and nodes present at that time

Parameters **t** – timestep

Returns a networkx Graph

tnetwork.DynGraphLS.change_times

DynGraphLS.**change_times**() → [<class 'int'>]

List of all times with a node/edge change

Return the list of all times at which a change node change/link :return: list of int

Conversion to different formats

DynGraphLS.to_DynGraphSN([slices, weighted])	Convert to a snapshot representation.

tnetwork.DynGraphLS.to_DynGraphSN

DynGraphLS.**to_DynGraphSN**(*slices=None*, *weighted=True*)

Convert to a snapshot representation.

Parameters **slices** – can be one of

- None, snapshot_affiliations are created according to the frequency of the dynamic network (default one),
- an integer, snapshot_affiliations are created using a sliding window
- a list of periods, represented as pairs (start, end), each period yielding a snapshot

Returns a dynamic graph represented as snapshot_affiliations, the weight of nodes/edges correspond to their presence time during the snapshot

Aggregation

DynGraphLS.cumulated_graph([times, weighted])	Compute the cumulated graph.
DynGraphLS.slice(start, end)	Keep only the selected period
DynGraphLS.aggregate_sliding_window([…])	Return a new dynamic graph without modifying the original one, aggregated using sliding windows of the desired size.

tnetwork.DynGraphLS.cumulated_graph

DynGraphLS.**cumulated_graph**(*times=None*, *weighted=True*)

Compute the cumulated graph.

Return a networkx graph corresponding to the cumulated graph of the given period (whole graph by default)

Parameters **times** – a pair (start,end)

Returns a networkx (weighted) graph

tnetwork.DynGraphLS.slice

DynGraphLS.**slice**(*start*, *end*)

Keep only the selected period

Parameters

- **start** – time of the beginning of the slice (inclusive)
- **end** – time of the end of the slice (exclusive)

tnetwork.DynGraphLS.aggregate_sliding_window

DynGraphLS.**aggregate_sliding_window**(*bin_size=None*, *shift=None*, *t_start=None*, *t_end=None*, *weighted=True*)

Return a new dynamic graph without modifying the original one, aggregated using sliding windows of the desired size. If Shift is not provided or equal to bin_size, windows are non overlapping. If no parameter is provided, creates a single graph aggregating the whole period. Yielded graphs are weighted (weight: number of apparition of edges during the period)

Parameters

- **bin_size** – desired size of bins, in the internal time unit (not necessarily equals to the number of snapshot_affiliations)
- **shift** – time distance (shift) between the start of two successive bins, in the internal time unit (not necessarily number of sn)
- **t_start** – time step to start the binning (default: first)
- **t_end** – time step (not included) to stop the binning (default: last)

Returns a DynGraph_SN object

Other

DynGraphLS.code_length()	
DynGraphLS.write_interactions(filename)	**param filename**

tnetwork.DynGraphLS.code_length

DynGraphLS.**code_length**()

tnetwork.DynGraphLS.write_interactions

DynGraphLS.**write_interactions**(*filename*)

Parameters **filename** –

Returns

2.4.2 Read/Write/Load

Functions to read, write and load dynamic graphs.

Simple example

```
import tnetwork as tn
sn = tn.read_snapshots("file_to_Read")
tn.write_snapshots(sn,"file_to_write")
```

Load example graphs

A few dynamic graphs are already included in the library and can be loaded in one command in the chosen format

graph_socioPatterns2012([format])	Function that return the graph of interactions between students in 2012, from the SocioPatterns project.
graph_socioPatterns_Hospital([format])	Function that return the graph of interactions in the hospital of Lyon between patients and medical staff, from the SocioPatterns project.
graph_socioPatterns_Primary_School([format])	Function that return the graph of interactions between children and teachers, from the SocioPatterns project.
graph_GOT()	Return Game of Thrones temporal network

tnetwork.graph_socioPatterns2012

tnetwork.**graph_socioPatterns2012**(*format=None*)

Function that return the graph of interactions between students in 2012, from the SocioPatterns project. >>> dg = tn.graph_socioPatterns2012()

Returns

tnetwork.graph_socioPatterns_Hospital

tnetwork.**graph_socioPatterns_Hospital**(*format=None*)

Function that return the graph of interactions in the hospital of Lyon between patients and medical staff, from the SocioPatterns project. >>> dg = DynGraphSN.graph_socioPatterns_Hospital()

Returns

tnetwork.graph_socioPatterns_Primary_School

tnetwork.**graph_socioPatterns_Primary_School**(*format=None*)

Function that return the graph of interactions between children and teachers, from the SocioPatterns project. >>> dg = DynGraphSN.graph_socioPatterns_Primary_School()

Returns

tnetwork.graph_GOT

tnetwork.**graph_GOT**()

Return Game of Thrones temporal network

See: https://figshare.com/articles/TV_Series_Networks_of_characters/2199646/11

Returns

Read/Write graphs

Read/Write Generic

read_interactions(file[, frequency, format, . . .])	Read link stream data
from_pandas_interaction_list(interactions, . . .)	

tnetwork.read_interactions

`tnetwork.`**`read_interactions`**(*file*, *frequency=1*, *format=None*, *time_first_column=False*, *sep='\t'*, *columns=None*)

Read link stream data

Parameters

- **file** – file to read
- **frequency** – frequency of data collection, i.e., smallest possible difference between successive timestamps
- **format** – by default, the most efficient format is selected automatically based on encoding length.
- **time_first_column** – If there are only 3 columns, you can use True if time is on the first column adn false if it is on the last
- **sep** – column separator
- **columns** – if there are more than 3 columns, give column names, the used one being "n1", "n2" and "time"

Returns

tnetwork.from_pandas_interaction_list

`tnetwork.`**`from_pandas_interaction_list`**(*interactions*, *format*, *frequency=1*, *source='n1'*, *target='n2'*, *time='time'*)

Read/Write snapshot graphs

read_interactions(file[, frequency, format, . . .])	Read link stream data
read_snapshots(inputDir[, format, . . .])	Read as one file per snapshot
write_snapshots(dynGraph, outputDir, format)	Write one file per snapshot

tnetwork.read_snapshots

`tnetwork.`**`read_snapshots`**(*inputDir: str*, *format=None*, *frequency=1*, *prefix=''*) → tnetwork.dyn_graph.dyn_graph_sn.DynGraphSN

Read as one file per snapshot

Read a dynamic graph as a directory containing one file per snapshot. If the format is not provided, it is infered automatically from file extensions

Parameters

- **inputDir** – directory where the files are located
- **format** – a string among edges(edgelist)|ncol|gefx|gml|pajek|graphML, by default, the extension of the files

Returns a DynGraphSN object

tnetwork.write_snapshots

tnetwork.**write_snapshots**(*dynGraph: tnetwork.dyn_graph.dyn_graph_sn.DynGraphSN*, *outputDir: str*, *format: str = None*)

Write one file per snapshot

Write a dynamic graph as a directory containing one file for each snapshot. The format of files can be chosen.

Parameters

- **dynGraph** – a dynamic graph
- **outputDir** – address of the directory to write
- **format** – default edgelist, choose among edges(edgelist)|ncol|gefx|gml|pajek|graphML

Read/Write interval graphs

read_interactions(file[, frequency, format, …])	Read link stream data
read_period_lists(file_address)	Read as list of periods
write_as_IG(graph, filename)	Write a corresponding json file
write_period_lists(theDynGraph, fileOutput)	Write as list of periods
write_ordered_changes(dynNet, fileOutput[, …])	Write as list of successive changes

tnetwork.read_period_lists

tnetwork.**read_period_lists**(*file_address: str*)

Read as list of periods

Read an interval graph as a list of periods, for the graph, the nodes, and the edges

See write_IG for an explanation of the format

Parameters **file_address** –

tnetwork.write_as_IG

tnetwork.**write_as_IG**(*graph*, *filename*)

Write a corresponding json file

Parameters **filename** –

Returns

tnetwork.write_period_lists

`tnetwork.write_period_lists(theDynGraph: tnetwork.dyn_graph.dyn_graph_ig.DynGraphIG, fileOutput: str)`

Write as list of periods

Write an interval graph graph as a list of periods, for the graph, the nodes, and the edges

Exemple of result:

```
SG  0:100
N   N1  0:10 50:60
N   NODE_3  0:20 30:60
E1  N1  NODE_3  5:10
```

Means that the graph exists from time 0 to 100, it contains 2 nodes (N1 and NODE_3) that exist each over 2 intervals and one edge between those 2 nodes during the interval from 5 to 10

Parameters

- **theDynGraph** – a dynamic graph
- **fileOutput** – the address of the file to write

tnetwork.write_ordered_changes

`tnetwork.write_ordered_changes(dynNet: tnetwork.dyn_graph.dyn_graph_ig.DynGraphIG, fileOutput, dateEveryLine=False, nodeModifications=False, separator='\t', edgeIdentifier='l')`

Write as list of successive changes

(use with caution, not tested recently) Write the dynamic network as a list of successive changes. There are several variants:

- OML :ordered modif list with dates as #DATE and no nodes (Online Modification List)
- OMLN : with nodes
- OMLR : with repeated dates
- OMLRN : nodes and repeated dates

Parameters

- **dynNet** – dynamic network
- **fileOutput** – address of file to write
- **dateEveryLine** – if true, date is repeated for each modification (each line). If false, date modification is on its own line (#DATE) before the modifications happening at this date
- **nodeModifications** – write not only edges but also nodes modifications
- **separator** – choose a separator
- **edgeIdentifier** – character to differenciate edges from nodes.

Read/Write Link Streams

read_interactions(file[, frequency, format, ...])	Read link stream data
read_LS(filename)	Read TS json format
write_as_LS(graph, filename)	**param filename**

tnetwork.read_LS

tnetwork.**read_LS**(*filename*)

Read TS json format

Parameters **filename** –

Returns

tnetwork.write_as_LS

tnetwork.**write_as_LS**(*graph*, *filename*)

Parameters **filename** –

Returns

Read/Write Communities

Read/Write snapshot snapshot_affiliations

write_com_SN(dyn_communities, output_dir[, ...])	Write directory, 1 file = snapshot_affiliations of a snaphshot
read_SN_by_com(inputDir[, sn_id_transformer])	Read directory, 1 file = snapshot_affiliations of a snaphshot

tnetwork.write_com_SN

tnetwork.**write_com_SN**(*dyn_communities: tnetwork.dyn_community.communities_dyn_sn.DynCommunitiesSN*, *output_dir*, *asNodeSet=True*)

Write directory, 1 file = snapshot_affiliations of a snaphshot

Write dynamic snapshot_affiliations as a directory containing one file for each snapshot.

Two possible formats:

Affiliations:

```
node1   com1    com2
node2   com1
node3   com2    com3    com4
```

Node Sets:

```
com:com1    n1   n2   n3
com:another_com    n1    n4   n5
```

Parameters

- **dynGraph** – a dynamic graph
- **outputDir** – address of the directory to write
- **asNodeSet** – if True, node sets, otherwise, snapshot_affiliations

tnetwork.read_SN_by_com

tnetwork.**read_SN_by_com**(*inputDir*, *sn_id_transformer=None*, ***kwargs*)

Read directory, 1 file = snapshot_affiliations of a snaphshot

By default, the name of the file is used as snapshot id. A function can be passed to associate a different ID snapshot to files

The format to read is:

```
node1   com1    com2
node2   com1
node3   com2    com3    com4
...
```

Parameters

- **inputDir** – directory
- **sn_id_transformer** – a function taking a str and
- **kwargs** – a separator can be passed with parameter separator

Returns a dynamic community object

Read/Write interval graph snapshot_affiliations

write_IGC(dyn_communities, outputFile[, …])	Write snapshot_affiliations as interval lists

tnetwork.write_IGC

tnetwork.**write_IGC**(*dyn_communities: tnetwork.dyn_community.communities_dyn_ig.DynCommunitiesIG*, *outputFile*, *renumber=False*)

Write snapshot_affiliations as interval lists

Format is:

```
node1   com1=5:10   com2=10:20
node2   com1=0:100  com3=50:100
```

use with caution, not tested for some time

Parameters

- **dyn_communities** – dynamic snapshot_affiliations
- **outputFile** – address of file to write
- **renumber** – use successive ids instead of original community ids

2.4.3 Visualization

Some methods are proposed to visualize dynamic networks and snapshot_communities. A simple demo of usage can be found here.

Vizualising graphs is already a difficul problem in itself, and adding the dynamic makes it an ever harder task.

We propose two views of the data:

- Using static graphs at the desired step
- Using a longitudinal view of nodes only

`plot_as_graph`(dynamic_graph[, communities, ...])	Plot to see the static graph at each snapshot
`plot_longitudinal`([dynamic_graph, ...])	A longitudinal view of nodes/snapshot_communities

tnetwork.plot_as_graph

tnetwork.**plot_as_graph**(*dynamic_graph*, *communities=None*, *ts=None*, *width=800*, *height=600*, *slider=False*, *to_datetime=False*, *bokeh=False*, *auto_show=False*, ***kwargs*)

Plot to see the static graph at each snapshot

can be row of graphs or an interactive graph with a slider to change snapshot. In all cases, the position of nodes is the same in all snapshots.

The position of nodes is determined using the networkx force directed layout, addition parameters of the function are passed to this functions (e.g., iterations=100, k=2. . .)

Parameters

- **dynamic_graph** – DynGraphSN
- **communities** – dynamic snapshot_affiliations of the network (can be ignored)
- **ts** – time of snapshot(s) to display. single value or list. default None means all snapshots.
- **slider** – If None, a slider allows to interactively choose the step (work only in jupyter notebooks on a local machine)
- **to_datetime** – one of True/False/function. If True, step IDs are converted to dates using datetime.utcfromtimestamp. If a function, should take a step ID and return a datetime object.
- **width** – width of the figure
- **height** – height of the figure

Returns bokeh layout containing slider and plot, or only plot if no slider.

tnetwork.plot_longitudinal

tnetwork.**plot_longitudinal**(*dynamic_graph=None*, *communities=None*, *sn_duration=None*, *to_datetime=False*, *nodes=None*, *width=800*, *height=600*, *bokeh=False*, *auto_show=False*)

A longitudinal view of nodes/snapshot_communities

Plot snapshot_affiliations such as each node corresponds to a horizontal line and time corresponds to the horizontal axis

Parameters

- **dynamic_graph** – DynGraphSN or DynGraphIG
- **communities** – dynamic snapshot_affiliations, DynCommunitiesSN or DynCommunitiesIG
- **sn_duration** – the duration of a snapshot, as int or timedelta. If none, default is the network frequency
- **to_datetime** – one of True/False/function. If True, step IDs are converted to dates using datetime.utcfromtimestamp. If a function, should take a step ID and return a datetime object.
- **nodes** – If none, plot all nodes in lexicographic order. If a list of nodes, plot only those nodes, in that order
- **width** – width of the figure
- **height** – height of the figure

2.4.4 Dynamic Communities Classes

For each representation of dynamic graphs, there is a corresponding representation of dynamic partitions:

- DynGraphSN == DynCommunitiesSN (snapshots)
- DynGraphIG == DynCommunitiesIG (interval graphs)

Dynamic communities are (currently) identified by labels, i.e. each community is associated with a unique label, and two nodes that have the same labels (in the same or in different time steps) belongs to the same (dynamic) community.

Sequences of snapshots representations

class tnetwork.**DynCommunitiesSN**(*snapshots=None*)

Dynamic communities as sequences of snapshots

Communities are represented as a SortedDict, key:time, value: dict id:{set of nodes}

Adding and removing affiliations

DynCommunitiesSN.add_affiliation(nodes, …)	Affiliate node(s) to community(ies) at time(s)
DynCommunitiesSN.add_community(t, nodes[, id])	Add a community at a time
DynCommunitiesSN.set_communities(t[, …])	Affiliate nodes given a dictionary representation

tnetwork.DynCommunitiesSN.add_affiliation

DynCommunitiesSN.**add_affiliation**(*nodes*, *cIDs*, *times*)

Affiliate node(s) to community(ies) at time(s)

Add belonging for the provided node(s) to the provided communitie(s) at the provided time(s). (all nodes, at all times, in all communities) If communities do not exist, they are created.

Parameters

- **nodes** – accept set/list of nodes or single node

- **times** – accept list of times or single time
- **cIDs** – accept lists of coms or single com

Returns

tnetwork.DynCommunitiesSN.add_community

`DynCommunitiesSN.add_community(t, nodes, id=None)`

Add a community at a time

Create a community at time t with the provided nodes and id (random id if not provided)

Parameters

- **t** – time
- **nodes** – a community provided as a set/list of nodes
- **id** – optional id, otherwise, new unique one

tnetwork.DynCommunitiesSN.set_communities

`DynCommunitiesSN.set_communities(t, communities=None)`

Affiliate nodes given a dictionary representation

Given a clustering provided as a dict id:{set of nodes} , set this clustering at the provided time (replace any existing clustering at that time)

Parameters

- **t** – a time instant
- **communities** – communitie as dict id:{set of nodes}

Accessing affiliations

DynCommunitiesSN.affiliations([t])	Affiliations by nodes
DynCommunitiesSN.communities([t])	Communities
DynCommunitiesSN.snapshot_affiliations([t])	Affiliations by snapshots
DynCommunitiesSN.snapshot_communities([t])	Affiliations by communities

tnetwork.DynCommunitiesSN.affiliations

`DynCommunitiesSN.affiliations(t=None)`

Affiliations by nodes

If t is given, return affiliation at this t as a dict, key=node, value=set of communities else, return a dict, key:node, value: dict community:list of times

Parameters **t** – time

Returns dictionary, key=node, value=dict community:list of times or if t is not None: dict community:list

tnetwork.DynCommunitiesSN.communities

DynCommunitiesSN.**communities**(*t=None*)

Communities

If t is given, return communities at this t as a dict, key=node, value=set of communities else, return a dict, key:node, value: dict community:list of times

Parameters **t** – time

Returns dictionary, key=node, value=dict community:list of times or if t is not None: dict community:list

tnetwork.DynCommunitiesSN.snapshot_affiliations

DynCommunitiesSN.**snapshot_affiliations**(*t=None*)

Affiliations by snapshots

If t is given, return affiliation at this t as a dict, key=node, value=set of communities else, return a sorted dict, key:time, value: dict node:communities

Parameters **t** – time

Returns sorted dict, key:time, value: dict node:communities or key=node, value=set of communities

tnetwork.DynCommunitiesSN.snapshot_communities

DynCommunitiesSN.**snapshot_communities**(*t=None*)

Affiliations by communities

If t is given, return communities at this t as a bidict id:{set of nodes} else, return a sorted dict, key:time, value: dict id:{set of nodes}

Parameters **t** – time

Returns a dict id:{set of nodes}

Statistics/Other

DynCommunitiesSN.communities_duration()	Duration of each community
DynCommunitiesSN.affiliations_durations([…])	Duration of affiliations
DynCommunitiesSN.snapshots_timesteps()	Return the list of time steps
DynCommunitiesSN.automatic_node_order()	Return an order of nodes optimized for longitudinal plotting

tnetwork.DynCommunitiesSN.communities_duration

DynCommunitiesSN.**communities_duration**()

Duration of each community

Returns {id:duration}

tnetwork.DynCommunitiesSN.affiliations_durations

`DynCommunitiesSN.`**`affiliations_durations`**(*nodes=None*, *communities=None*)

Duration of affiliations

Return the duration in each community (for non-zero values) for the provided nodes and the provided communities (default: all) return set of triplets (n,c,duration), or set of pairs of one if the parameters has a single value, or a single value if single node and single com

Parameters

- **`nodes`** – node(s) for which we want durations. single node or set of nodes
- **`communities`** – communities(s) for which we want durations. single community or set of communities

Returns set of triplets (n,c,duration), or set of pairs of one if the parameters has a single value, or a single value if single node and single com

tnetwork.DynCommunitiesSN.snapshots_timesteps

`DynCommunitiesSN.`**`snapshots_timesteps`**()

Return the list of time steps

Returns list of time steps

tnetwork.DynCommunitiesSN.automatic_node_order

`DynCommunitiesSN.`**`automatic_node_order`**()

Return an order of nodes optimized for longitudinal plotting

Note: code is not optimized, could be improved! :return: list of nodes names

Converting

DynCommunitiesSN.to_DynCommunitiesIG(sn_duration)	Convert to SG communities

tnetwork.DynCommunitiesSN.to_DynCommunitiesIG

`DynCommunitiesSN.`**`to_DynCommunitiesIG`**(*sn_duration*, *convertTimeToInteger=False*)

Convert to SG communities

Parameters

- **`sn_duration`** – time of a snapshot, or None for automatic: each snapshot last until start of the next
- **`convertTimeToInteger`** – if True, communities IDs will be forgottent and replaced by consecutive integers

Returns DynamicCommunitiesIG

Interval graph representations

class tnetwork.**DynCommunitiesIG**(*start=None*, *end=None*)

Dynamic communities as interval graphs

This class maintains a redondant representation for faster access:

- _by_node: for each node, for each community, Interval of affectation (affectations)
- _by_com: for each com, for each node, Interval of affectation (communities)

Note that they are hidden for this reason, if you modify one, you need to be careful maintaining the other one. You can however access them without problem directly, or use the corresponding functions (affiliation and communities)

Adding and removing snapshot_affiliations

DynCommunitiesIG.add_affiliation(nodes, ...)	Affiliate node n to community com for period times
DynCommunitiesIG.add_affiliations_from(...)	Add communities provided as a cluster
DynCommunitiesIG.remove_affiliation(n, com, ...)	Remove affiliations

tnetwork.DynCommunitiesIG.add_affiliation

DynCommunitiesIG.**add_affiliation**(*nodes*, *cIDs*, *times*)

Affiliate node n to community com for period times

Parameters

- **nodes** – node or list/set of nodes
- **cIDs** – community or list/set of communities. str
- **times** – period as an Interval object, or a pair (start,end) or list of pairs

tnetwork.DynCommunitiesIG.add_affiliations_from

DynCommunitiesIG.**add_affiliations_from**(*communities*, *times*)

Add communities provided as a cluster

Given a community provided as a dict id:{set of nodes} , add it for the period times (intervals)

Parameters

- **communities** – dict id:{set of nodes}
- **times** – an Intervals object or a single period as a pair (start, end)

tnetwork.DynCommunitiesIG.remove_affiliation

DynCommunitiesIG.**remove_affiliation**(*n: str*, *com*, *times: tnetwork.utils.intervals.Intervals*)

Remove affiliations

remove affiliations of node n from community com between the period times

Parameters

- **n** – node
- **com** – community
- **times** – Intervals

Accessing snapshot_affiliations

DynCommunitiesIG.affiliations([t])	Affiliations by nodes
DynCommunitiesIG.communities([t])	Affiliations by communities
DynCommunitiesIG.affiliations_durations([…])	Durations of affiliations

tnetwork.DynCommunitiesIG.affiliations

DynCommunitiesIG.**affiliations**(*t=None*)

Affiliations by nodes

Parameters **t** – time of the affiliations ro return. Default: all

Returns either a dictionary (by node) of dictionaries (by community) of Intervals if t==None or a dictionary (by node) of list of snapshot_communities

tnetwork.DynCommunitiesIG.communities

DynCommunitiesIG.**communities**(*t=None*)

Affiliations by communities

Parameters **t** – time of the community ro return. Default: all

Returns either a dictionary (by community) of dictionaries (by node) of Intervals if t==None or a dictionary (by community) of Intervals

tnetwork.DynCommunitiesIG.affiliations_durations

DynCommunitiesIG.**affiliations_durations**(*nodes=None*, *communities=None*)

Durations of affiliations

Return the duration in each community (for non-zero values) for the provided nodes and the provided communities (default: all) return set of triplets (n,c,duration), or set of pairs of one if the parameters has a single value, or a single value if single node and single com

Parameters

- **nodes** – node(s) for which we want durations. single node or set of nodes
- **communities** – communities(s) for which we want durations. single community or set of communities

Returns set of triplets (n,c,duration), or set of pairs of one if the parameters has a single value, or a single value if single node and single com

Other functions

DynCommunitiesIG.nodes_main_com()	Main community for each node
DynCommunitiesIG.nodes_natural_order()	Nodes by lexicographic order
DynCommunitiesIG.nodes_ordered_by_com([node2com])	Nodes ordered by their main community

tnetwork.DynCommunitiesIG.nodes_main_com

DynCommunitiesIG.**nodes_main_com**()

Main community for each node

Function that return for each node the community in which it spends the most time

Returns dictionary, {node:community)

tnetwork.DynCommunitiesIG.nodes_natural_order

DynCommunitiesIG.**nodes_natural_order**()

Nodes by lexicographic order

Returns list od nodes

tnetwork.DynCommunitiesIG.nodes_ordered_by_com

DynCommunitiesIG.**nodes_ordered_by_com**(*node2com=None*)

Nodes ordered by their main community

Return nodes such as those with the same main community are close to each other. By default, use the main community according to internal function nodes_main_com Another order can be passed in parameter.

Parameters **node2Com** – a dictionary associating a node to its main affiliation

Returns list of nodes

2.4.5 Dynamic Community Detection

A simple demo of usage can be found here.

Dynamic community detection is the problem of discovering snapshot_communities in dynamic networks.

There are two types of methods implemented: those that are written in pure python and those who require an external tool.

Those in pure python are part of the *tnetwork.DCD* module while others are in *tnetwork.DCD.external*.

Below is a list of implemented methods, with the type of dynamic networks they are designed to manage. Note that this type of network is unrelated with the tnetwork representation: a snapshot representation can be used to encode a snapshot graph, a link stream or an interval graph. The possible types of dynamic networks are:

- snapshot: The graph is well defined at any t, changes tend to occur synchronously
- interval gaph: The graph is well defined at any t, but graph changes are not synchrone, changes appear edge by edge

- link stream: graphs at any time t are poorly defined, graphs can be studied only by studying a $Delta$ period of aggregation

Table 31: Types of dynamic networks expected by each method

Method	Type of dynamic network
iterative_match	snapshots
smoothed_graph	snapshots
label_smoothing	snapshots
smoothed_louvain	snapshots
rollingCPM	snapshots
MSSCD	link stream
muchaOriginal	snapshots
dynamo	interval graph

Some external algorithms require matlab, and the matlab-python engine, ensuring the connection between both. How to explain it is explained on the matlab website, currenty there: https://fr.mathworks.com/help/matlab/matlab_external/install-the-matlab-engine-for-python.html

Internal algorithms

These algorithms are implemented in python.

`iterative_match`(dynNetSN[, CDalgo, . . .])	Community Detection by iterative detection and matching
`label_smoothing`(dynNetSN[, CDalgo, . . .])	Community detection by label smoothing
`smoothed_louvain`(dynNetSN[, match_function, . . .])	Community Detection using smoothed louvain
`rollingCPM`(dynNetSN[, k, elapsed_time])	This method is based on Palla et al[1].
`smoothed_graph`(dynNetSN[, alpha, . . .])	Smoothed graph approach
`MSSCD`(dyn_graph[, t_granularity, . . .])	Multi Scale Stable Community Detection

tnetwork.DCD.iterative_match

`tnetwork.DCD.`**`iterative_match`**(*dynNetSN*, *CDalgo='louvain'*, *match_function=<function jaccard>*, *threshold=0.3*, *elapsed_time=False*, *multithread=False*)

Community Detection by iterative detection and matching

This algorithm is inspired by the one proposed by Greene et al., [1] but additionally to the detection of match between communities in consecutive snapshots, a post process assign labels to communities, based on the following rules:

- A community "send" its label to the community the most similar in the next snapshot
- If a community "receives" several labels from communities in the previous snapshot, it selects the one of the community the most similar.

[1]Greene, Derek, Donal Doyle, and Padraig Cunningham. "Tracking the evolution of snapshot_communities in dynamic social networks." 2010 international conference on advances in social networks analysis and mining. IEEE, 2010.

Parameters

- **dynNetSN** – a dynamic network

- **CDalgo** – community detection to apply at each step. Can be a function returning a clustering, or the string "louvain" or "smoothedLouvain
- **match_function** – a function that gives a matching score between two communities (two sets of nodes). Default: jaccard. If None, no matching is done
- **threshold** – a threshold for match_function below which snapshot_communities are not matched
- **multithread** – If true, run in parallel. Some bugs in macOs/windows.

tnetwork.DCD.label_smoothing

tnetwork.DCD.**label_smoothing**(*dynNetSN*, *CDalgo='louvain'*, *match_function=<function jaccard>*, *threshold=0.3*, *multithread=False*, ***kwargs*)

Community detection by label smoothing

This method is based on falkowsky et al.[1]. It first detect communities in each snapshot, then try to match any community with any other one in any other snapshot, constituting a survival graph. A community detection algorithm is then applied on this survival graph, yielding dynamic snapshot_communities.

[1]Falkowski, T., Bartelheimer, J., & Spiliopoulou, M. (2006, December). Mining and visualizing the evolution of subgroups in social networks. In Proceedings of the 2006 IEEE/WIC/ACM International Conference on Web Intelligence (pp. 52-58). IEEE Computer Society.

Parameters

- **dynNetSN** – a dynamic network
- **CDalgo** – community detection to apply at each step. Can be a function returning a clustering, or the string "louvain" or "smoothedLouvain"
- **match_function** – a function that gives a matching score between two snapshot_communities (two sets of nodes). Default: jaccard
- **threshold** – a threshold for match_function below which snapshot_communities are not matched

Returns DynCommunitiesSN

tnetwork.DCD.smoothed_louvain

tnetwork.DCD.**smoothed_louvain**(*dynNetSN*, *match_function=<function jaccard>*, *threshold=0.3*, ***kwargs*)

Community Detection using smoothed louvain

This algorithm is a naive implementation of the method proposed by [1]. The idea is that for each snapshots, the louvain algorithm is ran, but instead of being initialized with each node in its own community as usual, the partition obtained in the previous partition is used.

The label attribution process is the same described in the paper XXX, see method simple_matching for details.

Internally, it calls the simple_matching method, the same parameters can be passed to it.

[1]Aynaud, T., & Guillaume, J. L. (2010, May). Static community detection algorithms for evolving networks. In 8th International symposium on modeling and optimization in mobile, Ad Hoc, and wireless networks (pp. 513-519). IEEE.

Parameters **dynNetSN** – a dynamic network

Returns DynCommunitiesSN

tnetwork.DCD.rollingCPM

`tnetwork.DCD.rollingCPM`(*dynNetSN: tnetwork.dyn_graph.dyn_graph_sn.DynGraphSN*, *k=3*, *elapsed_time=False*)

This method is based on Palla et al[1]. It first computes overlapping snapshot_communities in each snapshot based on the clique percolation algorithm, and then match snapshot_communities in successive steps using a method based on the union graph.

[1] Palla, G., Barabási, A. L., & Vicsek, T. (2007). Quantifying social group evolution. Nature, 446(7136), 664.

Parameters

- **dynNetSN** – a dynamic network (DynGraphSN)
- **k** – the size of cliques used as snapshot_communities building blocks
- **elapsed_time** – if True, will return a tuple (communities,time_elapsed)

Returns DynCommunitiesSN

tnetwork.DCD.smoothed_graph

`tnetwork.DCD.smoothed_graph`(*dynNetSN*, *alpha=0.9*, *match_function=<function jaccard>*, *threshold=0.3*, ***kwargs*)

Smoothed graph approach

This approach is a naive implementation of the idea proposed in [1]. To sum up, at each snapshot, a new graph is create which is the combination of the graph at this step and a graph in which edges are present between any two nodes belonging to the same community in the previous step. Note than in the original paper, a method is proposed to greatly reduce the complexity of the solution, but this method is not implemented here.

Alpha is a parameter to tune how important is the weight of the current topology compared with previous partition.

The label attribution process is the same described in the paper XXX, see method simple_matching for details.

Internally, it calls the simple_matching method, the same parameters can be passed to it.

[1]Guo, C., Wang, J., & Zhang, Z. (2014). Evolutionary community structure discovery in dynamic weighted networks. Physica A: Statistical Mechanics and its Applications, 413, 565-576.

Parameters

- **dynNetSN** –
- **alpha** – parameter setting relative importance of past VS current graph. 1: only current, 0: only previous

Returns

tnetwork.DCD.MSSCD

`tnetwork.DCD.MSSCD`(*dyn_graph*, *t_granularity=1*, *t_persistance=3*, *t_quality=0.7*, *t_similarity=0.3*, *similarity=<function jaccard>*, *CD='louvain'*, *QC=<function score_conductance>*, *weighted_aggregation=True*, *Granularity=None*, *start_time=None*, *elapsed_time=False*, *as_dyn_com=True*)

Multi Scale Stable Community Detection

Method described in [1]. This method allows to find stable communities accross multiple temporal scales. In summary, it creates new snapshots by aggregating the existing ones. At each granularity level, it discover stabel communities by

- 1) applying a community detection algorithm at each step
- 2) keeping communities with the highest quality score as seeds
- 3) Expand those seeds to neighbor snashots as long as they remain relevant accordin to the quality score
- 4) keep as stable only communities that are present in several successive snapshots

[1] Boudebza, S., Cazabet, R., Nouali, O., & Azouaou, F. (2019). Detecting Stable Communities in Link Streams at Multiple Temporal Scales. LEG workshop, @ECML-PKDD 2019

Parameters

- **dyn_graph** – a dynamic graph
- **t_granularity** – (θ_γ min temporal granularity,scale to analyze
- **t_persistance** – θ_p minimum number of successive occurences for the community to be persistant
- **t_quality** – θ_q threashold of community quality
- **t_similarity** – θ_s threashold of similarity between communities
- **similarity** – (CSS)function that give a score of similarity between communities. Default: jaccard
- **CD** – CD community detection algorithm. A function returning a set of set of nodes. By default, louvain algorithm
- **QC** – (QC)function to determine the quality of communities. Default: inverse of conductance
- **weighted_aggregation** – if true, the aggregation over time periods is done using weighted networks
- **Granularity** – (Γ) can be used to replace the default scales. List of int.
- **start_time** – the date at which to start the analysis. Can be useful, for instance, to start analysis at 00:00
- **as_dyn_com** – if true, return a dynamic community object. If False, a custom format with quadruplets (nodes, duration, granularity, quality)

Returns a dynamic community object (default) or a list of quadruplets, see parameter as_dyn_com

External algorithms

These algorithms call external code provided by authors, and thus might require installing additional softwares (java, matlab).

dynamo(dyn_graph[, elapsed_time, timeout])	DynaMo algorithm
transversal_network_mucha_original(dyn_graph)	Multiplex community detection, Mucha et al.
transversal_network_leidenalg(dyn_graph[, ...])	Multiplex community detection reimplemented in leidenalg
estrangement_confinement(dyn_graph[, ...])	Estrangement confinement

tnetwork.DCD.externals.dynamo

`tnetwork.DCD.externals.dynamo`(*dyn_graph: tnetwork.dyn_graph.dyn_graph_sn.DynGraphSN*, *elapsed_time=False*, *timeout=10*)

DynaMo algorithm

Requires JAVA Algorithm introduced in [1]. In summary, maintain a high modularity solution through local updates of community structure

[1]Zhuang, D., Chang, M. J., & Li, M. (2019). DynaMo: Dynamic Community Detection by Incrementally Maximizing Modularity. IEEE Transactions on Knowledge and Data Engineering.

Parameters

- **dyn_graph** –
- **elapsed_time** –
- **timeout** –

Returns

tnetwork.DCD.externals.transversal_network_mucha_original

`tnetwork.DCD.externals.transversal_network_mucha_original`(*dyn_graph: tnetwork.dyn_graph.dyn_graph_sn.DynGraphSN*, *om=0.5*, *form='local'*, *elapsed_time=False*, *matlab_session=None*)

Multiplex community detection, Mucha et al.

Algorithm described in [1]

Brief summary: a single network is created by adding nodes between themselves in different snaphsots. A modified modularity optimization algorithm is run on this network

For this function, it is necessary to have Matlab installed And to set up the matlab for python engine, see how to there https://fr.mathworks.com/help/matlab/matlab_external/install-the-matlab-engine-for-python.html (you can find the value of matlabroot by tapping matlabroot in your matlab console)

If you do not have matlab, you can try to use the transversal_network_leidenalg which is slower but requires only a package installation

[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), 876-878.

Parameters

- **dyn_graph** – dynamic network
- **om** –
- **form** –
- **elapsed_time** –
- **matlab_session** –

Returns

tnetwork.DCD.externals.transversal_network_leidenalg

`tnetwork.DCD.externals.transversal_network_leidenalg`(*dyn_graph: tnetwork.dyn_graph.dyn_graph_sn.DynGraphSN*, *interslice_weight=1*, *elapsed_time=False*)

Multiplex community detection reimplemented in leidenalg

Algorithm described in [1] (see method *mucha_original* for more information) This function use the implementation in the leidenalg library instead of the original matlab implementation. It requires the installation of the leidenalg library (including igraph). It is usually slower than the original implementation (but does not require matlab)

[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), 876-878.

Parameters

- **dyn_graph** – dynamic network
- **interslice_weight** –
- **elapsed_time** –

Returns

tnetwork.DCD.externals.estrangement_confinement

`tnetwork.DCD.externals.estrangement_confinement`(*dyn_graph: tnetwork.dyn_graph.dyn_graph_sn.DynGraphSN*, *tolerance=1e-05*, *convergence_tolerance=0.01*, *delta=0.05*, *elapsed_time=False*, ***kwargs*)

Estrangement confinement

Algorithm introduced in [1]. Uses original code.

[1]Kawadia, V., & Sreenivasan, S. (2012). Sequential detection of temporal communities by estrangement confinement. Scientific reports, 2, 794.

Parameters

- **delta** – see original article
- **convergence_tolerance** – see original article
- **tolerance** – see original article

Returns

2.4.6 Benchmark Generator

A simple demo of usage can be found here.

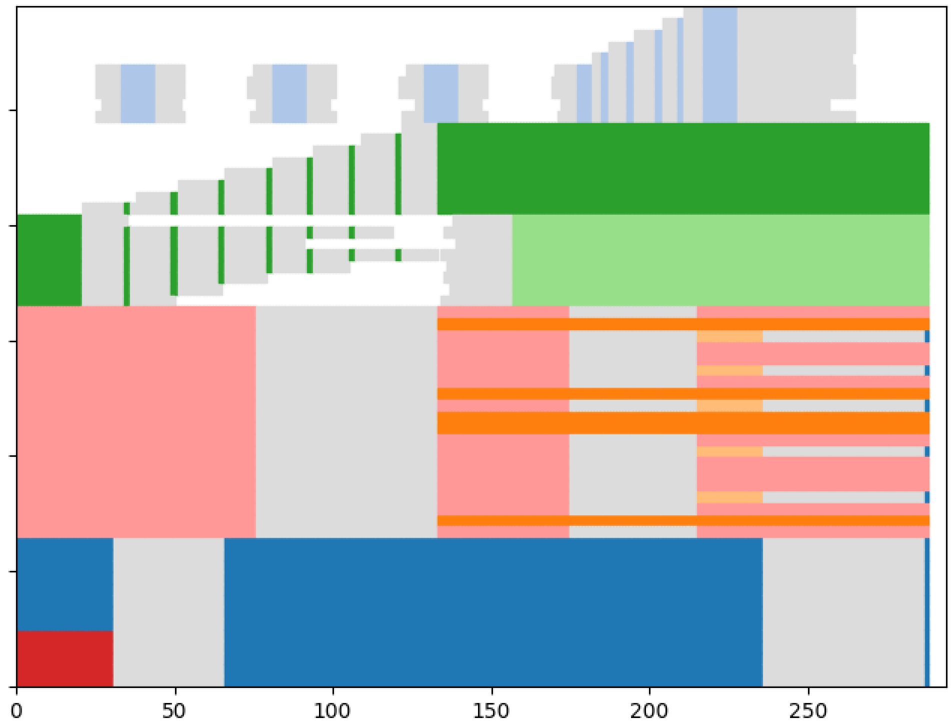
The library implements several benchmark generators. The aim of those benchmark is to generate both a temporal graph and a *reference* dynamic community structure.

Currently, two benchmarks are implemented:

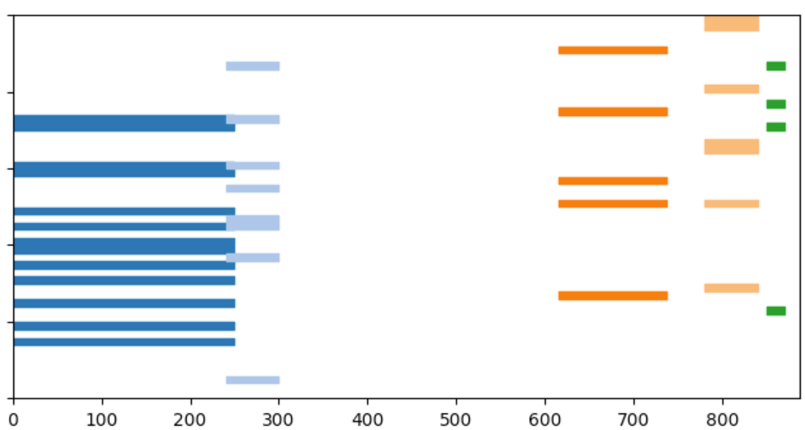
- Benchmark with custom event scenario

- Benchmark with stable, multiple temporal scale communities

Example of custom scenario



Example of stable communities



Benchmark with custom communities

class tnetwork.**ComScenario**(*alpha=0.8*, *external_density_penalty=0.05*, *random_noise=0*, *verbose=False*, *variant='deterministic'*)

This class manages the community evolution scenario

It implements the benchmark described in XXX

Behavior to keep in mind:

1) Any node that does not belong to a community is condered "dead". Note that it can reappear later if it belongs to a community again. As a consequence, a node alive but not belonging to any community must be represented as a node belonging to a community of size 1

2)There are not really persistent community, every time a community is modified in any way, a new community is created, and it is only because they have the same name (label) that they are considered part of the same dynamic community.

As a consequence, to kill a dynamic community, one simply needs to stop using its label.

ComScenario.__init__([alpha, …])	Initialize the community generation class

tnetwork.ComScenario.__init__

ComScenario.**__init__**(*alpha=0.8*, *external_density_penalty=0.05*, *random_noise=0*, *verbose=False*, *variant='deterministic'*)

Initialize the community generation class

When initializing, we can set the parameters of the link generation

Parameters

- **alpha** – alpha parameter that determines the density of communities decrease with size
- **external_density_penalty** – beta, how smaller the density of outside community is compared to a a community of the same size
- **random_noise** – beta_r, fraction of existing edges that are randomly rewired at each step
- **verbose** – If true, print debugging information
- **variant** – the variant of the generator controls the way edges are generated. Currently, only "deterministic" is fully suported

Function to define events

ComScenario.INITIALIZE(sizes, labels)	Function to initialize the dynamic networks with communities that already exist at the beginning
ComScenario.BIRTH(size, label, **kwargs)	Creates a new community
ComScenario.DEATH(com, **kwargs)	Kill a community
ComScenario.MERGE(toMerge, merged, **kwargs)	Merge the communities in input into a single community with the name (label) provided in output
ComScenario.SPLIT(toSplit, newComs, sizes, …)	Split a single community into several ones.
ComScenario.THESEUS(theComTh[, nbNodes, …])	Create a theseus ship operation.

Continued on next page

Table 35 – continued from previous page

ComScenario.RESURGENCE(theComTh[, death_period])	Create a resurgence operation.
ComScenario.GROW_ITERATIVE(com, nb_nodes2Add)	Make a community grow node by node
ComScenario.SHRINK_ITERATIVE(com, …[, …])	Make a community shrink node by node
ComScenario.MIGRATE_ITERATIVE(comFrom, …)	Make nodes of a community migrate to another one
ComScenario.ASSIGN(comsBefore, comsAfter, …)	Define a custom event
ComScenario.CONTINUE(com, **kwargs)	Keep a community unchanged

tnetwork.ComScenario.INITIALIZE

ComScenario.**INITIALIZE**(*sizes: [<class 'int'>], labels: [<class 'str'>] = None*)
Function to initialize the dynamic networks with communities that already exist at the beginning

Parameters

- **sizes** – list of the communities sizes (same order as names)
- **labels** – list of the communities labels (if None, unique labels are given automatically)

tnetwork.ComScenario.BIRTH

ComScenario.**BIRTH**(*size: int, label: str = None, **kwargs*)
Creates a new community

Parameters

- **size** – number of nodes to create
- **label** – label of the community (default will create a random label)

Returns the community created (community object)

tnetwork.ComScenario.DEATH

ComScenario.**DEATH**(*com: tnetwork.DCD.community.Community, **kwargs*)
Kill a community

Returns empty list

tnetwork.ComScenario.MERGE

ComScenario.**MERGE**(*toMerge: [<class 'tnetwork.DCD.community.Community'>], merged: str, **kwargs*)
Merge the communities in input into a single community with the name (label) provided in output

Parameters

- **toMerge** – labels of snapshot_affiliations to merge
- **merged** – label of the merged community (can be same as one of the input or not

Returns the merged community (community object)

tnetwork.ComScenario.SPLIT

ComScenario.**SPLIT**(*toSplit: tnetwork.DCD.community.Community, newComs: [<class ’str’>], sizes: [<class ’int’>], **kwargs*)

Split a single community into several ones. Note that to control exactly which nodes are moved, one should use migrate instead

Parameters

- **toSplit** – label of the community to split
- **newComs** – labels to give to the new snapshot_affiliations (list). The label of the community before split can be or not among them
- **sizes** – sizes of the new snapshot_affiliations, in number of nodes. In the same order as newComs.

Returns a list of snapshot_affiliations resulting from the split.

tnetwork.ComScenario.THESEUS

ComScenario.**THESEUS**(*theComTh: tnetwork.DCD.community.Community, nbNodes=None, wait_step=1, delay=1, **kwargs*)

Create a theseus ship operation.

Parameters

- **theComTh** – the community to modify
- **nbNodes** – the number of nodes to be replaced
- **delay** – the waiting time before the first change
- **wait_step** – the waiting time between each node replacement

Returns a tuple of snapshot_affiliations, current ship, new ship

tnetwork.ComScenario.RESURGENCE

ComScenario.**RESURGENCE**(*theComTh: tnetwork.DCD.community.Community, death_period=20, **kwargs*)

Create a resurgence operation.

Parameters

- **theComTh** – the community to modify
- **death_period** – time to remain dead

Returns a tuple of snapshot_affiliations, current ship, new ship

tnetwork.ComScenario.GROW_ITERATIVE

ComScenario.**GROW_ITERATIVE**(*com, nb_nodes2Add, wait_step=1, delay=1, **kwargs*)

Make a community grow node by node

The community com add nodes2add nodes one by one, with an interval delay between each :param com: community to grow :param nodes2Add: nb nodes to add :param delay: the waiting time before the first change :param wait_step: the waiting time between each node addition :return:

tnetwork.ComScenario.SHRINK_ITERATIVE

ComScenario.**SHRINK_ITERATIVE**(*com*, *nb_nodes2remove*, *wait_step=1*, *delay=1*, ***kwargs*)

Make a community shrink node by node

The community com lose nodes2add nodes one by one, with an interval delay between each :param com: community to shrink :param nodes2remove: nb nodes to remove :param delay: the waiting time before the first change :param wait_step: the waiting time between each node removal :return:

tnetwork.ComScenario.MIGRATE_ITERATIVE

ComScenario.**MIGRATE_ITERATIVE**(*comFrom*, *comTo*, *nbNodes*, *wait_step=1*, *delay=1*, ***kwargs*)

Make nodes of a community migrate to another one

The community comFrom lose nodes2add nodes one by one, that join the community comTo, with an interval delay between each migration

Parameters

- **comFrom** – community to shrink
- **comTo** – community to grow
- **nbNodes** – nb nodes to move
- **delay** – the waiting time before the first change
- **wait_step** – the waiting time between each node change

Returns

tnetwork.ComScenario.ASSIGN

ComScenario.**ASSIGN**(*comsBefore: [<class 'tnetwork.DCD.community.Community'>], comsAfter: [<class 'str'>], splittingOut: [{<class 'str'>}], **kwargs*)

Define a custom event

Migrate nodes from a set of snapshot_affiliations to another set of snapshot_affiliations. Can be used to move a set of nodes from a community to another or any other more complex scenario.

Parameters

- **comBefore** – Ccommunities in input
- **comsAfter** – label(s) to give to the resulting communities
- **splittingOut** – How to distribute nodes in output. It is a list of same lenght than comsAfter, and each element of the list is a set of names of nodes. Note that if some nodes present in input does not appear in output, they are considered "killed"

Returns the communities resulting from the operation (list)

tnetwork.ComScenario.CONTINUE

ComScenario.**CONTINUE**(*com*, ***kwargs*)

Keep a community unchanged

By using parameters delay and/or triggers, CONTINUE makes the community com_before to stay unchanged for some time.

Parameters **com** – the community to keep unchanged

Returns the same community

Run

ComScenario.run()	Function to call when the scenario has been defined to actually execute it.

tnetwork.ComScenario.run

ComScenario.**run**()

Function to call when the scenario has been defined to actually execute it. Return a dynamic network and the corresponding dynamic partition

Returns a couple, first element is the dynamic network, second element is the dynamic partition

Toy example

This is the generator of toy examples used in the original paper.

generate_toy_random_network(**kwargs)	Generate a small, toy dynamic graph
generate_simple_random_graph([nb_com, …])	Generate a simple random dynamic graph with community structure

tnetwork.generate_toy_random_network

tnetwork.**generate_toy_random_network**(***kwargs*)

Generate a small, toy dynamic graph

Generate a toy dynamic graph with evolving communities, following scenario described in XXX Optional parameters are the same as those passed to the ComScenario class to generate custom scenarios

Returns pair, (dynamic graph, dynamic reference partition) (as snapshots)

tnetwork.generate_simple_random_graph

tnetwork.**generate_simple_random_graph**(*nb_com=10, min_size=5, max_size=15, operations=20, mu=0, mu_noise=0.01*)

Generate a simple random dynamic graph with community structure

This is the generator described in XXX. It generates a graph with dynamic community structure which is a combination of successive merge and splits.

Parameters

- **nb_com** – number of initial communities
- **min_size** – size below which communities cannot be split
- **max_size** – size above which community split
- **operations** – number of operations (merge/split) to execute (involves random communities)

- **mu** – parameter to set how well defined is the community structure (0=>perfect community structure) more precisely, it defines: alpha=1-mu, beta=mu
- **mu_noise** – set the mu_r, i.e., fraction of edges randomly rewired at each snapshot

Returns pair (graph, communities)

Community class

class tnetwork.DCD.community.**Community**(*comScenario*, *label=None*)

Class representing communities in a benchmark scenario

When generating a benchmark using the scenerio generator, communities returned by event definition functions are instances of this class.

This class has some public functions to check the names, the nodes, and the number of edges of the community. The edges themselves cannot be checked during the scenario description, since they are generated when calling the run function of the ComScenario class.

Community.label()	Get the name (label) of this structure :return: name :rtype: str
Community.nodes()	Get the nodes of this structure :return: list of nodes :rtype: [str]
Community.nb_intern_edges()	return the number of edges expected in this community :return:

tnetwork.DCD.community.Community.label

Community.**label**()

Get the name (label) of this structure :return: name :rtype: str

tnetwork.DCD.community.Community.nodes

Community.**nodes**()

Get the nodes of this structure :return: list of nodes :rtype: [str]

tnetwork.DCD.community.Community.nb_intern_edges

Community.**nb_intern_edges**()

return the number of edges expected in this community :return:

Benchmark with stable, multiple temporal scales communities

generate_multi_temporal_scale([nb_steps, ...])	Generate dynamic graph with stable communities

tnetwork.DCD.multi_temporal_scale.generate_multi_temporal_scale

`tnetwork.DCD.multi_temporal_scale.generate_multi_temporal_scale`(*nb_steps=5000*, *nb_nodes=100*, *nb_com=10*, *noise=None*, *max_com_size=None*, *max_com_duration=None*)

Generate dynamic graph with stable communities

This benchmark allows to generate temporal networks as described in *Detecting Stable Communities in Link Streams at Multiple Temporal Scales. Boudebza, S., Cazabet, R., Nouali, O., & Azouaou, F. (2019).*.

To sum up the method, *stable* communities are generated (i.e., no node change). These communities exist for some periods, but have different *temporal scales*, i.e., some of them have a high frequency of edges (their edges appear at every step) while others have a lower frequency (i.e., each edge appear only every t steps). To simplify, communities are complete cliques.(but for the low frequency ones, we might observe only a small fraction of their edges in every step)

The basic parameters are the number of steps, number of nodes and number of communities. There are other parameters allowing to modify the random noise, the maximal size of communities and the maximal duration of communities, that are by default assigned with values scaled according to the other parameters.

Parameters

- **nb_steps** – steps in the graph
- **nb_nodes** – total nb nodes
- **nb_com** – nb desired communities
- **noise** – random noise at each step, i.e. probability for any edge to exist at any step. default,1/(nb_nodes**2)
- **max_com_size** – max number of nodes. Default: nb_nodes/4
- **max_com_duration** – max community duration. Default: nb_steps/2

Returns

2.4.7 Evaluation of Dynamic Communities

This section contains functions useful to evaluate the quality of dynamic communities.

They were introduced in XXX.

They can be split in 3 categories:

- Evaluation of an average value at each step (*similarity_at_each_step*,'quality_at_each_step')
- Evaluation of smoothness (*SM_L*,'SM_N','SM_P')
- Longitudinal evaluation (*longitudinal_similarity*)

A benchmark is also proposed that can be used to reproduce the results presented in the paper XXX.

Main evaluation functions

similarity_at_each_step(. . . [, score])	Compute similarity at each step

Continued on next page

Table 40 – continued from previous page

quality_at_each_step(dynamicCommunities, . . .)	Compute a community quality at each step
SM_L(dyn_com[, sn_duration])	Smoothness for labels
SM_N(dyn_com)	Smoothness for nodes
SM_P(dyn_com)	Smoothness for partitions
longitudinal_similarity(. . . [, score, . . .])	Longitudinal similarity

tnetwork.DCD.analytics.dynamic_partition.similarity_at_each_step

tnetwork.DCD.analytics.dynamic_partition.**similarity_at_each_step**(*dynamicCommunityReference: tnetwork.dyn_community.communities_dy dynamicCommunityObserved: tnetwork.dyn_community.communities_dy score=None*)

Compute similarity at each step

It takes into account the fact that the reference might by incomplete. (remove from the observations all nodes/time not present in the reference)

Parameters

- **dynamicCommunityReference** – the dynamic partition to use as reference
- **dynamicCommunityObserved** – the dynamic partition to evaluate
- **score** – score to use, default adjusted NMI

Returns pair (list of scores, list of sizes)

tnetwork.DCD.analytics.dynamic_partition.quality_at_each_step

tnetwork.DCD.analytics.dynamic_partition.**quality_at_each_step**(*dynamicCommunities: tnetwork.dyn_community.communities_dyn_sn dynamicGraph: tnetwork.dyn_graph.dyn_graph_sn.DynGraph score=None*)

Compute a community quality at each step

Parameters

- **dynamicCommunities** – dynamic communities as SN
- **score** – score to use, default: Modularity

Returns pair(scores, sizes)

tnetwork.DCD.analytics.dynamic_partition.SM_L

tnetwork.DCD.analytics.dynamic_partition.**SM_L**(*dyn_com, sn_duration=1*)

Smoothness for labels

Inverse of the entropy by node :param dyn_com: dyanamic partition :param sn_duration: used to indicate the duration of snapshots if provided graph is a snapshot graph :return: SM-L score

tnetwork.DCD.analytics.dynamic_partition.SM_N

`tnetwork.DCD.analytics.dynamic_partition.`**`SM_N`**(*dyn_com*)

Smoothness for nodes

Inverse of the number of node changes :param dyn_com: dynamic partition :return: SM-N score

tnetwork.DCD.analytics.dynamic_partition.SM_P

`tnetwork.DCD.analytics.dynamic_partition.`**`SM_P`**(*dyn_com*)

Smoothness for partitions

Averge of the NMI between successive snapshots :param dyn_com: dynamic partition :return: SM-P score

tnetwork.DCD.analytics.dynamic_partition.longitudinal_similarity

`tnetwork.DCD.analytics.dynamic_partition.`**`longitudinal_similarity`**(*dynamicCommunityReference: tnetwork.dyn_community.communities_dy dynamicCommunityObserved: tnetwork.dyn_community.communities_dy score=None, convert_coms_sklearn_format=True*)

Longitudinal similarity

The longitudinal similarity between two dynamic clusters is computed by considering each couple (node,time) as an element belong to a cluster, a cluster containing therefore nodes in differnt times It takes into account the fact that the reference might by incomplete by removing from the partition to evaluate all (node,time) not present in the reference.

Parameters

- **`dynamicCommunityReference`** – the dynamic partition used as reference (ground truth)
- **`dynamicCommunityObserved`** – the dynamic partition to evaluate (result of an algorithm)
- **`score`** – community comparison score, by default the adjsted NMI. (sklearn)
- **`convert_coms_sklearn_format`** – if the score expect in input clusters represented as in sklearn, True. if False, score will receive in input lists of sets of nodes

Returns score

Helper functions that could be used to evaluate smoothness

nb_node_change(dyn_com)	Compute the total number of node changes
entropy_by_node(dyn_com[, sn_duration, …])	Compute the entropy by node.
consecutive_sn_similarity(dynamicCommunity)	Similarity between partitions in consecutive snapshots.

tnetwork.DCD.analytics.dynamic_partition.nb_node_change

`tnetwork.DCD.analytics.dynamic_partition.`**`nb_node_change`**(*dyn_com: tnetwork.dyn_community.communities_dyn_sn.DynCom*

Compute the total number of node changes

Measure of smoothness at the level of nodes, adapated to evaluate glitches

Parameters **`dyn_com`** – The dynamic community

Returns total number of node changes

tnetwork.DCD.analytics.dynamic_partition.entropy_by_node

`tnetwork.DCD.analytics.dynamic_partition.`**`entropy_by_node`**(*dyn_com*, *sn_duration=1*, *fast_on_sn=False*)

Compute the entropy by node.

For each node, compute the shannon entropy of its labels. (always same label=min entropy, every step a new label=max entropy) return the average value for all nodes

Parameters

- **`dyn_com`** – dynamic community to evaluate, can be SN or IG
- **`sn_duration`** – if graph is SN, used to discretize

Returns

tnetwork.DCD.analytics.dynamic_partition.consecutive_sn_similarity

`tnetwork.DCD.analytics.dynamic_partition.`**`consecutive_sn_similarity`**(*dynamicCommunity: tnetwork.dyn_community.communities_*
score=None)

Similarity between partitions in consecutive snapshots.

Compute the average of a similarity score between all pair of successive partitions

Parameters

- **`dynamicCommunity`** – the dynamic partition to evaluate
- **`score`** – the score to use for computing the similarity between each pair of snapshots. default: Overlapping NMI

Returns pair (list of scores, list of partition sizes (avg both partitions))

Benchmark

DCD_benchmark(methods_to_test, mus[, ...])	Compute stats and running time for methods

tnetwork.DCD.benchmarking.DCD_benchmark

`tnetwork.DCD.benchmarking.`**`DCD_benchmark`**(*methods_to_test, mus, nb_coms=[10], subsets=None, iterations=2, min_size=5, max_size=15, operations=20, only_time_statistics=False*)

Compute stats and running time for methods

Function to reproduce benchmarks in XXX. Given methods and some parameters, run algorithms, compute stats, and return the results.

Due to some occasional crashes with some methods, it is safer to call the method several times with subsets of parameters and combine the results later.

For scalability tests, don't forget to set only_time_statistics=True

Parameters

- **methods_to_test** – dictionary {method_name,method}
- **mus** – list of mu values (float)
- **nb_coms** – list of number of communities
- **subsets** – list of subset sizes to test
- **iterations** – number of iteration for each combination of parameters
- **min_size** – min size of communities
- **max_size** – max size of communities
- **operations** – number of events in the random graph
- **only_time_statistics** – if True, do not compute statistics such as average modularity, smoothness etc., which are very time consuming.

Returns communities as a dictionary {ID:{ID:{"}

2.4.8 Intervals Class

class `tnetwork.utils.`**`Intervals`**(*initial=None*)

Class used to represent complex intervals

This class is used to represent periods of existence of nodes and edges. Nodes and edges can exist during not continuous periods (e.g., from time 2 to 5, and from time 7 to 8). Those intervals are represent as closed on the left and open on the right, i.e., [2,5[and [2,8[. If we were to use closed intervals on the right, we would be confronted to ponctual overlaps (without duration), which cause troubles. Furthermore, intervals are often used to represent discrete time events. If we want to express that an edge exist during one hour, from 8a.m. to 9a.m, representing it as [8,9[gives the following results:

- Does the edge exist at 8a.m? -> answer YES
- Does the edge exist at 9a.m? -> answer NO
- Duration -> 1h

When intervals are added, overlapping ones are merged, i.e. if the current Intervals contains [0,3[and [4,5[and we add the interval [2,4[, The resulting Interval will be [0,5[

This class uses a sorted dictionary to maintain efficiently a proper complex interval, key=start date, value=pair(start,end)

The attribute "interv" contains the interval (a SortedDict) and can be safely manipulated

Adding and removing intervals

Intervals.__init__([initial])	Instantiate intervals
Intervals.add_interval(interval)	Add the provided interval to the current interval object.
Intervals.__add__(o)	Add two Intervals using + operator
Intervals.__sub__(o)	Substract an interval from other using - operator

tnetwork.utils.Intervals.__init__

Intervals.**__init__**(*initial=None*)

Instantiate intervals

Instanciate an intervals object. Can be initialized by a list of intervals

Parameters **initial** – a single interval as a pair (start, end), or a list of pair or an Interval object

tnetwork.utils.Intervals.add_interval

Intervals.**add_interval**(*interval*)

Add the provided interval to the current interval object.

Note that the method is relatively slow since all cases need to be checked. One could use a specific, optimized function to add specifically at the end: _add_interval_at_the_end

Parameters **interval** – provided as a pair (start, end)

tnetwork.utils.Intervals.__add__

Intervals.**__add__**(*o*)

Add two Intervals using + operator

```
>>> a = Intervals((0,2))
>>> b = Intervals((1,6))
>>> c = a+b
```

Parameters **o** – other interval

Returns

tnetwork.utils.Intervals.__sub__

Intervals.**__sub__**(*o*)

Substract an interval from other using - operator

```
>>> a = Intervals((0,6))
>>> b = Intervals((1,2))
>>> c = a-b
```

Parameters **o** – other interval

Returns

Accessing Intervals properties

Intervals.contains_t(t)	Return True if the provided t is in the current Intervals
Intervals.contains(period)	Is the period contained in this Interval
Intervals.__contains__(time)	Defines the in operator
Intervals.periods()	Return the periods as a list of pairs (start, end)
Intervals.duration()	Duration of the interval
Intervals.start()	First date of the Intervals
Intervals.end()	Last date of the interval

tnetwork.utils.Intervals.contains_t

Intervals.**contains_t**(*t*)

Return True if the provided t is in the current Intervals

Parameters **t** – a time step to test

Returns True if the time is in the interval, False otherwise

tnetwork.utils.Intervals.contains

Intervals.**contains**(*period*)

Is the period contained in this Interval

Check if the provided period is included in the (active time of the) current Interval

Parameters **period** – the period to test

Returns True or False

tnetwork.utils.Intervals.__contains__

Intervals.**__contains__**(*time*)

Defines the in operator

```
>>> a = Intervals((0,6))
>>> b = Intervals((1,2))
>>> if b in a:
>>>     print("b is contained in a")
```

Parameters **o** – other interval

Returns

tnetwork.utils.Intervals.periods

`Intervals.periods()`

Return the periods as a list of pairs (start, end)

Returns list of pairs

tnetwork.utils.Intervals.duration

`Intervals.duration()`

Duration of the interval

Return the duration of this interval, i.e. the sum of the difference between end and start for all periods in the current interval object. :return:

tnetwork.utils.Intervals.start

`Intervals.start()`

First date of the Intervals

Returns int

tnetwork.utils.Intervals.end

`Intervals.end()`

Last date of the interval

Returns int

Operations

Intervals.intersection(other_Intervals)	Intersection with another Intervals
Intervals.union(other_Intervals)	Union with another Intervals
Intervals.__eq__(other)	Defines the = operator

tnetwork.utils.Intervals.intersection

`Intervals.intersection(other_Intervals)`

Intersection with another Intervals

return the intersection between the current interval and the one provided as parameter, i.e. a new Interval containing periods in common between them.

Parameters **intervals** – intervals provided as a Intervals object

Returns a new Intervals object

tnetwork.utils.Intervals.union

`Intervals.union(other_Intervals)`

Union with another Intervals

Return the union between the current interval and the one provided as parameter, i.e. a new interval containing all sub-intervals of both. (if they overlap, it is handled)

Parameters **`intervals`** – intervals provided as a Intervals object

Returns a new Intervals object

tnetwork.utils.Intervals.__eq__

`Intervals.`**`__eq__`**`(`*other*`)`

Defines the = operator

Checks if two intervals cover the same periods :param other: :return:

Index

Symbols

A