Journal of Complex Networks (2021) **00**, 1–24
doi: 10.1093/comnet/cnaa027

Evaluating community detection algorithms for progressively evolving graphs

Remy Cazabet[†]
Univ de Lyon, CNRS, Université Lyon 1, LIRIS, UMR5205 Villeurbanne France
[†]Corresponding author. Email: remy.cazabet@gmail.com

Souâad Boudebza
Université Mohamed Seddik Benyahia de Jijel, BP 98 Ouled Aïssa, 18000, Jijel, Algeria

and

Giulio Rossetti
Information Science and Technology Institute of the Italian National Research Council Pisa, Italy

Edited by: Prof. Matjaz Perc

[Received on 3 June 2020; editorial decision on 17 July 2020; accepted on 20 July 2020]

Many algorithms have been proposed in the last 10 years for the discovery of dynamic communities. However, these methods are seldom compared between themselves. In this article, we propose a generator of dynamic graphs with planted evolving community structure, as a benchmark to compare and evaluate such algorithms. Unlike previously proposed benchmarks, it is able to specify any desired evolving community structure through a descriptive language, and then to generate the corresponding progressively evolving network. We empirically evaluate six existing algorithms for dynamic community detection in terms of instantaneous and longitudinal similarity with the planted ground truth, smoothness of dynamic partitions and scalability. We notably observe different types of weaknesses depending on their approach to ensure smoothness, namely *Glitches*, *Oversimplification* and *Identity loss*. Although no method arises as a clear winner, we observe clear differences between methods, and we identified the fastest, those yielding the most smoothed or the most accurate solutions at each step.

Keywords: Dynamic networks, Community detection, Dynamic communities, Network generator.

1. Introduction

Many algorithms have been proposed in recent years to discover evolving communities in dynamic networks. Because few empirical comparisons of them have been conducted, their relative strengths and weaknesses are mostly unknown. A likely reason for the scarcity of such work is the lack of reliable benchmarks to generate synthetic graphs, that is, an equivalent to the LFR benchmark [1] in static settings. Several benchmarks have already been proposed (e.g. [2–5]), but none of them allow to generate a dynamic network corresponding to a scenario of community evolution described by the experimenter.

In this article, we will focus on the problem of detecting communities in *progressively evolving graphs*, that is, graphs for which the graph is well defined at any given time, and change at a slow rate. Such graphs are common in human activities [6], for instance friendships in social networks, physical infrastructures (electricity/transport network, etc.) and physical proximity between individuals captured at a high rate using personal sensors [7].

CAZABET ET AL.

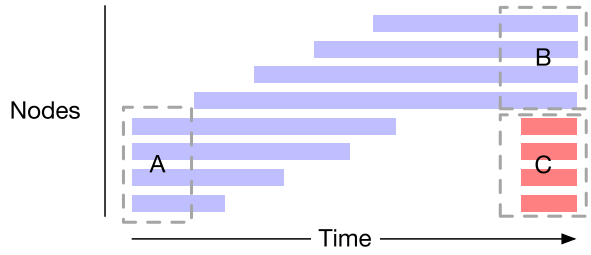


FIG. 1. Illustration of the Ship of Theseus paradox. C is composed of the same nodes as A, but A progressively transforms into B. The choice to consider that B or C is *the same community* as A therefore depends only on the community life-cycle.

Since reproducibility is paramount in such a work, we provide an open-source implementation of the benchmark generator, algorithm implementations and evaluation scores as a fully documented python library and a notebook allowing to reproduce the results.[1]

1.1 *Ship of Theseus and the nature of dynamic communities*

Communities lifecycles—their history, the events they undergo, etc.—are of utmost importance since they are what makes the difference between static and dynamic community detection. Indeed, two algorithms that agree on what is the best partition for each static graph composing the dynamic network might still disagree on what the corresponding *dynamic* communities are. A good example of this problem is to consider the *ship of Theseus* paradox.

The paradox of the ship of Theseus is an ancient thought experiment about the identity of an object evolving through time. It can be formulated as follows:

Let us consider a famous ship, the ship of Theseus, composed of planks, and kept in a harbour as a historical artefact. As time passes, some planks deteriorate and need to be replaced by new ones. After a long enough period, all the original planks of the ship have been replaced. Can we consider the ship in the harbour to still be the same ship of Theseus? If not, at which point exactly did it cease to be the same ship?

Another aspect of the problem arises if we add a second part to the story. Let's consider that the old planks were stored in a warehouse, repaired, and that a new ship, identical to the original one, is built with them. Should this ship, just built out, be considered as the *real* ship of Theseus?

Let's call the original ship A, the ship that stayed in the harbour B, and the reconstructed from original pieces, C.

In terms of dynamic community detection, this scenario can be modelled (Fig. 1) by a progressively evolving community c_1(ship A), that nodes leave one after the others until all of them have been replaced (ship B). A new community c_2 appears after that, composed of the same nodes as the original community c_1 (ship C).

[1] Library: https://tnetwork.readthedocs.io/en/latest/, last accessed 21 August 2020.
Experiments reproduction: https://tinyurl.com/y7a2lrbz, last accessed 21 August 2020.

A static algorithm analysing the state of the network at every step would be able to discover that there is, at each step, one community (at the beginning) and two at the end. But the whole point of dynamic community detection is to yield a longitudinal description, and therefore, to decide when two ships at different points in time are *the same* or not.

The benchmark we propose is designed to represent complex evolution scenarios such as the ship of Theseus, in addition to usually defined events such as merge and split. It allows representing progressive changes for each event, for example, add and remove edges incrementally such that two originally distinct communities merge into a single one. Unlike previous benchmarks, any community evaluation scenario can be described and generated using an appropriate language, and the network has both stable links and a community structure with known properties. We detail the difference with previous methods in Section 2.

The rest of the article is organized as follows. In Section 2, we introduce previously proposed benchmarks, and we emphasize the added value of our proposition. In Section 3, we detail the generation process of our benchmark. Finally, in Section 4, we compare several algorithms with different smoothing approaches on networks generated using the proposed benchmark.

2. Related works

A few methods have already been introduced in the literature to generate benchmark graphs for progressively evolving communities.

In Granell *et al.* [3], two cyclic scenarios are proposed: one generates nodes migration (a set of nodes switch from a larger community to a smaller, and back), the other generates sequences of Merge-Split. In both cases, communities are defined as Stochastic Block Models (SBM), parameterized by fixed internal density p^{in} and external density p^{out}.

In Bazzi *et al.* [2], a generic method is introduced to generate multilayer networks with community structures. It requires to define an *interlayer dependency tensor* encoding the probability for node u_i in layer l_a (u_i, l_a) to copy its community assignment from node u_j in layer l_b (u_j, l_b). In the most simple case, for dynamic networks, the community of (u_i, l_t) is defined as depending only on the affiliation of the same node in the previous layer (u_i, l_{t-1}). A random iterative process is used to attribute nodes to communities in each layer, satisfying both the constraints of the interlayer dependency tensor and a chosen distribution of community sizes. Edges are added in a second step, independently for each t, according to a degree corrected SBM, parameterized by a unique mixing parameter $\mu \in [0, 1]$, where 0 corresponds to all edges falling inside communities, and 1 corresponds to no community structure. Note that edges are picked independently at each step, and events such are split or merge cannot be represented.

In RDYN [4], community's lifecycles are created randomly based on rules: events (merge/split) have a probability to occur, and when an event occurs, involved communities are chosen by a random selection biased by the size of communities. Edges evolve gradually by combining two processes: a decay that makes old edges disappear, and a biased growth that reinforces community structure. The generation is driven by parameters, such as number of nodes, average degree, mixing coefficient, probability of node appearance, probability of node action, etc. Because the structure of communities at any given time is not necessarily well defined, the quality of communities is evaluated at each step using *conductance*, and communities are included in the ground truth only when the value is higher than a threshold. This benchmark generates progressive events with complex lifecycles, but the evolution of communities is fully determined by internal mechanisms, and one cannot represent custom scenarios. The properties of generated communities are also not fully known, apart from their conductance property, since it is the result of an ad hoc dynamic process.

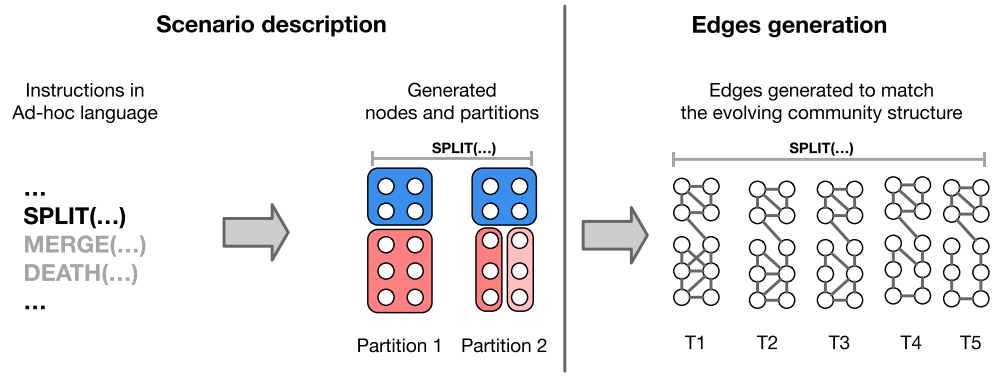


FIG. 2. Illustration of the benchmark generation process.

The method proposed by Sengupta *et al.* [5] yields overlapping communities. At each step, a community event (birth, death, split and merge) occurs with probability p and a node event (add and remove) with probability p'. The initial community structure is generated by a static algorithm [8]. Edges inside each community are distributed randomly (ErdÅ's–Renyi graph), with a probability decreasing with size: $\frac{\alpha}{n^\gamma}$ with n the number of nodes and $\alpha, \gamma \in]0, 1[$ chosen parameters. Edges are later modified randomly according to two factors: (1) random modifications at each step and (2) gradual evolution to match changes in the community structure. For instance, if two communities merge, edges to add are drawn at random among disconnected pairs of nodes in the resulting community in order to reach the desired density. Rules are introduced to ensure that communities stay coherent (not becoming too small, not allowing simultaneous operations on the same nodes, etc.) Limitations are comparable to those of RDYN.

In several other articles, notably [9–16], ad hoc benchmarks were introduced, usually to test one specific scenario, with similar or more restricted scopes.

Unlike all previous methods, the benchmark we propose introduces a language to represent any scenario of community evolution by specifying events (merge, split, etc.), either through its complete description, or by drawing randomly sequences of events (see Section 4.2 for examples). It also generates a network with (1) stable links (links present in t are likely to be present in $t + 1$), (2) communities with known properties (see Section 3.2.1 for details) and (3) able to represent progressive events, such as a progressive merge or split.

To the best of our knowledge, a single paper has been published so far comparing empirically dynamic community detection algorithms: in [17], five methods have been tested on RDyn benchmark [4]. They were compared in terms of average community quality at each step. In this article, we compare on different aspects, by introducing measures of smoothness and longitudinal quality (see Section 4.3)

3. Synthetic network generation process

The benchmark we propose follows a two-step process (Fig. 2):

(1) **Scenario description**: The experimenter defines initial communities and the scenario of their evolution (sequence of events).

(2) **Edges generation**: Edges are generated by a partly random process. They form a dynamic network corresponding to the described community structure, satisfying some community quality properties.

3.1 *Scenario description*

Any scenario can be described by a set of *community events*, each of them modifying the affiliation of nodes. To represent these community events, we define instructions allowing to represent the most common ones (Merge, Split, Birth, Death, etc.), or any arbitrary, more complex event.

We therefore define a language allowing to describe algorithmically any community evolution scenario. This language is composed of instructions, in the following form:

C ← **EVENT**(**parameters**)[**delay**,**triggers**]

Where

- **C** is a list of communities yielded by this event, a community being a tuple $< \text{ID}, L, N >$ with:
  - ID a unique identifier for this community. Each community yielded by an event has a new, unique ID, for example, an event that takes a community and removes one of its node will yield a community with a new unique ID.
  - L the *label* associated with this community. The label corresponds to the *identity* of the community: for example, after a split, we can choose to attribute new labels to both resulting communities, or to give the label of the split community to one of the resulting ones.
  - N the set of nodes composing this community.
- **EVENT** is the type of event (e.g. MERGE, BIRTH, etc.)
- **parameters** differ for each event. They are, for instance, communities to merge, labels associated to the yielded communities, etc.
- **triggers** are the set of communities that must be ready (have been yielded by a previous event) for this event to be triggered
- **delay** is a number of steps to wait before starting the event after the triggers conditions are fulfilled.

To sum up, the instruction above means that the event *EVENT* will start *delay* steps after all communities in *triggers* appeared. This trigger/delay mechanism allows to define complex relations between communities, for instance, the division of a community being triggered by the apparition of another—topologically unrelated—community.

We first define a single event, *Assignment*, allowing to represent any change between an arbitrary number of communities.

ASSIGN(BEFORE-COM, AFTER-NODES, AFTER-LABELS)

- *BEFORE* is an ordered list of communities that will be modified
- *AFTER-NODES* is an ordered list of sets of nodes, each set of nodes corresponding to a community yielded by this event
- *AFTER-LABELS* is an ordered list of labels to attribute to yielded communities.

```
Input: BEFORE-COM, AFTER-NODES, AFTER-LABELS
Global AC: set of currently active communities
begin
    for c ∈ BEFORE-COM do
        AC ← AC \ c
    end
    NEW-COMS ← [] ;
    for i ← 0 to length(AFTER-LABELS) do
        NEW-COMS[i] ← <NEW-C-ID(), AFTER-LABELS[i], AFTER-NODES[i]> ;
        AC ← AC ∪ NEW-COMS[i] ;
    end
    return NEW-COMS
end
```

Algorithm 1: instruction **ASSIGN**. **NEW-C-ID()** is a function that generate a new, unique community identifier.

Most articles in the literature agree on a set of commonly found events impacting communities, such as SPLIT and MERGE.

In Appendix A, we define some of these common events, based on the ASSIGN event: BIRTH, DEATH, MERGE, SPLIT, INITIALIZE and THESEUS, which corresponds to the ship of Theseus paradox presented in the Section 1. The benchmark python library contains definition for additional events: CONTINUE (a community continues without change for a given period), RESURGENCE (a community disappears and re-appears with identical nodes some time later), and operations of progressive, node by node change: GROW-ITERATIVE, SHRINK-ITERATIVE and MIGRATE-ITERATIVE (nodes migrate from one community to another).

These instructions can be combined to define any scenario, either by listing all desired events or by writing a program to generate scenarios by picking instructions randomly. See Section 4.2 for examples.

3.2 *Edges generation*

In the previous section, we have seen how to define the community evolution scenario. In this section, we address the generation of edges to fit this scenario.

Communities can have, at each step, two states: *stable*, when it is not involved in any event, or *evolving* otherwise. In the state *stable*, edges of the community are generated following the *Deterministic Strongly Assortative Block Model* (DSABM) (See Section 3.2.1).

When an event is triggered, the communities involved switch to the *evolving* state. Internal edges are known according to the DSABM before and after the event. Therefore, we make one edge modification (addition/removal) at each step of the dynamic network evolution, until reaching the final state. At each step, external edges are also generated according to the DSABM.

3.2.1 *Deterministic strongly assortative block model*

A common approach to generate static community benchmarks is to use a SBM [18]. An edge probability matrix P of size $r \times r$ is defined, with r the number of blocks (communities), and edges between each pair of blocks are generated according to this probability matrix. We adopt a variation of the SBM, that we call DSABM.

A block model is said to be *Strongly Assortative* if $P_{ii} > P_{ij}$ for each i and j such as $i \neq j$. This corresponds to the original definition of communities as *groups of nodes that are more densely connected*

than the rest of the network. Note that we adopted a Strongly Assortative structure for the sake of simplicity, the extension to other Block Models is straightforward.

We adopt a *deterministic* block model to solve the problem of the generation of *slowly evolving* community structures. In the situation when one wants the network topology to follow a block structure and to allow this block structure to evolve, one needs to comply with two apparently antagonistic requirements:

- **Economy of change**: we want as few edge modifications as possible to go from a network satisfying the partition t_1 to one satisfying another partition t_2.
- **Random internal structure**: we want the graph at each step to be compatible with the definition of block models, that is, edges between communities should be chosen at random, and not depend on previous partitions.

These two requirements are antagonistic as soon as the internal density of communities changes between two steps. We can illustrate this problem with the following example: let us assume that we have at time t two disconnected communities of size 4 and density 1. Each community contains six edges. Let us now consider that the two communities merge at time $t + 1$ into a single community of density 0.5. This community should have 14 internal edges. If we try to maximize the *economy of change* objective, we will add only two edges chosen randomly among the missing edges. The resulting community will thus be composed of two cliques of size 4 connected by only two edges, a structure very unlikely to obtain through a random edge selection for a single community of size 8 and density 0.5.

Conversely, maximizing the random internal structure by resampling independently networks at each step will lead to unstable edges, in particular for sparse blocks. This would not be compatible with a scenario of a progressive evolution from a network with a partition to a network with another one.

In previous benchmarks using SBM, the economy of change is usually ignored and random edges are generated at every step [2]. In Granell *et al.* [3], a solution is introduced: all pairs of nodes in the initial graph are ordered in a random fashion, and when edges need to be added or removed to reach the final state, they are chosen according to this predefined order.

We propose to generalize the method introduced in [3] to make it work for any type of scenario.

To each pair of nodes is assigned a fixed Latent Affinity score $\Omega \in [0, 1]$. When a new node n is added to a network $G = (V, E)$, a random value is assigned to each pair between n and every node in V. When we need to attribute edges for a pair of blocks (communities c_1, c_2), we first compute the number q of edges according to the probability matrix $P_{c_1 c_2}$. (Therefore, interpreted as the *fraction of existing edges* rather than an independent probability of observing each edge). We then select the q pairs of highest Ω among pairs of nodes in this community.

The number and the position of edges in a block are therefore selected in a *deterministic* way for a given community structure and given latent affinities. Note that several runs of the same dynamic community scenario nevertheless lead to different dynamic networks, since latent affinities are assigned randomly.

3.2.2 *Communities density* The DSABM used to generate the network corresponding to the desired community structure requires to define a Probability Matrix P. As for any Strongly Assortative SBM, we want $P_{ii} > P_{ij} | i \neq j$. This is commonly solved by selecting two parameters p^{ext} and p^{in}, such that $P_{ij} = p^{\text{ext}}$ and $P_{ii} = p^{\text{in}}$ However, we think that this choice is unrealistic for partitions with communities

of heterogeneous sizes: intuitively, a community of size 3–5 must have an intern density >0.7 to be well defined, while a community of size 100 with a comparable density seems unlikely in empirical networks. A community *growing* in size should therefore see its density *shrink*. Additionally, it has been observed for dynamic graphs that the average degree tends to grow with the graph size [19]. To validate empirically those intuitions, we compute the density and average degree of communities in a collection of large graphs with ground-truth communities from the SNAP dataset repository [20]. We use two definitions of communities:

(1) The so-called *Ground Truth Communities*, corresponding to collected labels of nodes. Note that the communities found are not defined by the topology of the network, but based on metadata.
(2) Communities found in the same networks by the Louvain algorithm [21]. Communities are thus topologically defined and correspond to a high value of Modularity.

We can observe in Fig. 3 that the average density of communities tends to shrink with their sizes, while their average degree tends to grow, independently of the definition of communities considered. The slope of these trends depends on the network.

We therefore propose to model the density of each community using a simple function, compatible with observations. We define the average degree of a community c, relatively to its number of nodes n_c and a density coefficient $\alpha \in]0, 1]$ as:

$$\bar{k_c} = (n_c - 1)^{\alpha}$$

The density of each community c is thus defined as:

$$p^{in}(c) = \frac{n_c(n_c - 1)^{\alpha}}{n_c(n_c - 1)} = (n_c - 1)^{\alpha - 1}$$

And the number of internal edges in a community:

$$m_c = \left\lceil \frac{n_c \times \bar{k_c}}{2} \right\rceil$$

If $\alpha = 1$, communities are cliques, and the density decreases faster with size for lower α.

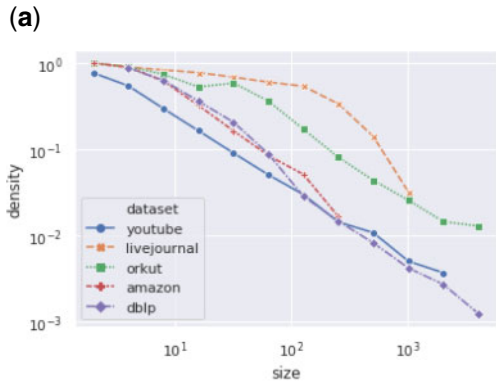
The external density p is defined as: $p = \beta p^{\in}(c_V)$ with β a parameter of community identifiability and c_V the whole graph seen as a community.

As a consequence, the internal and external density of all communities and their evolution with size are fixed with only two parameters: $\alpha, \beta \in [0, 1]$.

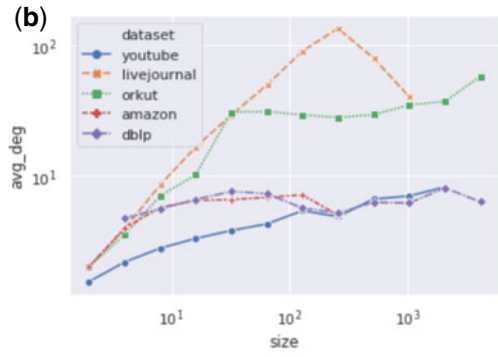
These parameters can be chosen to vary the sharpness of communities as for any SBM-based approach. In this article, we vary $\alpha \in [0.5, 1]$ and $\beta \in [0, 0.5]$.

3.3 *Random punctual noise*

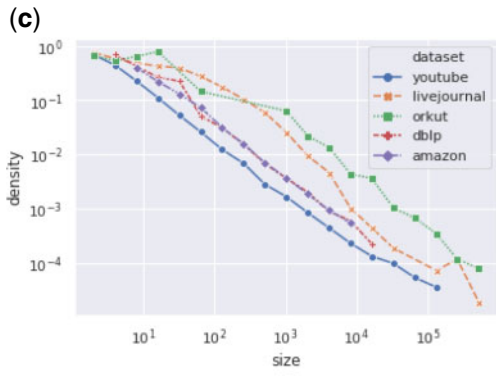
Edges generated by the DSABM are deterministic for a given Ω, that is, if the communities do not change, the graph also stays unchanged. It has been proposed [22] that in real dynamic networks, one can differentiate a stable *backbone* from random, short-lived fluctuations. We add a parameter $\beta_r \in [0, 1]$ to our benchmark, such that, at each step, a fraction β_r of edges are rewired at random to differentiate



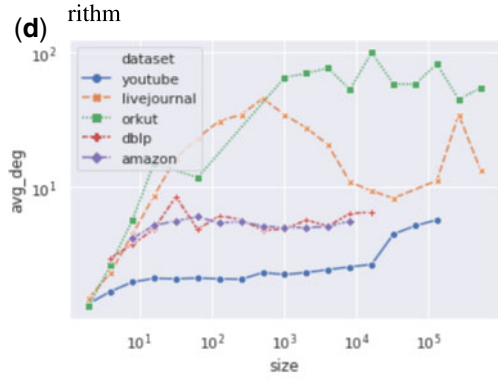
Average of densities of communities, Ground Truth Communities



Average of internal mean degree of communities, communities discovered by the Louvain algorithm



Average of densities of communities, communities discovered by the Louvain algorithm



Average of internal mean degree of communities, communities discovered by the Louvain algorithm

FIG. 3. Relation between densities/mean degrees of communities relatively to their size in several large networks with available ground-truth communities. We observe similar trends for most networks: density decreasing with size, average internal degree increasing with size.

the imperfections in community structures that are part of the backbone (controlled by β) and the ones which are not (β_r).

3.3.1 *Algorithmic complexity* Each community being handled independently, the complexity of the generation process is not prohibitive: networks with hundreds of nodes and thousands of evolution step can be generated in a few seconds, and with thousands of nodes and tens of thousands of steps in a few minutes. The generation of very large networks is nevertheless not possible with the current implementation, due to the usage of the latent affinity score, which requires storing n^2 values. This constraint could be removed by using more advanced methods, for instance based on deterministic hash functions.

4. Experiments

In this section, we evaluate several algorithms on networks generated using the proposed benchmark. We first introduce the algorithms to compare, and then conduct three experiments: qualitative evaluation on a complex scenario, quantitative evaluation on randomly generated networks, and scalability evaluation.

4.1 *Algorithms*

Many algorithms for Dynamic Community Detection have been proposed in recent years [23]. Among them, we have selected six algorithms with different smoothing techniques. Our goal is not to search for the best algorithm, but rather to study the consequences of choices made to integrate the dynamic into the community detection process. We have selected algorithms based on the following criteria:

- Being based on *Modularity optimization*. We want all methods to agree on the definition of the best static partition on a single network, so that their differences depend only on the dynamic of the network. We chose the Modularity optimization approach because it is the most widespread, although a similar work could be done with SBM or Matrix factorization [24] based approaches, for instance.
- They represent well the variety of approaches used to tackle the dynamic aspect
- Their source code is available, or implementing them faithfully is not too difficult.
- They are scalable enough. We had for instance to discard popular methods such as DYNMOGA [25], Estrangement Confinement [26] and FacetNet [12], whose complexity is not compatible with having hundreds of steps of evolution.

The algorithms compared in this article are the following:

No-Smoothing: The approach we will use as a reference consists in applying a static algorithm on the snapshot at each step, and then matching the most similar communities in consecutive steps, based on the Jaccard Coefficient. We use the Louvain method [21] at each step, and the matching process, common to several approaches, is described in Section 4.1.1.

Implicit-Global This method introduced in [27] uses a form of *implicit smoothing* [23]: at each step, the Louvain algorithm is run, but instead of starting it with each node in its own community, the previous partition is used as seed.

DYNAMO [28] is a recent method updating at each evolution step the community structure according to changes in the graph, based on a set of local rules. The primary goal is to be faster than *No-smoothing* while reaching similar Modularity scores, by avoiding to recompute communities from scratch at each step. However, as Implicit Global, it also introduces some smoothing by staying close to a previous local minimum (*implicit local* smoothing). We used the implementation by the authors.

Smoothed-Graph This method is a variant of the one proposed in [29]. A community detection algorithm (in our case, Louvain) is run at each step t on a graph whose smoothed adjacency matrix is defined as follows: $A^t_{ij} = \alpha A^t_{ij} + (1-\alpha)C^{t-1}_{ij}$ where $C^{t-1}_{ij} = 1$ if i and j belongs to the same community at step $t-1$, 0 otherwise.

Transversal-Network is a popular method introduced by Mucha *et al.* [30] with a *Cross-Time* approach, that is, communities at t depends on earlier and later steps of the network. The principle of the method is to build a single transversal network by adding inter-snapshot coupling links and to apply a Louvain-like community detection algorithm on this network, based on an adapted version of the modularity. We used the original implementation by the authors.

Label-Smoothing is a method introduced by Falkowski *et al.* [31] whose first step is the same as the No-Smoothing algorithm, but instead of matching communities between pairs of successive steps, a *Community survival graph* is created by considering each community in each step as a node, and an edge connects any two communities with a Jaccard coefficient above a threshold, with the Jaccard value as weight. Community detection is applied to this graph to define *communities of static communities*, thus defining dynamic communities. We implement it using the Louvain algorithm for both steps, and similar parameters as for the matching method described in Section 4.1.1.

All algorithms have several parameters that could be modified to improve the results. However, community detection being an un-supervised problem by definition, these parameters usually cannot be tuned. We therefore used the default parameters from the implementation (DYNAMO) or used by the authors themselves ($\omega = 0.5$, [30]). $\alpha = 0.9$ [29]).

4.1.1 *Persistent labels attribution*

Methods *No-Smoothing*, *Implicit-Global*, *DYNAMO* and *Smoothed-Graph* detect communities at each step, with or without smoothing, but do not attribute persistent labels to communities. For all those methods, we therefore attribute persistent labels to communities using the same procedure, inspired by [11]. First, a similarity score is computed between any two pairs of communities between adjacent snapshots using the Jaccard coefficient, and any pair of communities with a value of similarity above a threshold (0.3, in our experiments) is considered a potential match. Then, two communities c_a in t and c_b in $t+1$ are matched if (i) c_b is the most similar community to c_a in $t+1$ AND (ii) c_a is the most similar community to c_b in t. If a community in $t+1$ is not matched to any community in t, it receives a new label. When creating our benchmark, we respect this logic, that is, in case of merge or split, the communities the most similar before and after the event share the same label (we avoid ties in our scenarios).

4.2 *Qualitative evaluation on an ad hoc scenario*

In this section, we generate a small, deterministic scenario to observe qualitatively how each algorithm behaves. The scenario is designed to include several particular cases such as a Ship of Theseus, resurgence of communities, and successive merge and split of communities. It is described as follows (note that the algorithm is a functional python code, using the provided implementation library):

```
# 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,  name="R", delay=25)
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)
```

Listing 1. Defining an ad hoc scenario. Resurgence is an operation that makes a community disappear and reappear with the same label after a delay.

We generate two networks using this scenario with different parameters, a *sharp* scenario ($\alpha = 0.9, \beta = 0.05$), and a *blurred* one ($\alpha = 0.8, \beta = 0.25$). We add a small random noise $\beta_r = 0.01$.

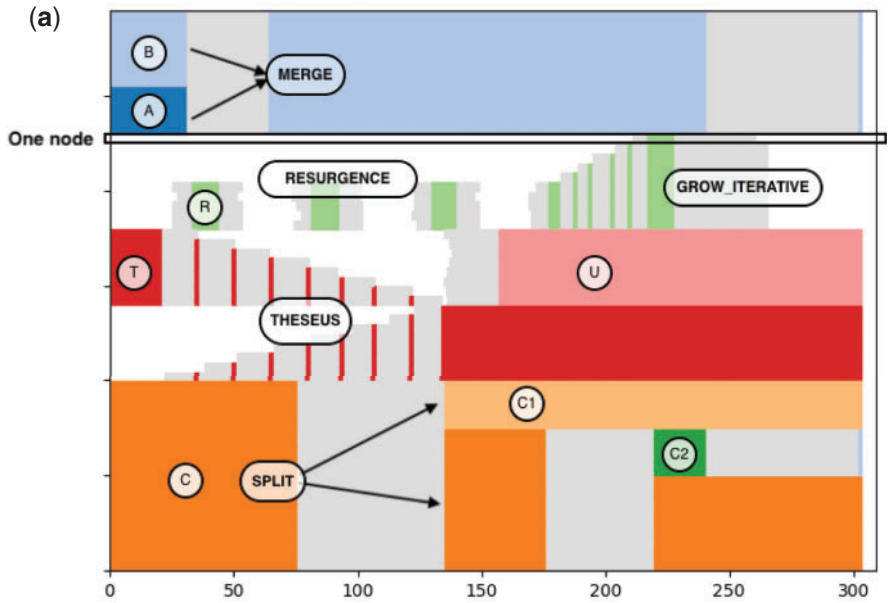
In Fig. 4, we can see the ground truth communities corresponding to this scenario, together with the initial state of the network at $t = 0$ for both sets of parameters. To represent the planted communities, we use the Temporal Activity Map (TAM [32]) visualization approach, that is, each node has a fixed vertical position, edges are not represented, time is on the horizontal axis, and colours correspond to community affiliations (two nodes with the same colour belongs to the same community, whether they are in the same timestep or different ones). Nodes appear grey when they have no known affiliation, which corresponds to periods during which events are *on-going*, affected communities not being properly defined. Nodes not present in the network at a given time appear white.

In Fig. 5, we can compare the results obtained by the No-Smoothing and the Label-Smoothing approaches on the sharp variant. We can observe that the non-smoothed algorithm already matches quite well the ground truth, without too much instability. When using the Label-Smoothing approach, despite having the exact same partition at each step initially, some important differences arise: (i) the resurgent community is identified as such (yellow community for all of them), (ii) the ship of Theseus is split differently.

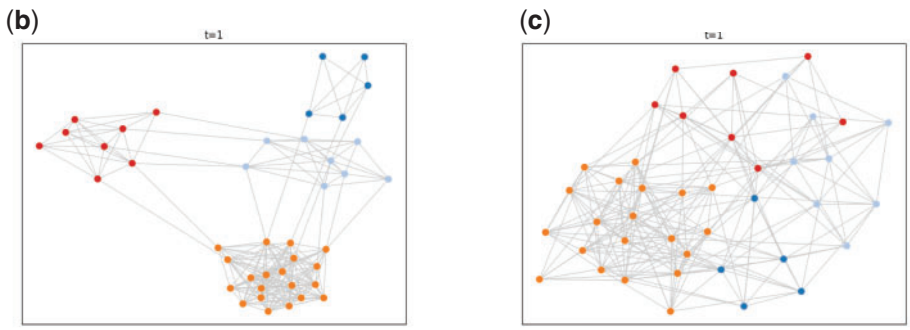
In Fig. 6, the same results are shown for all methods on the blurred variant. From this figure, we define three types of problems that can be observed in dynamic communities:

- *Glitches* corresponds to individual nodes switching arbitrarily between communities for short periods.
- *Identity loss* corresponds to a community label being lost and replaced by a different one due to a short-lived change, while the community stayed mostly coherent.
- *Oversimplification* corresponds to topologically distinct communities at a given time being merged to improve the smoothness of the solution.

We can observe that there are much fewer glitches in some smoothed approaches (Implicit-Gobal, Smoothed-Graph) compared with No-Smoothing or DYNAMO. Similarly, Identity loss observed in the No-Smoothing approach for the ship of Theseus (orange $\rightarrow$ green $\rightarrow$ blue $\rightarrow$ purple, etc.) is partially solved by Label-Smoothing, Implicit-Global and Smoothed-Graph. However, Oversimplification also appears, for instance with communities A and B being considered as a single one for Smoothed-Graph.



Planted dynamic communities, represented using the TAM visualization. Each node is represented as a thin horizontal line. Colors represent communities. Grey areas represent anbiguous affiliations. Events are identified by arrows and names, e.g., blue communities A and B merge into a singe community identified as B, and this process last from steps 30 to 60.



The static graph at time t=0, version *sharp* ($\alpha = 0.9, \beta = 0.05, \beta_r = 0.01$)

The static graph at time t=0, version *blurred* ($\alpha = 0.8, \beta = 0.25, \beta_r = 0.01$)

FIG. 4. A simple scenario of community evolution.

4.3 *Quantitative evaluation of quality on a random scenario*

In order to quantitatively evaluate the quality of dynamic communities discovered by each algorithm, we design a generator of random scenarios, based on the following principle: given a number of communities m, a minimal and maximal community sizes $s^{\min}$ and $s^{\max}$, and a number of operations o, we repeat o times the following process (code available in the implementation):

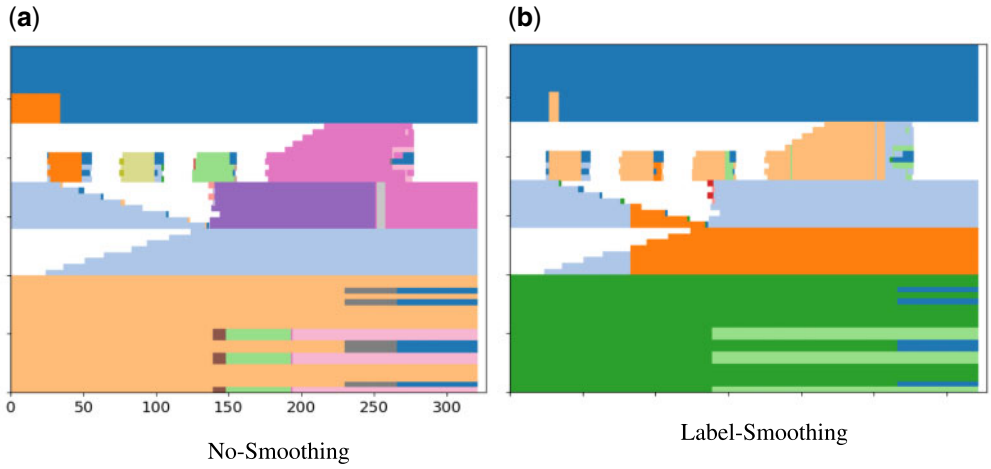


FIG. 5. Comparison of partitions obtained using two different methods on the ad-hoc scenario, *sharp* flavour. Most communities are captured accurately, with some key differences: resurgence events are identified by Label-Smoothing but not by the other, The ship of theseus is labelled differently, etc.

(1) Pick a community c at random.

(2) If $|c| > s^{\max}$, split it in two, the largest resulting community inherits $\frac{2}{3}$ of its nodes.

(3) Else, merge it with the smallest remaining community.

4.3.1 *Scores for the evaluation of dynamic communities* All algorithms selected for evaluation have the same quality function in the static case: Modularity. We introduce quality functions to assess different aspects of the quality of dynamic communities: evaluation of their quality at each point in time, of their smoothness, and of their longitudinal quality.

Evaluation at each step

First, we use scores to evaluate the quality of communities at each step taken independently. As already done for instance in [4], we use the average values for each step of static scores. We use average Modularity $\overline{Q}$ to assess the intrinsic quality, average AMI $\overline{\text{AMI}}$ and average ARI score $\overline{\text{ARI}}$ to compare with the ground truth. AMI is the adjusted for chance version of the Normalized Mutual Information, while ARI is the adjusted rand index. Note that, in theory, a static approach run at each time without smoothing should have the highest values for these metrics, thus high scores in these metrics alone are not good measurements of the quality of dynamic communities.

Evaluation of smoothness

We introduce 3 scores to evaluate the smoothness of dynamic partitions.

- **SM-P** is defined as 1 - the average NMI between all pairs of successive snapshots, and measure the *smoothness at the level of* ***partitions***. (Higher is better)
  More formally, SM-P $= 1 - \frac{1}{T-1}\sum_t^{T-1} NMI(G_t, G_{t+1})$
- **SM-N** is defined as the inverse of the number of affiliation change (summed for all nodes). It measures the *smoothness at the level of* ***nodes***. (Higher is better)

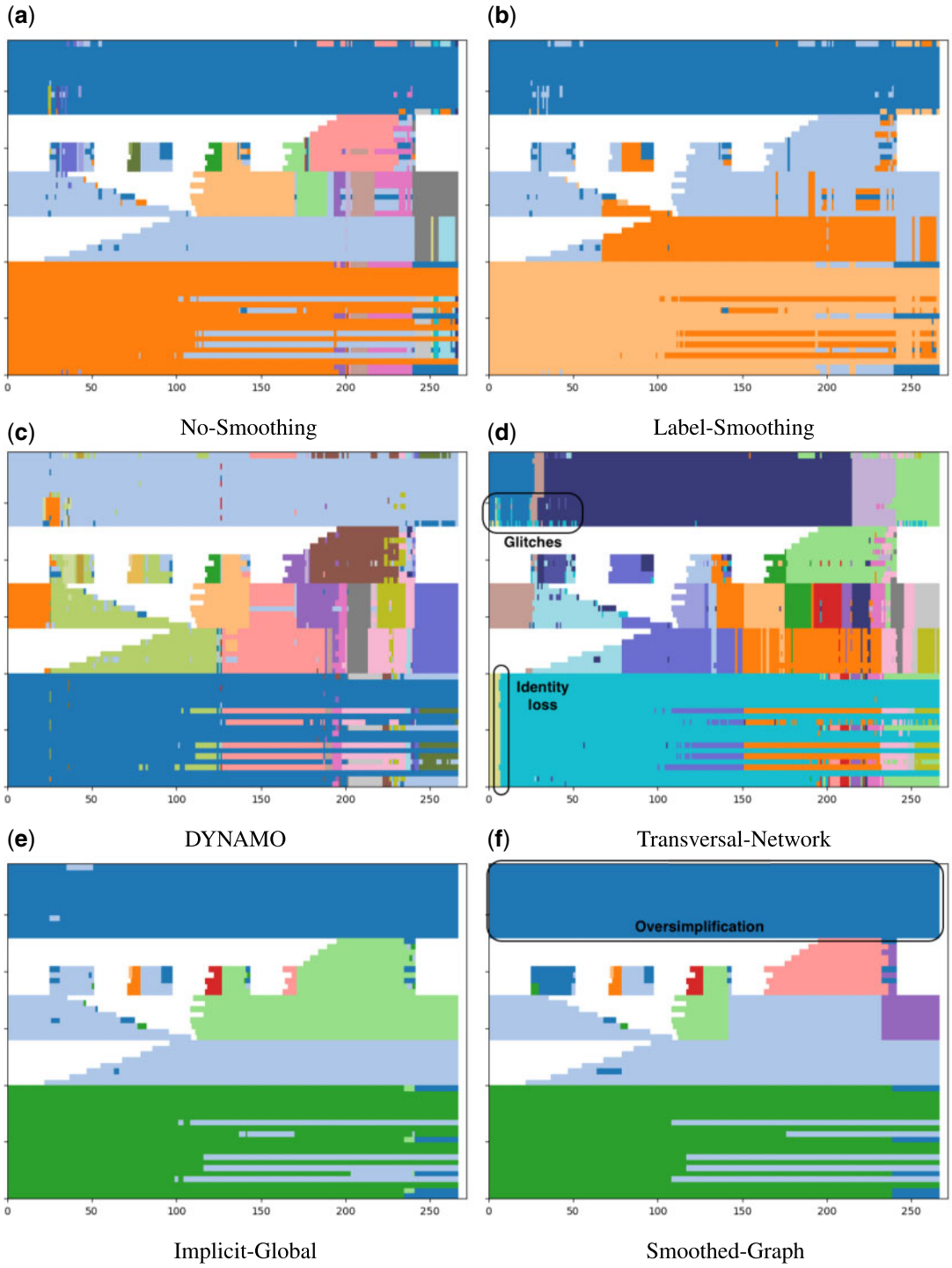


FIG. 6. Comparison of partitions obtained using all methods on the ad-hoc scenario, *blurred* variant. We annotated exampled of typical smoothing problems: Glitches, Identity loss and Oversimplification

More formally, SM-N = $1/(\sum_n^V \sum_t^{T-1} \delta_{L(n,t),L(n,t+1)})$, with $L(n,t)$ the label of node n at time t and $\delta_{i,j}$ the kronecker delta, that is, $\delta_{i,j} = 1$ if $i = j$, 0 otherwise.

- **SM-L** is defined as the inverse of the average Shannon entropy of node labels, that is, for each node, we compute the entropy H of its label affiliations, and SM-L is the inverse of the average among all nodes. It measures the *smoothness at the level of* ***labels***. (Higher is better)
  More formally, SM-L = $1/\sum_n^V H(L(n))$, with H the Shannon Entropy and $L(n)$ the probabilities of observing each label when picking node n at a random time t.

It is important to note that they measure different aspects of smoothness: SM-P is independent of labels and is little impacted by the instability of single nodes. SM-N is sensible to glitches (short-lived, spurious changes). SM-L is little sensible to glitches but is impacted by long-term instability, for example, an unjustified label change which is not reversed.

Longitudinal scores

Finally, we introduce scores to compare dynamic partitions with a ground truth longitudinally. Let's define the longitudinal partition as follows:

Definition 1 A **longitudinal partition** associates a label l to each set of tuples (n,t), with n a node and t a timestep.

Given a reference longitudinal partition L^{ref} and a longitudinal partition to compare L, any static community comparison function can be applied on those longitudinal partitions as with any partition. We apply AMI and ARI comparison function, hereafter called LAMI and LARI.

4.3.2 *Experimental settings* We fix parameters as $m = 10$, $s^{\min} = 5$, $s^{\max} = 15$, $o = 20$. We define an initially sharp partition with $\alpha = 1$ and $\beta = 0$. We make the sharpness vary by using a parameter μ such as $\alpha = 1 - \mu$ and $\beta = \mu$. We also add random noise $\beta_r = 0.01$. We repeat each experiment 20 times. On average, the resulting networks have 100 nodes and 1200 steps (snapshots).

4.3.3 *Results* Figure 7 synthesizes the results by scoring method, while Fig. 8 synthesizes the results by method, for a value of $\mu = 0.2$, which seems to be a tipping point. We make the following observations:

- In terms of instantaneous quality ($\overline{AMI}$, $\overline{ARI}$, $\overline{Q}$), as expected the No-Smoothing approach has the highest values, together with the Transversal-Network, for all μ. We can note that $\overline{AMI}$ starts at 1 (for $\mu = 0$) and do not fall below 0.3 for any method, showing that the community structure, although blurred, stays roughly detectable.
- In terms of smoothness, two methods have high scores for the three aspects: Implicit-Global and Smoothed-Graph. Label-Smoothing has the highest scores in most settings for the $SM - L$ scores, which measure label smoothness. DYNAMO is the least stable in most cases.
- No method seems to be a clear winner in terms of Longitudinal similarity with the ground truth. We can note however that for sharp communities ($\mu < 0.15$), the No-Smoothing approach always obtains the highest score, while for higher μ, Smoothed-Graph and in some cases Label-Smoothing obtain the highest scores. This is coherent with the observation that static algorithms become unstable when the community structure is not unambiguously defined, and that smoothing is therefore needed to obtain stable dynamic communities.

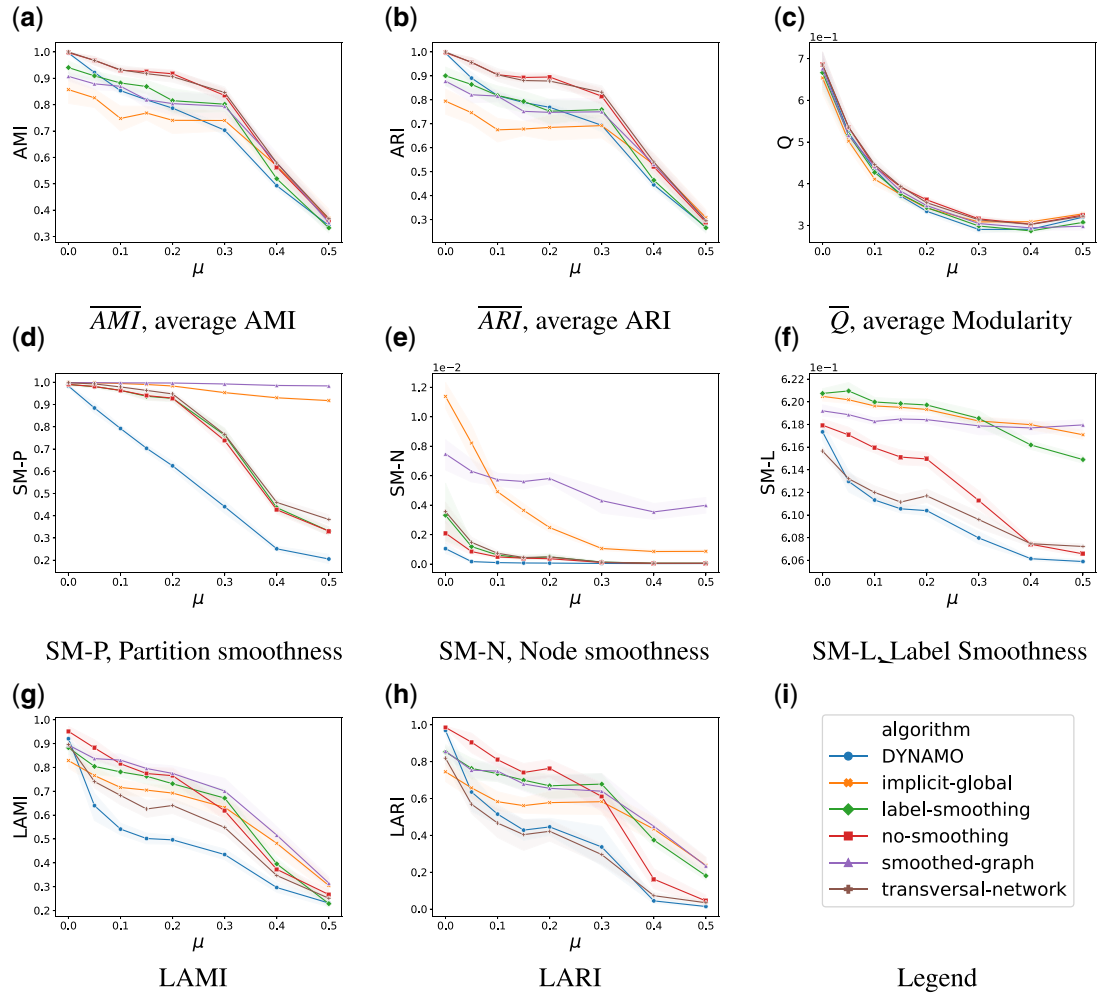


FIG. 7. Change in dynamic partitions properties when the μ parameter increases (higher score is better). Top row: Instantaneous Quality functions. Middle row: Smoothness quality functions. Bottom row: Longitudinal Quality functions. We observe clear differences between algorithms: no-smoothing and transversal network have among the highest scores in instantaneous Quality functions, but among the lowest in smoothness ones. Smoothed graphs and implicit global have the opposite behaviour. Despite those opposite approaches, no-smoothing and smoothed graph both have high scores in longitudinal scores.

From Fig. 8, we can make the following additional observations: Smoothed-Graph and Implicit-Global provide the strongest smoothing, but, as a consequence, compared with the reference No-Smoothing method, they have communities of lower quality in each individual snapshot.

4.4 *Scalability evaluation*

Another important aspect to consider in comparing methods is their capacity to handle large networks with many steps of evolution. The complexity of the No-Smoothing approach, for instance, is simple to

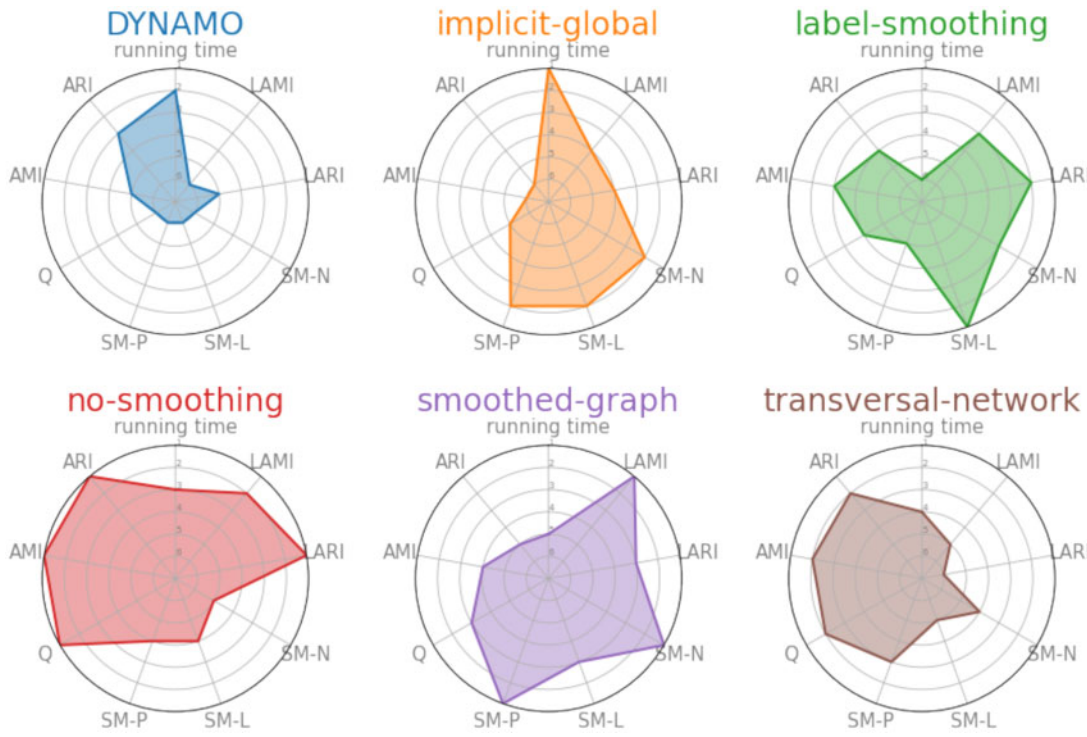


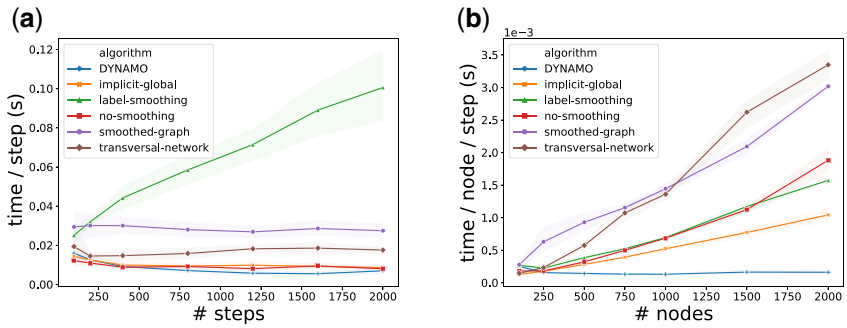
FIG. 8. Radar chart of ranks of different methods for $\mu = 0.2$. Higher score is better. We observe that the No-Smoothing method obtains among the best results in instantaneous scores (Q,AMI,ARI), while Smoothed-Graph and Implicit-Global obtain higher scores in smoothing scores. Label-Smoothing has the highest score in label smoothness.

estimate. It can be defined as $TO^{CD} + (T-1)O^{M}$, with T the number of steps, O^{CD} the complexity to run the static community detection algorithm at each step, O^{M} the complexity of the matching process between consecutive partitions. The first part, which is the most costly, can be trivially parallelized. The computational complexity is therefore linearly proportional to the number of steps and depends on the size of the network at each step and the chosen static algorithm. Other algorithms have complexities that depend on other factors and are harder to formulate theoretically in comparable terms.

In this section, we compare empirically the scalability of the chosen methods by varying two parameters: either we fix the size of the network at each step, and vary the number of steps, or the contrary. More formally, we use the same algorithm to generate random dynamic graphs as before, but:

- In the first test, we change the number of operations $o = 50$ in order to have a large number of steps, and run computations on subsets of the first x steps, varying x.
- In the second test, we vary the number of initial communities m while keeping the same average size. We run algorithms on a slice composed of the first 50 steps only.

In Fig. 9, we can observe that DYNAMO is the fastest method when the number of nodes or steps becomes large. This result is expected since the method is the only one among those tested that make only *local* changes according to modifications between steps [33].



Scalability for different number of steps, for a same size of graphs (n=50)

Scalability for different size of graphs, for a same duration (50 steps)

FIG. 9. Running time (in seconds) of the different algorithms depending on the average size of graphs and their number of steps. DYNAMO is by far the fastest method, thanks to its incremental approach. We observe that label smoothing is the only approach whose running time at each step grows with the number of steps.

Most of the tested methods have a complexity that is linear with the number of steps when the graph size is fixed. Label-Smoothing is the exception with complexity increasing quickly with the number of steps, because the similarity of communities in every step needs to be computed with communities in all other steps.

When the size of networks increase, all methods but DYNAMO have a complexity increasing faster than the number of nodes, which is expected since even the Louvain algorithm has a complexity superior to $O(n)$.

In particular, Transversal-Network and Smoothed-Graph approaches quickly become prohibitive. A possible explanation is that both methods work with dense matrices, due to the computation of an intermediate network representation in which a dense matrix is subtracted to the adjacency matrix. Optimized implementations could partly solve those problems.

Obviously, those results are highly dependent on the implementation of each method, and on the number of available cores for parallelization. For DYNAMO and transversal-network, we used implementations provided by the authors, respectively in Java for the first, and Matlab for the other. For the other methods, we used our own implementation in python, relying on the networkx [34], CDlib [35] and sklearn [36] libraries. We run the code on a 4 cores, 16GB of RAM computer.

From those observations, we can conclude that all tested algorithms but DYNAMO are able to handle small graphs with a few thousand steps, or larger graphs with a few hundred steps, but are not designed to handle large graphs with many steps of evolution, due to the handling of dynamic graphs as a succession of snapshots.

5. Conclusion

We have proposed a benchmark to generate dynamic graphs following any community evolution scenario using an appropriate language, and used it to compare several algorithms. By using examples and quantitative analysis, we have shown the weaknesses and strengths of several approaches, in particular the effect of smoothing on the quality of dynamic communities.

A limitation of the current implementation of the benchmark is the space complexity induced by the latent affinity Ω, which is in n^2, thus not practical for large graphs.

Variants of the method could be introduced to test different types of dynamic networks: one could test the influence of taking snapshots at coarser temporal granularity using sliding windows, or generate link streams by associating a spawning probability to edges of the currently generated dynamic graphs. Currently, generated networks are non-oriented and unweighted, but the benchmark could be trivially extended to generate such graphs.

Although we proposed scores specifically designed to evaluate dynamic partitions, the question of which score to use remains an important research question. In this work, we did not take into account the events themselves (*split*, *merge*, etc.) in evaluation scores, and based our longitudinal scores on labels only. We think that this approach is not fully satisfying and a proper way to take into account both the stability of communities and their similarity with a ground truth defined at each step should be investigated further.

Funding

This work is partially supported by BITUNAM Project ANR-18-CE23-0004 and by the European Community's H2020 Program under the funding scheme 'INFRAIA-01-2018-2019: Research and Innovation action', Grant Agreement n. 871042 'SoBigData++: European Integrated Infrastructure for Social Mining and Big Data Analytics'.

References

1. Lancichinetti, A., Fortunato, S. & Radicchi, F. (2008) Benchmark graphs for testing community detection algorithms. *Phys. Rev. E*, **78**, 046110.
2. Bazzi, M., Jeub, L. G., Arenas, A., Howison, S. D. & Porter, M. A. (2016) Generative benchmark models for mesoscale structure in multilayer networks. *Physical Review Research*, **2**, 023100
3. Granell, C., Darst, R. K., Arenas, A., Fortunato, S. & Gómez, S. (2015) Benchmark model to assess community structure in evolving networks. *Phys. Rev. E*, **92**, 012805.
4. Rossetti, G. (2017) RDYN: graph benchmark handling community dynamics. *J. Complex Netw.*, **5**, 893–912.
5. Sengupta, N., Hamann, M. & Wagner, D. (2017) Benchmark generator for dynamic overlapping communities in networks. *2017 IEEE International Conference on Data Mining (ICDM)* IEEE, pp. 415–424.
6. Perc, M. & Szolnoki, A. (2010). Coevolutionary gamesâŁ"a mini review. *BioSystems*, **99**, 109–125.
7. Genois, M. & Barrat, A. (2018) Can co-location be used as a proxy for face-to-face contacts? *EPJ Data Sci.*, **7**, 11.
8. Chykhradze, K., Korshunov, A., Buzun, N., Pastukhov, R., Kuzyurin, N., Turdakov, D. & Kim, H. (2014) Distributed generation of billion-node social graphs with overlapping community structure. *Complex Networks V*. Springer, pp. 199–208.
9. Benyahia, O., Largeron, C., Jeudy, B. & Zaïane, O. R. (2016) Dancer: dynamic attributed network with community structure generator. *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*. Springer, pp. 41–44.
10. Ghasemian, A., Zhang, P., Clauset, A., Moore, C. & Peel, L. (2016) Detectability thresholds and optimal algorithms for community structure in dynamic networks. *Phys. Rev. X*, **6**, 031005.
11. Greene, D., Doyle, D. & Cunningham, P. (2010) Tracking the evolution of communities in dynamic social networks. *2010 International Conference on Advances in Social Networks Analysis and Mining (ASONAM)*. IEEE, pp. 176–183.
12. Lin, Y.-R., Chi, Y., Zhu, S., Sundaram, H. & Tseng, B. L. (2008) Facetnet: a framework for analyzing communities and their evolutions in dynamic networks. *Proceedings of the 17th international conference on World Wide Web*. ACM, pp. 685–694.
13. Sarzynska, M., Leicht, E. A., Chowell, G. & Porter, M. A. (2015) Null models for community detection in spatially embedded, temporal networks. *J. Complex Netw.*, **4**, 363–406.

14. Tantipathananandh, C. & Berger-Wolf, T. Y. (2011) Finding communities in dynamic social networks. *2011 IEEE 11th International Conference on Data Mining (ICDM)*. IEEE, pp. 1236–1241.

15. Xu, K. S. & Hero, A. O. (2014) Dynamic stochastic blockmodels for time-evolving social networks. *IEEE J. Select. Top. Signal Process.*, **8**, 552–562.

16. Zhang, X., Moore, C. & Newman, M. E. (2017) Random graph models for dynamic networks. *Eur. Phys. J. B*.

17. Coppens, L., De Venter, J., Mitrovi, S. & De Weerdt, J. (2019) A comparative study of community detection techniques for large evolving graphs. *LEG@ ECML: The third International Workshop on Advances in Managing and Mining Large Evolving Graphs collocated with ECML-PKDD*. Springer.

18. Holland, P. W., Laskey, K. B. & Leinhardt, S. (1983) Stochastic blockmodels: first steps. *Soc. Netw.*, **5**, 109–137.

19. Leskovec, J., Kleinberg, J. & Faloutsos, C. (2005) Graphs over time: densification laws, shrinking diameters and possible explanations. *Proceedings of the Eleventh ACM SIGKDD International Conference on Knowledge Discovery in Data Mining*. pp. 177–187.

20. Leskovec, J. & Krevl, A. (2014) SNAP datasets: Stanford Large Network Dataset collection. http://snap.stanford.edu/data.

21. Blondel, V. D., Guillaume, J.-L., Lambiotte, R. & Lefebvre, E. (2008) Fast unfolding of communities in large networks. *J. Stat. Mech.*, P10008.

22. Kobayashi, T., Takaguchi, T. & Barrat, A. (2019) The structured backbone of temporal social ties. *Nat. Commun.*, **10**, 1–11.

23. Rossetti, G. & Cazabet, R. (2018) Community discovery in dynamic networks: a survey. *ACM Comput. Surv. (CSUR)*, **51**, 1–37.

24. Li, H. J., Wang, L., Zhang, Y., & Perc M. (2020). Optimization of identifiability for efficient community detection. *N. J. Phys.,* 22.

25. Folino, F. & Pizzuti, C. (2013) An evolutionary multiobjective approach for community discovery in dynamic networks. *IEEE Trans. Knowl. Data Eng.*, **26**, 1838–1852.

26. Kawadia, V. & Sreenivasan, S. (2012) Sequential detection of temporal communities by estrangement confinement. *Sci. Rep.*, **2**, 794.

27. Aynaud, T. & Guillaume, J.-L. (2010) Static community detection algorithms for evolving networks. *8th International Symposium on Modeling and Optimization in Mobile, Ad Hoc, and Wireless Networks.* IEEE, pp. 513–519.

28. Zhuang, D., Chang, M. J., & Li, M. (2019). DynaMo: dynamic community detection by incrementally maximizing modularity. *IEEE Trans. Knowl. Data Eng.*

29. Guo, C., Wang, J. & Zhang, Z. (2014) Evolutionary community structure discovery in dynamic weighted networks. *Physica A*, **413**, 565–576.

30. 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**, 876–878.

31. Falkowski, T. & Spiliopoulou, M. (2007) Data mining for community dynamics. *KI*, **21**, 23–29.

32. Linhares, C. D., Travençolo, B. A., Paiva, J. G. S. & Rocha, L. E. (2017) DyNetVis: a system for visualization of dynamic networks. *Proceedings of the Symposium on Applied Computing*. ACM, pp. 187–194.

33. Cazabet, R. & Rossetti, G. (2019) Challenges in community discovery on temporal networks. *Temporal Network Theory*. Springer, pp. 181–197.

34. Hagberg, A., Swart, P. & S Chult, D. (2008) Exploring network structure, dynamics, and function using NetworkX. Technical Report, Los Alamos National Lab.(LANL), Los Alamos, NM (United States).

35. Rossetti, G., Milli, L. & Cazabet, R. (2019) CDLIB: a python library to extract, compare and evaluate communities from complex networks. *Appl. Netw. Sci.*, **4**, 52.

36. Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., Vanderplas, J., Passos, A., Cournapeau, D., Brucher, M., Perrot, M. & Duchesnay, E. (2011) Scikit-learn: machine learning in Python. *J. Mach. Learn. Res.*, **12**, 2825–2830.

A. Appendix: Definition of community events

In this appendix, we give the implementation of several well-known community events. Note that these events, and a few more, are implemented in the library released with this article.

BIRTH(NB-NODES, LABEL) is an instruction to create a new community with a label LABEL composed of a number of NB-NODES newly created nodes. Note that a new community could appear by taking nodes from one or several existing communities, but this should be represented through an ad-hoc ASSIGN event.

Input: NB-NODES, LABEL
begin
  N ← [];
  for *i* ← *0* **to** *NB-NODES* **do**
    N ← N ∪ NEW-NODE() ;
  end
  return ASSIGN([],[N],[LABEL]) ;
end

Algorithm 2: instruction **BIRTH**

DEATH(COM) is an instruction to make the community COM disappear, and its nodes leave the network.

Input: *COM*
begin
  ASSIGN([COM],[],[]) ;
end

Algorithm 3: instruction **DEATH**

MERGE(C-BEFORE, L-AFTER) is used to merge two or more communities into a single one. It is an instruction with two parameters:

- C-BEFORE the list of communities to merge
- L-AFTER the label of the resulting community (if not provided, a new unique label is generated)

The label in L-AFTER can be either one of those of the communities in C-BEFORE or a new one (conservation or not of the label by one of the resulting communities). It is defined by the following algorithm:

Input: C-BEFORE, L-AFTER
begin
  N ← [];
  for *C* ∈ *C-BEFORE* **do**
    N ← N ∪ C.N ;
  end
  return ASSIGN(C-BEFORE,[N],[L-AFTER]) ;
end

Algorithm 4: instruction **MERGE**

SPLIT(C-BEFORE, L-AFTER, SIZES) is used to split a community by creating new ones of sizes described in the SIZE parameter. Nodes are chosen randomly. It can be defined as a command with three parameters:

- C-BEFORE: the community to split
- L-AFTER: the ordered list of labels of the communities created by the split event
- SIZES: an ordered list containing the sizes of the resulting communities. The sum of the elements of this list must be equal to the number of nodes in C-BEFORE

It is defined by the following algorithm:

Input: C-BEFORE, SIZES, L-AFTER
begin
  N ← C-BEFORE.N ;
  AFTER-NODES ← [] ;
  for *i* ← *0* **to length***(SIZES)* **do**
    AFTER-NODES[i] ← **randomChoice**(N, SIZES[i]) ;
    N ← N \ AFTER-NODES[i] ;
  end
  return ASSIGN([C-BEFORE], AFTER-NODES, L-AFTER) ;
end

Algorithm 5: instruction **SPLIT**. randomChoice(L,n) function select randomly n elements in list L

THESEUS(COM, NB-NODES) COM is the community to modify. NB-NODES corresponds to the number of nodes to replace in the original community. If NB-NODES is equal to the number of nodes, then the event corresponds exactly to the scenario described in Fig. 1.

Note that we use the ASSIGN instruction to describe, at each step, that the community simultaneously lose a node and gain a new one, and then at the end to make the community reappear with the original nodes.

Input: COM, NB-NODES
begin
  N-CURRENT ← copy(COM.N);
  N-ORIGINAL ← copy(COM.N) ;
  COM-CURRENT ← COM ;
  for *i* ← *0* **to** *NB-NODES* **do**
    REMOVED ← **randomChoice**(N-CURRENT,1) ;
    N-CURRENT ← N-CURRENT \ REMOVED ;
    ADDED ← NEW-NODE() ;
    [COM-CURRENT] ← ASSIGN([COM-CURRENT], [(COM-CURRENT.N ∪ ADDED) \ REMOVED], [COM-CURRENT.L]) ;
  end
  B ← ASSIGN([], [N-ORIGINAL], [NEW-COM-ID()]) ;
  return [COM-CURRENT, B]
end

Algorithm 6: instruction **THESEUS**. copy function allows to make a copy of a list, such as it can be modified without affecting the original

INITIALIZE(SIZES, LABELS) allows setting the community structure in the first step.

- SIZES is the list of sizes of initial communities
- LABELS is the list of labels of those communities

The function returns the corresponding communities composed of newly created nodes. The code is omitted for simplicity, but present in the provided implementation.