

DANCer: dynamic attributed networks with community structure generation

C. Llargeron¹ · P. N. Mougel¹ ·
O. Benyahia¹ · O. R. Zaiane² 

Received: 10 February 2016 / Revised: 24 September 2016 / Accepted: 3 February 2017 /
Published online: 2 March 2017
© Springer-Verlag London 2017

Abstract Most networks, such as those generated from social media, tend to evolve gradually with frequent changes in the activity and the interactions of their participants. Furthermore, the communities inside the network can grow, shrink, merge, or split, and the entities can move from one community to another. The aim of community detection methods is precisely to detect the evolution of these communities. However, evaluating these algorithms requires tests on real or artificial networks with verifiable ground truth. Dynamic networks generators have been recently proposed for this task, but most of them consider only the structure of the network, disregarding the characteristics of the nodes. In this paper, we propose a new generator for dynamic attributed networks with community structure that follow the properties of real-world networks. The evolution of the network is performed using two kinds of operations: Micro-operations are applied on the edges and vertices, while macro-operations on the communities. Moreover, the properties of real-world networks such as preferential attachment or homophily are preserved during the evolution of the network, as confirmed by our experiments.

Keywords Social network · Graph generator · Community structure

1 Introduction

Recently, significant attention has been paid to the emerging field of temporal network analysis. The study of these kinds of complex networks is interdisciplinary, leading to terminology where a single concept can have different names: temporal graph, evolving graph, evolutionary network, time-varying graph, time-aggregated graph, time-stamped graph, dynamic

✉ C. Llargeron
christine.llargeron@univ-st-etienne.fr

¹ Univ Lyon, UJM-Saint-Etienne, CNRS, Institut d'Optique Graduate school, Laboratoire Hubert Curien UMR 5516, F-42023 SAINT-ETIENNE, France

² Department of Computer Science, University of Alberta, Edmonton, Canada

network, dynamic graph, and so on. The main challenge with these forms of networks is to track the changes in their configuration or in their community structure over time that might help anticipate important transformations in the whole system. To this end, new approaches are proposed in the literature to analyze the topological and temporal structure of networks and to model their behavior over time. Most of these methods utilize not only the relationships between the entities but also the attributes describing their characteristics. However, the current lack of benchmarks makes the evaluation of the results and comparison of the methods difficult. To check whether a method can capture changes in a network, in particular the community evolution, suitable benchmarks are needed. To fill this gap, we have designed a generator for static attributed networks with community structure [20]. This generator allows to generate networks having communities and attributed nodes. Moreover, properties of real-world networks are respected by the generated network, like, for instance, the preferential attachment which requires that the more connected nodes are more likely to receive new links, or the homophily property according to which nodes are more likely to connect to nodes having similar attributes. We propose herein an extended version of this generator for dynamic attributed networks with community structure, called DANCer. The objective is to provide a sequence of attributed graphs with a well-defined partition of the vertices into communities that progresses from one-time snapshot to the other while preserving the properties of real networks. Thus, DANCer allows a researcher to generate a first network having certain properties and then produce a sequence of other ones by modifying one or several parameters to weaken the properties of the first network. The researcher can study the robustness of pattern mining methods toward the degradation of this reference network.

The model underlying this generator is presented in the following section Sect. 2 and the algorithm of DANCer in Sect. 3. The experiments are described in Sect. 4 and the related work in Sect. 5. Section 6 states our conclusion.

2 Model

We provide a new generator of attributed dynamic networks with community structure. Such networks can be represented by (1) a sequence of T attributed graphs $\mathcal{G}_i = (\mathcal{V}_i, \mathcal{E}_i)$, $i \in \{1, \dots, T\}$, where \mathcal{V}_i is a set of vertices, \mathcal{E}_i a set of undirected edges and where for each vertex $v \in \mathcal{V}_i$ and each attribute $A \in \mathcal{A}$, v_A denotes the attribute value of A assigned to vertex v and (2) a sequence of T partitions \mathcal{P}_i of \mathcal{V}_i , $i \in \{1, \dots, T\}$ which gives a community for each vertex in the corresponding graph \mathcal{G}_i , $i \in \{1, \dots, T\}$.

The generation of the network is carried out in two phases. In phase one, an initial graph $\mathcal{G}_1 = (\mathcal{V}_1, \mathcal{E}_1)$ is built while respecting the well-known network properties such as preferential attachment or homophily. In the second phase, the initial graph is modified through two kinds of operations. The first set of operations, called micro-operations, because they are local, are applied on the vertices and edges, whereas the second kind of operations are applied on the communities, i.e., at a macro-level. In the next subsections, we present the assumptions underlying the generation of the graph, the properties of the latter and the dynamic operations allowing us to build a sequence of graphs.

2.1 Hypothesis and properties of a graph

The proposed generator is based on the following hypothesis concerning the properties of a graph in the sequence.

We consider that a graph (i.e., the network at a single timestamp) has a community structure if the nodes are grouped into sets densely connected, while they are less connected to vertices belonging to other groups. Moreover, a graph has a cluster structure if the nodes are grouped into sets relatively homogeneous with regard to the attributes. Usually, the groups defined according to the relationships are called communities, whereas those defined according to the attributes are called clusters, but, in our article, we assume that the structure defined on a graph is based on the attributes and the relationships together. So the membership to a group (called indifferently cluster or community) depends on the attributes and structural links in such a way that firstly, there should be more edges between vertices belonging to the same community than between vertices from different communities and secondly, two vertices belonging to the same community are more likely to be similar in terms of attributes than two vertices belonging to different communities. Moreover, we suppose that these groups do not overlap, and consequently, they define a set of partitions, one for each graph: \mathcal{P}_i of \mathcal{V}_i $i \in \{1, \dots, T\}$, such that (1) $\forall (C_1, C_2) \in \mathcal{P}_i \times \mathcal{P}_i, C_1 \cap C_2 = \emptyset$; (2) $\forall C \in \mathcal{P}_i, C \neq \emptyset$; and (3) $\cup_{C \in \mathcal{P}_i} C = \mathcal{V}_i$.

Note that this structure, based on the attributes and relationships, is compatible with the homophily hypothesis, according to which two vertices are more likely to be connected if they share common characteristics, and this property is verified inside the communities but also between communities [21,26]. So, the more similar the vertices, the more likely they are to be connected.

Finally, our model respects the preferential attachment according to which new nodes prefer to join the more connected nodes existing in the network. Thus, each node is connected to an existing node with a probability proportional to the number of links of the chosen node. Given this hypothesis, our model can be considered as an extension of the Barabási–Albert model: It leads to scale-free networks characterized by a degree distribution with a heavy tail, which can be approximated by a power law distribution such that the fraction of vertices $P(k)$ having a degree k follows the law $P(k) \sim k^{-\gamma}$ where γ ranges typically between 2 and 3 [3]. However, as noted in [22], usually in social networks, the actors do not have a global knowledge of the network. Consequently, the preferential attachment model is more likely to be local.

Given these hypotheses, the proposed model allows us to generate attributed graphs having the following properties.

P1. Local preferential attachment: The local preferential attachment states that a vertex is more likely to create connections with vertices having a high degree and which are close [22].

P2. Small world: This property indicates that most vertices can be reached from every other through a small number of edges. According to [32], in a small-world network, the average shortest path is proportional to the logarithm of the number of vertices. The diameter can also be used to evaluate the small-world property since it is defined as the maximum distance between any two vertices, where the distance is the minimum number of edges on the path from one vertex to the other. It has been shown that real networks exhibit very small diameters, notably the well-known “six-degrees of separation” [4,27].

P3. Community structure: A community structure appears when vertices can be grouped in such a way that vertices in a group are more connected to vertices in the same group compared to other vertices. While there is no formal definition of a network community, several measures have been proposed to control the community structure. In this article, we

consider the well-known modularity measure from [28]. Moreover, the average clustering coefficient from [32] is given as an indication of the transitivity of connections in the network.

P4. Homogeneity: The groups, which compose a partition of the vertices, are homogeneous if two vertices belonging to the same group are more likely to be similar in terms of attributes than two vertices belonging to different groups. To measure this property, one can use the within inertia ratio. Given \mathcal{P} , a partition of vertices and $d(v_1, v_2)$, the Euclidean distance between the real attribute vectors assigned to the vertices v_1 and v_2 , the within inertia ratio is $\frac{\sum_{C \in \mathcal{P}} (P_C \sum_{v \in C} d(v, g_C)^2)}{\sum_{v \in \mathcal{V}} d(v, g)^2}$ where g_C is the center of gravity of the vertices in C (i.e., its centroid), P_C the weight of C and g is the global center of gravity of all the vertices.

P5. Homophily: While the homogeneity is only defined according to the attribute values, the homophily is defined according to relationships and attribute values. Indeed, this property is verified when similar vertices according to their attributes tend to be more connected than dissimilar vertices. To measure this property, we adapted the test introduced by [9] for numeric attributes. This test compares an *expected homophily measure* corresponding to the probability for two vertices to be similar with an *observed homophily measure* defined as the probability that two linked vertices are similar. If the expected measure is significantly less than the observed measure, then there is evidence for homophily.

Given an initial graph having these properties, our aim is to simulate its evolution. In the following section, we present the methodological choices for modeling common dynamic operations with the objective of conserving the mentioned properties in the graph at each timestamp.

2.2 Dynamic operations

The evolution of the network is performed using two kinds of operations: micro (or local)-operations and macro-operations. The micro-operations are applied on the vertices or edges, while the macro-operations are applied on the communities. A list of these operations is presented in Table 1.

2.2.1 Micro-operations

The micro-operations consist in removing or adding vertices and edges or updating their attributes. They are illustrated through examples in Figs. 1, 2, 3, 4, 5 and 6.

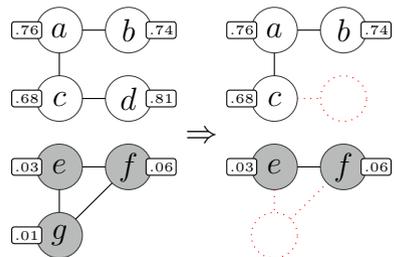
Remove vertex: The first micro-operation consists in removing a vertex. This is presented in Algorithm 5 described in Sect. 3. Intuitively, a vertex connected to many other vertices is unlikely to leave the network. To model this behavior, we consider that only vertices connected to a single other vertex (i.e., having a degree equal to one) or in a triangle may leave the network. This constraint ensures that removing a vertex will not split a community into several disconnected components. The operation is illustrated in Fig. 1. Note that in the example, the vertices a or c cannot be removed because their removal disconnects their community, whereas it is not the case for d and g . Indeed, even though the degree of vertex g is two, since it is part of a triangle, it can be removed.

Add vertex: The operation, adding a vertex, allows new vertices to be inserted in a community. This is presented in Algorithm 6 described in Sect. 3. According to the homophily property **P5**, the attribute values of the new vertex are selected from the attribute values of

Table 1 Description of the operations in the context of a social network

Operations	Description
<i>Micro-operations</i>	
Add vertex	A new actor joins the network in an existing community
Remove vertex	An actor leaves the network
Add within edge	A connection is created between two actors belonging to the same community
Remove within edge	A connection is removed between two actors belonging to the same community
Add between edge	A connection is created between two actors in distinct communities, i.e., creation of a bridge between two communities
Remove between edge	A connection is removed between two actors in distinct communities
Update attributes	An actor has some of its properties changed according to the social influence effect
<i>Macro-operations</i>	
Split community	An existing community is split into two new communities, and connections are removed between distant members of the original community
Merge communities	Two similar communities are merged into a single one, and new connections are created between the members of the new community
Migrate community	A group of actors leave their community to either create a new one or join an existing one

Fig. 1 Examples illustrating the micro-operation Remove vertex (removing vertices *d* and *g*). The gray scales correspond to the communities. Vertices are assigned a single attribute



the vertices in its community. More precisely, the value for each attribute is picked uniformly and randomly between the minimum and the maximum observed for the vertices in the community. New connections are also created with vertices inside its community using the preferential attachment property **P1** (i.e., vertices having a high degree are more likely to be selected as neighbors) and with vertices in other communities with a preference for vertices having similar attributes according to **P5**. In the example presented in Fig. 2, the new vertex *b* is connected to vertex *c* since it has the highest degree.

Update attributes: The attribute update allows the social influence effect to be modeled. This is presented in Algorithm 7 described in Sect. 3. It is well known that not only actors tend to connect to similar actors (i.e., the homophily effect), but also that an actor may become more similar to actors with which it is connected. This is commonly referred to as the social influence effect. For this reason, in the proposed model, a vertex may have one of its attributes replaced by a new value, uniformly and randomly selected between the minimum and the maximum observed in its neighborhood inside its community.

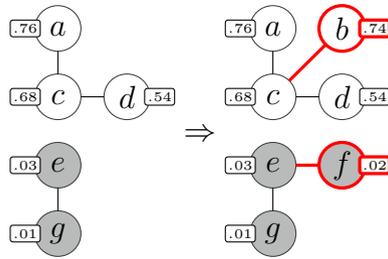
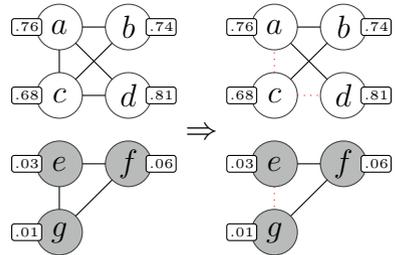


Fig. 2 Examples illustrating the micro-operation Add vertex (adding vertices *b* and *f*). The *gray scales* correspond to the communities. Vertices are assigned a single attribute

Fig. 3 Examples illustrating the micro-operation remove within edges (removing edges *a-c*, *c-d* and *e-g*). The *gray scales* correspond to the communities. Vertices are assigned a single attribute



Four operations are proposed for the edges because we choose to distinguish between edges within a community (i.e., connecting two vertices in the same community) and edges between communities (i.e., connecting two vertices belonging to different communities). This difference seems relevant because a within edge corresponds to a connection that could be qualified as natural (e.g., family, work, hobby) and often found in the network. On the other hand, a between edge corresponds to a bridge between communities which are by definition loosely connected. The between edges play a specific role as they allow communication over the whole network.

Remove within edge: The operation consisting of removing within edges allows a network with sparse communities to be generated. Algorithm 8 presented in Sect. 3 describes this operation, and an example is given in Fig. 3. Since this operation is intended to be local, we added the constraint that removing an edge should not split a community into several disconnected components, which could be considered as updating the set of communities. Consequently, an edge between two vertices may be removed only if these two vertices are connected through another member of their community.

Add within edge: As an opposite operation to the previous, this operation enforces the community structure by connecting vertices within a community according to the preferential attachment property P1. This is presented in Algorithm 9 and illustrated in Fig. 4.

Remove between edge: The between edges are removed by selecting vertices, which have dissimilar attribute values. This operation detailed in Algorithm 10 allows the construction of highly disconnected communities. In particular, it is possible to build a network where each community is a connected component. Note that removing such an edge may split a

Fig. 4 Examples illustrating the micro-operation add within edges (adding edges $b-d$ and $f-g$). The gray scales correspond to the communities. Vertices are assigned a single attribute

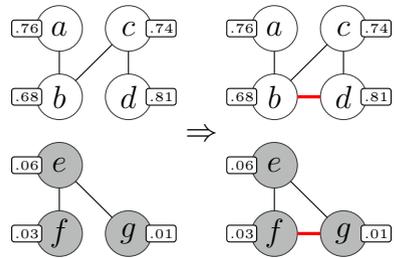


Fig. 5 Examples illustrating the micro-operation Remove between edges (removing edge $b-e$). The gray scales correspond to the communities. Vertices are assigned a single attribute

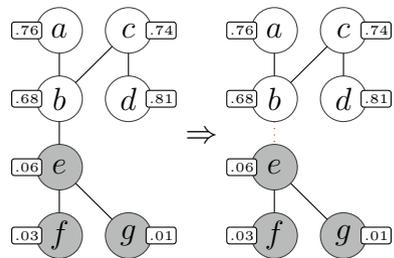
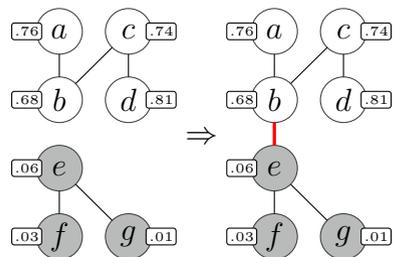


Fig. 6 Examples illustrating the micro-operation Add between edges (adding edge $b-e$). The gray scales correspond to the communities. Vertices are assigned a single attribute



connected component formed by at least two communities. Nonetheless, it is not considered as a macro-operation since it does not affect community membership (Fig. 5).

Add between edge: The operation of adding between edges, presented in Algorithm 11, allows two existing communities to be connected or more connected by linking two vertices belonging to each one. The first vertex must have a higher number of connections inside its community than outside. The second vertex is then selected among the most similar vertices with respect to the attribute values (Fig. 6).

2.2.2 Macro-operations

The macro-operations are applied to the communities, and consequently they may drastically change the network structure. They consist in (1) migrating members of a community to either a new community or an existing one, (2) splitting a community into two new subcommunities and (3) merging two existing communities into a single community. They are performed after the micro-operations in Algorithm 4 (lines 10–19).

The migration of actors, presented in Algorithm 12, allows either the creation of a new community or the migration of several actors from the community to another. To decide which vertices are migrating, we consider a randomly selected vertex and its neighborhood which satisfy constraints with respect to their connectivity. The idea is that a migrating vertex

Fig. 7 Examples illustrating the split macro-operation. The *gray scales* correspond to the communities. Vertices are assigned a single attribute

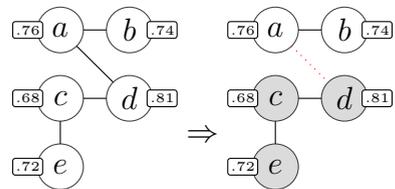
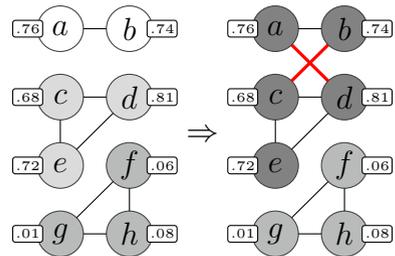


Fig. 8 Examples illustrating the merge macro-operation. The *gray scales* correspond to the communities. Vertices are assigned a single attribute



should be able to connect more easily with vertices in the destination community than with the vertices in its own community. More precisely, we compare the number of its neighbors in the destination community and in its original community using both the direct neighborhood and the 2-neighborhood (i.e., neighbors of the direct neighbors). As this operation may disconnect the community, a rewiring step is performed to restore the connectivity in the neighborhood of the migrating vertices.

To model a community split, presented in Algorithm 14, we first have to decide which community should be split and then find which vertices will join another community. The first hypothesis is that a community is more likely to be split if it is large in such a way that some members may hardly communicate inside the community. This intuition can be modeled by splitting the community having the highest diameter. The diameter measures eccentricity by checking the highest shortest path between pairs of nodes in the community. To reallocate the vertices, we consider that a split is due to the emergence of two leaders, corresponding to highly connected vertices that will be followed by the other members of the community depending on their geodesic proximity. Moreover, to remain consistent with the standard community definition, the connections between the members of the two new communities should be sparser than the connections within the new communities (Fig. 7).

For the merge operation, presented in Algorithm 13, we have to select communities which must be merged. This choice is done by considering, on the one hand, the attribute-based distance between the community members and on the other hand, their number of connections. Intuitively, two communities sharing similar interests and communicating easily are more likely to be merged. The distance between two communities is defined as the minimum of the pairwise Euclidean distance of their representative vertices. The merged communities are those having a distance below the average community distance and maximizing the number of between edges (Fig. 8).

Given this model and these operations, we present, in the following section, an algorithm taking into account the desired properties.

3 Algorithm

The following Generator named DANCer described as Algorithm 1 returns:

- an attributed dynamic network with community structure G defined by a sequence of T graphs $\mathcal{G}_i = (\mathcal{V}_i, \mathcal{E}_i), i \in \{1, \dots, T\}$, where \mathcal{V}_i is a set of vertices, \mathcal{E}_i a set of undirected edges and where for each vertex $v \in \mathcal{V}_i$ and each attribute $A \in \mathcal{A}$, v_A denotes the attribute value of A assigned to vertex v ;
- a sequence of T partitions \mathcal{P}_i of $\mathcal{V}_i, i \in \{1, \dots, T\}$ giving the community membership of the vertices for each graph \mathcal{G}_i such that:
 - $\forall (C_1, C_2) \in \mathcal{P}_i^2, C_1 \neq C_2, C_1 \cap C_2 = \emptyset$;
 - $\forall C \in \mathcal{P}_i, C \neq \emptyset$;
 - $\bigcup_{C \in \mathcal{P}_i} C = \mathcal{V}_i$

The previously described network properties can be controlled by the user with the parameters summarized in Table 2. Firstly, an initial graph $\mathcal{G}_1 = (\mathcal{V}_1, \mathcal{E}_1)$ and its corresponding partition \mathcal{P}_1 are generated by Algorithm 1 (lines 1–18). The reader is referred to [20] for a detailed discussion on this Algorithm and its parameters, notably the power law distribution of the function RandPI which ensures the local preferential attachment. From this first graph, several operations are then applied to obtain the other instances of the network (lines 22–24). For each new graph \mathcal{G}_i , two kinds of updates are performed by Algorithm 4, (Line 23 in Algorithm 1) either micro-updates (i.e., insert or remove vertices, within or between edges and update attribute values) or macro-updates (i.e., merging and splitting of communities or migrating vertices from one community to another). They are detailed in the following sections.

3.1 Micro-dynamic updates

The call to the micro-updates are performed on lines 1–2 of Algorithm 4. The micro-updates correspond to the following algorithms:

- RemoveVertices (Algorithm 5) and AddVertices (Algorithm 6) which, respectively, remove and insert vertices in the graph;
- UpdateAttributes (Algorithm 7) that updates the attribute values assigned to existing vertices;
- RemoveWithinEdges (Algorithm 8) and AddWithinEdges (Algorithm 9) which, respectively, removes and inserts edges between vertices in the same community;
- RemoveBetweenEdges (Algorithm 10) and AddBetweenEdges (Algorithm 11), respectively, remove and insert edges connecting vertices belonging to different communities.

Each micro-operation is performed with respect to an uniform probability P_{micro} . The higher this parameter, the more likely the micro-updates take place: If it is set to 0, then no micro-operation is performed, while if it is equal 1, the micro-operations are always run. The parameters $R_{removeVert}$, $R_{addVert}$, $R_{updatedAttributes}$, $R_{removeWthEdge}$, $R_{addWthEdge}$, $R_{removeBtwEdge}$, $R_{addBtwEdge}$ allow to quantify the number of objects (i.e., vertices or edges) that will be impacted by the corresponding micro-operation. The parameters are summarized in Table 2, and their role is described with the corresponding algorithms. Parameters for the first timestamp are explained in [20]. The micro-updates are performed independently for each community (line 1 in Algorithm 4), except for the operations on the between edges. The different algorithms are detailed below.

RemoveVertices: The vertex removal is presented in Algorithm 5. This operation is controlled by parameter $R_{removeVert}$, which corresponds to the ratio between the number of vertices removed and the candidates V_{IoAdd} defined as the vertices that can be removed from

Algorithm 1: Generator

Input: $N, E_{with}^{\max}, E_{btw}^{\max}, MTE, \mathcal{A}, K, NbRep, P_{randomCommunity}, T$
 // Micro-operations parameters
 $P_{micro}, R_{removeVert}, R_{addVert}, R_{removeWthEdge}, R_{addWthEdge}, R_{removeBtwEdge},$
 $R_{addBtwEdge}, R_{updatedAttributes}$
 // Macro-operations parameters
 $P_{merge}, P_{split}, P_{migrate}, P_{removeEdgeSplit}$
Output: - A sequence \mathcal{G} of T graphs $\mathcal{G}_i = (\mathcal{V}_i, \mathcal{E}_i, \mathcal{A}_i), i \in \{1, \dots, T\}$
 - A sequence \mathcal{P} of T partitions \mathcal{P}_i of $\mathcal{V}_i, i \in \{1, \dots, T\}$

- 1 $\mathcal{V} \leftarrow$ an arbitrary set of N vertices
 // Vertices and attributes generation
- 2 **for** $v \in \mathcal{V}$ **do**
- 3 // Generate numeric attributes for v
 for $A \in \mathcal{A}$ **do** $v_A \leftarrow \mathcal{N}(0, \sigma_A)$
- // Communities initialization
- 4 InitCommunities()
- 5 **for** $C \in \mathcal{P}$ **do** $C.rep \leftarrow C \cup \{v\}$
 // Batch vertex insertion
- 6 **while** $V_{toAdd} \neq \emptyset$ **do**
- 7 **for** $v \in Sample(V_{toAdd}, RandUni(\{1, \dots, |V_{toAdd}|\}))$ **do**
- 8 **if** $RandUni([0, 1]) < P_{randomCommunity}$ **then** $C \leftarrow RandUni(\mathcal{P})$ **else**
 $C \leftarrow \arg \min_{C' \in \mathcal{P}} dist(v, C'.rep)$
- 9 AddEdges(v, C)
- 10 $C \leftarrow C \cup \{v\}$
- 11 $V_{toAdd} \leftarrow V_{toAdd} \setminus \{v\}$
- 12 **for** $C \in \mathcal{P}$ **do** $C.rep \leftarrow Sample(C, \min(|C|, NbRep))$
- // Final edges insertion
- 13 $MTE \leftarrow \min(MTE, \sum_{C \in \mathcal{P}} \frac{|C| \times (|C| - 1)}{2})$
- 14 **while** $|\mathcal{E}| < MTE$ **do**
- 15 $v \leftarrow RandEdgeBtw(V)$
- 16 $E_{tri} \leftarrow \{\{v_1, v_2\} \mid v_1, v_2 \in neighwth(v) \wedge v_1 \neq v_2\} \setminus \mathcal{E}$
- 17 $\mathcal{E} \leftarrow \mathcal{E} \cup RandUni(E_{tri})$
- 18 **output** $\mathcal{G} = \{\mathcal{G}_1\}, \mathcal{P} = \{\mathcal{P}_1\}$
- 19 **repeat** $T - 1$ **times**
- 20 DynamicUpdates(\mathcal{G}, \mathcal{P})
- 21 **output** $\mathcal{G} = \{\mathcal{G}_1 \dots \mathcal{G}_T\}, \mathcal{P} = \{\mathcal{P}_1 \dots \mathcal{P}_T\}$

Algorithm 2: InitCommunities

- 1 $V_{init} \leftarrow Sample(\mathcal{V}, K \times NbRep)$
- 2 $\mathcal{P} \leftarrow K Medoids(V_{init}, K)$
- 3 $MinRep \leftarrow \min_{C \in \mathcal{P}} |C|$
- 4 **for** $C \in \mathcal{P}$ **do**
- 5 $g \leftarrow$ Center of gravity of the elements in C
- 6 $C \leftarrow \arg \min_{\substack{C' \subseteq C \\ |C'| = MinRep}} \sum_{v \in C'} d(v, g)$
- 7 **for** $v \in C$ **do**
- 8 $E_{with} \leftarrow RandUni(\{1, \dots, E_{with}^{\max}\})$
- 9 **repeat** E_{with} **times**
- 10 $v' \leftarrow RandUni(C \setminus \{v\})$
- 11 $\mathcal{E} \leftarrow \mathcal{E} \cup \{\{v, v'\}\}$

Table 2 Description of the dynamic network generator parameters

Parameter	Dom.	Algorithm	Description
First timestamp (i.e., graph G_1)			
K	\mathbb{N}^+	1	Number of communities
N	\mathbb{N}^+	1	Number of vertices
p	\mathbb{N}^+	1	Number of numerical attributes
$\mathcal{A} = \{\sigma_1, \dots, \sigma_p\}$		1	Standard deviations of the attributes generated using centered normal distributions
E_{wth}^{\max}	\mathbb{N}	1	Maximum number of edges connecting a new vertex to vertices in its community
E_{btw}^{\max}	$\{0, \dots, E_{wth}^{\max}\}$	1	Maximum number of edges connecting a new vertex to vertices in a different community
$NbRep$	\mathbb{N}^+	1	Maximum number of representatives of each community
MTE	\mathbb{N}	1	Minimum number of total edges
$P_{randomCommunity}$	$[0, 1]$	1	A threshold to decide whether a new vertex joins a randomly selected community or not
Micro-operations			
P_{micro}	$[0, 1]$	4	A threshold to select whether the micro-dynamic updates are performed or not
$R_{removeVert}$	$[0, 1]$	5	Ratio defining the number of vertices removed
$R_{addVert}$	$[0, 1]$	6	Ratio defining the number of vertices inserted
$R_{updatedAttributes}$	$[0, 1]$	7	Ratio defining the number of attributes updated
$R_{removeWthEdge}$	$[0, 1]$	8	Ratio defining the number of within edges removed
$R_{addWthEdge}$	$[0, 1]$	9	Ratio defining the number of within edges inserted
$R_{removeBtwEdge}$	$[0, 1]$	10	Ratio defining the number of between edges removed
$R_{addBtwEdge}$	$[0, 1]$	11	Ratio defining the number of between edges inserted
Macro-operations			
$P_{removeEdgeSplit}$	$[0, 1]$	14	Proba. to remove an edge between two vertices in the previously same community when splitting a community
$P_{migrate}$	$[0, 1]$	4	Proba. to perform the migrate vertices operation
P_{merge}	$[0, 1]$	4	Proba. to perform the merge operation
P_{split}	$[0, 1]$	4	Proba. to perform the split operation
T	\mathbb{N}^+	4	Number of graphs generated

a community without splitting it into several disconnected components. If it is set to 1, all the candidate vertices are removed. First, this set V_{toAdd} of vertices that can be potentially removed is selected (lines 1–3). It contains vertices that can be removed without splitting a community into two connected components. It corresponds to the vertices having a within degree equal to 1 (i.e., vertices at the boundaries) if such vertices exist (line 1), otherwise it corresponds to vertices in a triangle (line 3). The candidates for removal are also required not to be a community representative. Once this set has been computed, a number of vertices

Algorithm 3: AddEdges

```

Input:  $v, C$ 
// Within edges
1  $E_{wth} \leftarrow \text{Rand}_{PL}(\min(|C|, E_{wth}^{\max}))$ 
2 while  $\text{deg}_w(v) < E_{wth}$  do
3    $v' \leftarrow \text{Rand}_{EdgeWith}(C \setminus \text{nei}_{wth}(v))$ 
4    $\mathcal{E} \leftarrow \mathcal{E} \cup \{\{v, v'\}\}$ 
// Between edges
5  $E_{btw} \leftarrow \text{Rand}_{PL}(\min(E_{brw}^{\max}, E_{wth}) + 1) - 1$ 
6 while  $\text{deg}_b(v) < E_{btw}$  do
7    $v' \leftarrow \text{Rand}_{EdgeBtw}(v, \bigcup_{\substack{C' \in \mathcal{P} \\ C' \neq C}} C'.rep)$ 
8    $\mathcal{E} \leftarrow \mathcal{E} \cup \{\{v, v'\}\}$ 

```

Algorithm 4: DynamicUpdates

```

// Micro Updates
1 for  $C \in \mathcal{P}$  do
2   if  $\text{Rand}_{Uni}([0, 1]) < P_{micro}$  then RemoveVertices( $C$ ) if  $\text{Rand}_{Uni}([0, 1]) < P_{micro}$  then
   AddVertices( $C$ ) if  $\text{Rand}_{Uni}([0, 1]) < P_{micro}$  then UpdateAttributes( $C$ ) if
    $\text{Rand}_{Uni}([0, 1]) < P_{micro}$  then RemoveWithinEdges( $C$ ) if  $\text{Rand}_{Uni}([0, 1]) < P_{micro}$  then
   AddWithinEdges( $C$ ) if  $\text{Rand}_{Uni}([0, 1]) < P_{micro} \wedge |\mathcal{P}| > 1$  then
   RemoveBetweenEdges() if  $\text{Rand}_{Uni}([0, 1]) < P_{micro} \wedge |\mathcal{P}| > 1$  then
   AddBetweenEdges() for  $C \in \mathcal{P}$  do  $C.rep \leftarrow \text{Sample}(C, \min(|C|, NbRep))$ 
//
// Macro Updates
3 if  $\text{Rand}_{Uni}([0, 1]) < P_{migrate}$  then
4   MigrateVertices()
5   for  $C \in \mathcal{P}$  do  $C.rep \leftarrow \text{Sample}(C, \min(|C|, NbRep))$ 
6 else
7   if  $\text{Rand}_{Uni}([0, 1]) < P_{merge} \wedge |\mathcal{P}| \geq 2$  then
8     MergeCommunities()
9     for  $C \in \mathcal{P}$  do  $C.rep \leftarrow \text{Sample}(C, \min(|C|, NbRep))$ 
10  if  $\text{Rand}_{Uni}([0, 1]) < P_{split} \wedge \exists C \in \mathcal{P}, |C| \geq 2$  then
11    SplitCommunities()
12    for  $C \in \mathcal{P}$  do  $C.rep \leftarrow \text{Sample}(C, \min(|C|, NbRep))$ 

```

from the community depending on parameter $R_{removeVert}$ are picked uniformly and randomly from V_{toAdd} and removed from the graph (lines 4–9).

AddVertices: The algorithm which inserts new vertices into the graph is described as Algorithm 6. In this algorithm, parameter $R_{addVert}$ quantifies the number of vertices added. A high value allows exponential growth in the network to be modeled, in particular, if it is set to 1, the number of vertices in a community is doubled at each timestamp if there is no vertex removed. First, a number of vertices $\lceil |C| \times R_{addVert} \rceil$, depending on parameter $R_{addVert}$, are generated. The attribute values assigned to the new vertices are selected uniformly and randomly between the minimum and maximum observed in the community (line 3). Once the attribute values have been generated, within and between edges are inserted using the preceding function AddEdges presented in Algorithm 3, and the graph structure is updated. The reader is referred to [20] for a detailed discussion in this Algorithm 3.

Algorithm 5: RemoveVertices

Input: C
 1 $V_{toAdd} \leftarrow \{v \in C \mid deg_w(v) = 1 \wedge v \notin C.rep\}$
 2 **if** $V_{toAdd} = \emptyset$ **then**
 3 $V_{toAdd} \leftarrow \{v \in C \mid v \notin C.rep \wedge deg(v) = 2 \wedge \forall v_1, v_2 \in neig(v), \{v_1, v_2\} \in \mathcal{E}\}$
 4 **repeat** $\min(|V_{toAdd}|, \lceil |C| \times R_{removeVert} \rceil)$ **times**
 5 $v \leftarrow RandUni(V_{toAdd})$
 6 $V_{toAdd} \leftarrow V_{toAdd} \setminus \{v\}$
 7 $\mathcal{V} \leftarrow \mathcal{V} \setminus \{v\}$
 8 $\mathcal{E} \leftarrow \mathcal{E} \setminus \{\{v, v'\} \mid v' \in neig(v)\}$
 9 $C \leftarrow C \setminus \{v\}$

Algorithm 6: AddVertices

Input: C
 1 **repeat** $\lceil |C| \times R_{addVert} \rceil$ **times**
 2 $v \leftarrow$ a new vertex
 3 **for** $A \in \mathcal{A}$ **do** $v_A \leftarrow RandUni(\lceil \min_{v' \in C} v'_A, \max_{v' \in C} v'_A \rceil)$ AddEdges(v, C)
 4 $C \leftarrow C \cup \{v\}$
 5 $\mathcal{V} \leftarrow \mathcal{V} \cup \{v\}$

UpdateAttributes: Attribute values are updated according to the approach described in Algorithm 7. For each community C and each attribute $A \in \mathcal{A}$, a set of $\lceil |C| \times R_{updatedAttributes} \rceil$ vertices have their value of attribute A replaced by a value randomly and uniformly taken between the minimum and the maximum of A observed in their within neighborhood. This operation is controlled by the parameter $R_{updatedAttributes}$ which defines the number of updates. A high value indicates a network where the users have strong influence on the others. The higher this value is, the more likely a high proportion of vertices will be impacted, modeling social influence. Note that a value equal to 1 does not imply that all the vertices will have their attribute values updated, nor that all the attribute values of a vertex will be updated.

Algorithm 7: UpdateAttributes

Input: C
 1 **for** $A \in \mathcal{A}$ **do**
 2 **repeat** $\lceil |C| \times R_{updatedAttributes} \rceil$ **times**
 3 $v \leftarrow RandUni(C)$
 4 $v_A \leftarrow RandUni(\lceil \min_{v' \in neig_{wth}(v)} v'_A, \max_{v' \in neig_{wth}(v)} v'_A \rceil)$

RemoveWithinEdges: The removal of within edges (i.e., between vertices of the same community) is presented in Algorithm 8. Parameter $R_{removeWithEdge}$ gives the ratio for between edges that can be removed from the candidates (i.e., the edges in a community that can be removed without splitting the community into several disconnected components). A high value leads to a network having a weak community structure. If it is set to 1, all the candidates are removed. This set of candidates, E_{iri} , is computed (line 1) such that it contains

the edges that are part of a triangle formed by vertices belonging to the community. Only these edges are considered for removal to ensure that the community will remain connected. Then, a number of at most $\lceil |C| \times P_{RemoveEdgeWithin} \rceil$ edges are selected uniformly and randomly from E_{tri} and removed from the graph. Each time an edge is removed, the set E_{tri} is updated because some edges are no longer in a triangle anymore (line 6).

Algorithm 8: RemoveWithinEdges

Input: C

```

1  $E_{tri} \leftarrow \{\{v, v'\} \mid (v \in C) \wedge (v' \in \text{neig}_{wth}(v)) \wedge (\text{neig}_{wth}(v) \cap \text{neig}_{wth}(v') \neq \emptyset)\}$ 
2  $i \leftarrow 0$ 
3 while  $i \neq \lceil |C| \times R_{RemoveWthEdge} \rceil \wedge E_{tri} \neq \emptyset$  do
4    $e \leftarrow \text{RandUni}(E_{tri})$ 
5    $\mathcal{E} \leftarrow \mathcal{E} \setminus \{e\}$ 
6    $E_{tri} \leftarrow \{\{v, v'\} \mid (v \in C) \wedge (v' \in \text{neig}_{wth}(v)) \wedge (\text{neig}_{wth}(v) \cap \text{neig}_{wth}(v') \neq \emptyset)\}$ 
7    $i \leftarrow i + 1$ 
```

AddWithinEdges: Algorithm 9 describes within edges insertion. In this algorithm, parameter $R_{AddWthEdge}$ corresponds to the ratio of within edges inserted and, the number of within edges is set as $\lceil |C| \times R_{AddWthEdge} \rceil$. Consequently, the maximum number of new edges that can be inserted is the number of vertices in the community, for $R_{AddWthEdge} = 1$. For each inserted edge, two vertices v and v' are selected to be connected. The first vertex, v , is selected randomly and uniformly in C such that it is not already connected to all other vertices of C (line 3). The second vertex, v' , is selected among the vertices of C which are not in the neighborhood of v , by preferentially selecting the vertices having a higher degree (line 4). A new edge between v and v' is then inserted into the graph (line 5). Note that the condition on line 2 ensures that this operation will only occur if there is an edge that can be inserted.

Algorithm 9: AddWithinEdges

Input: C

```

1 repeat  $\lceil |C| \times R_{AddWthEdge} \rceil$  times
2   if  $C$  is not a complete graph then
3      $v \leftarrow \text{RandUni}(\{v \in C \mid C \setminus \text{neig}(v) \neq \{v\}\})$ 
4      $v' \leftarrow \text{RandEdgeWth}(C \setminus (\text{neig}(v) \cup \{v\}))$ 
5      $\mathcal{E} \leftarrow \mathcal{E} \cup \{\{v, v'\}\}$ 
```

RemoveBetweenEdges: The method of removing between edges (i.e., edges connecting vertices belonging to different communities) is described as Algorithm 10. Parameter $R_{RemoveBtwEdge}$ is the ratio of between edges removed. All between edges are removed when $R_{RemoveBtwEdge} = 1$ and, consequently, each community is a connected component but on the condition that no between edge is later inserted. The set of between edges in the graph is first computed on line 1. A fraction $\lceil |E_{btw}| \times R_{RemoveBtwEdge} \rceil$ of these edges depending on parameter $R_{RemoveBtwEdge}$ are then removed. To select which edge will be removed, a community C is selected uniformly and randomly among all the communities connected to at least one other community (line 3). Note that such a community C necessarily

exists, otherwise it would mean that there is no more between edge, which is not possible since only a fraction (possibly all) of the between edges are removed (c.f., the constraint on line 2). Then, a vertex v from C having at least one neighbor in another community is selected uniformly and randomly (line 4) and a second vertex v' is selected in the between neighborhood of v : v' is the most distant vertex w.r.t. its attributes (line 5). Finally, the edge $\{v, v'\}$ is removed from the graph (line 6).

Algorithm 10: RemoveBetweenEdges

```

1  $E_{btw} \leftarrow \{\{v, v'\} \in \mathcal{E} \mid \exists (C_1, C_2) \in \mathcal{P}^2, C_1 \neq C_2 \wedge v \in C_1 \wedge v' \in C_2\}$ 
2 repeat  $\lceil |E_{btw}| \times R_{removeBtwEdge} \rceil$  times
3    $C \leftarrow \text{RandUni}(\{C \in \mathcal{P} \mid \exists v \in C, deg_b(v) > 0\})$ 
4    $v \leftarrow \text{RandUni}(\{v \in C \mid deg_b(v) > 0\})$ 
5    $v' \leftarrow \arg \max_{v' \in \text{neig}_{btw}(v)} d(v, v')$ 
6    $\mathcal{E} \leftarrow \mathcal{E} \setminus \{\{v, v'\}\}$ 

```

AddBetweenEdges: Algorithm 11 depicts the approach used for the insertion of between edges. Parameter $R_{addBtwEdge}$ controls the number of between edges inserted at each timestamp. If $R_{addBtwEdge} = 1$, the final number of between edges is twice the number of between edges at the previous timestamp. A set of vertices candidate is first built: It corresponds to the vertices with a within degree higher than the between degree (Line 1). Then $\lceil |V_{cand}| \times R_{addBtwEdge} \rceil$ between edges are inserted by randomly and uniformly selecting a vertex from V_{cand} and a destination vertex belonging to the set of representatives with a higher probability for the vertices similar to v in terms of their attributes according to the Euclidean distance.

Algorithm 11: AddBetweenEdges

```

1  $V_{cand} \leftarrow \{v \in \mathcal{V} \mid deg_b(v) < deg_w(v)\}$ 
2 repeat  $\lceil |V_{cand}| \times R_{addBtwEdge} \rceil$  times
3    $v \leftarrow \text{RandUni}(V_{cand})$ 
4    $v' \leftarrow \text{Rand}_{EdgeBtw}(v, \bigcup_{\substack{C \in \mathcal{P} \\ v \notin C}} C.rep)$ 
5    $\mathcal{E} \leftarrow \mathcal{E} \cup \{\{v, v'\}\}$ 

```

3.1.1 Macro-updates

The macro-updates consist in three operations:

- Migrate a set of vertices from one community to another, possibly a new one (Algorithm 12, MigrateVertices),
- Merge two communities into a single one (Algorithm 13, MergeCommunities),
- Split a community into two new communities (Algorithm 14, SplitCommunities).

These operations are controlled by the parameters $P_{migrate}$, P_{merge} and P_{split} , which define the probability of performing the corresponding operations. Note that the latter operations

occur only if the migrate operation is not performed, i.e., if $P_{migrate} = 1$, then the merge and split operation will never happen (Algorithm 4 (line 10 and 13)).

MigrateVertices: The procedure to migrate vertices from one community to another, possibly a new one, is described as Algorithm 12. The general idea for migration is to select a migrant from an original community. This migrant will leave its community possibly with other vertices from its community. Firstly, the source community C_{from} (i.e., the original community of the departing vertices) is selected: that is the community having the highest diameter (line 1). Then, with an uniform probability given by parameter P_{merge} , the migrating vertices will join an existing community (lines 3 and 4) or build a new community (lines 6–9). In the first case (migrating to an existing community), the migrant is selected randomly and uniformly among the vertices of C_{from} , otherwise the migrant is selected randomly and uniformly among the vertices of C_{from} having at least one neighbor with a degree equal to 1 inside the community to avoid the creation of a community formed by a single vertex. Once the migrant has been selected, the set of migrating vertices is computed (lines 10–16). A vertex u in the within neighborhood of the migrant v joins the new community if either:

- it is connected to more vertices in the destination community C_{to} than in the source community C_{from} (i.e., $deg^{C_{to}}(u) > deg^{C_{from}}(u)$)
- or its neighborhood is connected to more vertices in C_{to} than in C_{from} (i.e., $\sum_{u' \in neig(u)} deg^{C_{to}}(u') > \sum_{u' \in neig(u)} deg^{C_{from}}(u')$)

As this operation can lead to new vertices satisfying the previous conditions, it is repeated until no vertex can migrate.

The next step aims at ensuring that the vertices in the original community remain connected (lines 17–27). The idea is to add edges to connect the vertices which belong to the previous neighborhood of the migrating vertices and which remain in the source community. These vertices are sequentially selected and connected. To avoid adding unnecessary edges, the order used to select a vertex depends on whether it is already connected to another vertices or not (test on line 21).

Finally, a last step ensures that the within degree of the vertices remains higher than their between degree (lines 28–32). To do that, the vertices not satisfying the condition are connected to other vertices in the community, randomly selected according to their degrees (cf. property **P1** in Sect. 2, lines 30 and 31).

MergeCommunities: The approach used to merge two communities is given in Algorithm 13. First, in lines 1–5, the two communities to merge are selected using the following approach. For each pair of communities C_a and C_b , the minimal distance between each pair of representatives in distinct communities is stored in $d_{min}^{C_a, C_b}$, and the average minimal distance d_{avg} is computed on all pairs of communities (lines 1–4). The communities selected (C_a, C_b) are the communities having a minimum distance $d_{min}^{C_a, C_b}$ lower than d_{avg} and the maximum number of between edges (line 5). In case of a tie, a couple is arbitrarily selected. This approach allows both the structural similarity (i.e., the number of between edges) and the similarity of the attributes (i.e., the two communities must be more similar than the average of the communities) to be taken into account. Once the couple has been selected, a new community C_{merge} is created by uniting the vertices (line 6). Edges are then inserted between the vertices of the previously disjointed communities to reinforce the community structure (lines 7–12). First, $density_{min}$ is set as the minimal density in the original communities C_a and C_b , where the density of a set of vertices V is the ratio between the number of edges connecting the vertices in V and the maximum number of possible edges in V , i.e., $density(V) = \frac{2 \cdot |\{(v_1, v_2) \in \mathcal{E} \mid (v_1, v_2) \in V \times V\}|}{|V| \cdot (|V| - 1)}$. Then at most $\lfloor \frac{1 + density_{min}}{2 \cdot density_{min}} \rfloor$ edges are

Algorithm 12: MigrateVertices

```

1  $C_{from} \leftarrow \arg \max_{C \in \mathcal{P}, |C| \geq 2} \text{diameter}(C)$ 
2 if  $\text{RandUni}([0, 1]) < P_{merge}$  then
3    $C_{to} \leftarrow \text{RandUni}(\mathcal{P} \setminus \{C_{from}\})$ 
4    $v \leftarrow \text{RandUni}(C_{from})$ 
5 else
6    $C_{to} \leftarrow \{\}$ 
7    $\mathcal{P} \leftarrow \mathcal{P} \cup \{C_{to}\}$ 
8    $K \leftarrow K + 1$ 
9    $v \leftarrow \text{RandUni}(\{v \in C_{from} \mid \exists v' \in \text{neig}(v), \text{deg}(v') = 1\})$ 
   // Update  $V_{migrate}$  the set of migrating vertices
10  $V_{toAdd} \leftarrow \{v\}$ 
11  $V_{migrate} \leftarrow \{v\}$ 
12 repeat
13    $C_{to} \leftarrow C_{to} \cup V_{toAdd}$ 
14    $C_{from} \leftarrow C_{from} \setminus V_{toAdd}$ 
15    $V_{toAdd} \leftarrow \{u \in \text{neig}^{C_{from}}(v) \mid$ 
      $\text{deg}^{C_{to}}(u) > \text{deg}^{C_{from}}(u)$ 
      $\vee \sum_{u' \in \text{neig}(u)} \text{deg}^{C_{to}}(u') > \sum_{u' \in \text{neig}(u)} \text{deg}^{C_{from}}(u')\}$ 
16 until  $V_{toAdd} = \emptyset$ 
   // Ensure that  $C_{from}$  remains connected
17  $V_{toAdd} \leftarrow \cup_{v' \in C_{to}} \text{neig}^{C_{from}}(v')$ 
18  $v_1 \leftarrow \text{RandUni}(V_{toAdd})$ 
19  $V_{toAdd} \leftarrow V_{toAdd} \setminus \{v_1\}$ 
20 while  $V_{toAdd} \neq \emptyset$  do
21   if  $\exists v_2 \in V_{toAdd}$  s.t.  $\{v_1, v_2\} \in \mathcal{E}$  then
22      $v_1 \leftarrow \text{RandUni}(\{v_2 \in V_{toAdd} \mid \{v_1, v_2\} \in \mathcal{E}\})$ 
23   else
24      $v_2 \leftarrow \text{RandUni}(V_{toAdd})$ 
25      $\mathcal{E} \leftarrow \mathcal{E} \cup \{\{v_1, v_2\}\}$ 
26      $v_1 \leftarrow v_2$ 
27    $V_{toAdd} \leftarrow V_{toAdd} \setminus \{v_1\}$ 
   // Ensure that the within degree is greater than the between degree
28 for  $C \in \{C_{to}, C_{from}\}$  do
29   for  $v' \in C$  do
30     while  $\text{deg}_w(v') < \text{deg}_b(v')$  do
31        $v'' \leftarrow \text{RandEdgeWth}(\{v \in C \mid \{v', v\} \notin \mathcal{E}\})$ 
32        $\mathcal{E} \leftarrow \mathcal{E} \cup \{\{v', v''\}\}$ 

```

inserted between two vertices belonging, respectively, to C_a and C_b , which are not already connected and which are selected randomly according to their degree. Finally, the partition and the number of communities K are updated on lines 13 and 14.

SplitCommunities: This macro-update operation consists in splitting a community into two subcommunities. It is described in Algorithm 14. First, the community C to split corresponds to the community having the highest diameter and at least two vertices (line 1). Intuitively, a community having a large diameter is more likely to be split since there are individuals that can hardly communicate with other members in the community. Then, two leaders are selected as seeds for the newly created communities. The first leader corresponds to the vertex of C having the highest degree (line 3). The second leader is selected among

Algorithm 13: MergeCommunities

```

1 for  $(C_a, C_b) \in \mathcal{P}^2$  do
2    $d_{\min}^{C_a, C_b} \leftarrow \min_{\substack{r_1 \in C_a.rep \\ r_2 \in C_b.rep}} d(r_1, r_2)$ 
3    $d_{avg} \leftarrow d_{avg} + d_{\min}^{C_a, C_b}$ 
4    $d_{avg} \leftarrow \frac{d_{avg}}{K \cdot (K-1)}$ 
5    $(C_a, C_b) \leftarrow \arg \max_{\substack{(C_a, C_b) \in \mathcal{P}^2 \\ C_a \neq C_b}} |\{v_1, v_2\} \in \mathcal{E} \mid v_1 \in C_a \wedge v_2 \in C_b \wedge d_{\min}^{C_a, C_b} < d_{avg}\}|$ 
6    $C_{merge} \leftarrow C_a \cup C_b$ 
7    $density_{\min} \leftarrow \min(density(C_a), density(C_b))$ 
8   repeat  $\lfloor \frac{1+density_{\min}}{2 \cdot density_{\min}} \rfloor$  times
9     if  $\exists \{v_1, v_2\} \notin \mathcal{E}$  s.t.  $v_1 \in C_a \wedge v_2 \in C_b$  then
10       $v_1 \leftarrow RandEdgeWith(\{v \in C_a \mid C_b \not\subseteq \text{neig}(v)\})$ 
11       $v_2 \leftarrow RandEdgeWith(\{v \in C_b \mid \{v, v_1\} \notin \mathcal{E}\})$ 
12       $\mathcal{E} \leftarrow \mathcal{E} \cup \{v_1, v_2\}$ 
13  $\mathcal{P} \leftarrow (\mathcal{P} \setminus \{C_a, C_b\}) \cup \{C_{merge}\}$ 
14  $K \leftarrow K - 1$ 

```

the five remaining vertices having the highest degree: that is the most distant vertex from the first leader, according to the geodesic distance d_g (line 4). On lines 5 and 6, the two new communities C_a and C_b are initialized in such way that C_a contains the first leader and C_b the second one. Vertices from the original community are then assigned either to C_a or C_b as follows. A vertex v is assigned to C_a if its geodesic distance to the first leader is smaller or equal to its geodesic distance to the second leader, otherwise it is assigned to the community C_b . Once all the vertices from C have been assigned to C_a or C_b , several edges are removed between the two new communities, while new edges are inserted inside to ensure community structure. The edge removal is described on lines 10 and 11. Each edge, connecting a vertex of C_a to a vertex of C_b , has a uniform probability defined by parameter $P_{removeEdgeSplit}$ to be removed. The within edges are then inserted. For each vertex v of C_a (or C_b), within edges are inserted for as long as either (1) the within neighborhood of v is empty or (2) its between degree is higher than its within degree and (3) v is not already connected to every other vertices in its community (line 14). While this condition is satisfied, a new within edge is created by connecting v to another vertex selected randomly with regard to its degree (lines 15 and 16). Finally, the partition and the number of communities are updated on lines 17 and 18.

3.2 DANCEer user interface

The generator is available under the terms of the GNU General Public License¹ and its user interface is presented in Fig. 9. It is formed by three views. On the left side, the user selects the generator parameters presented in Table 2. These parameters take their values between 0 and 1. Consequently, lower values lead to few changes in the network, whereas high values induce stronger modifications. The central part displays the generated network or the change in its communities. Several measures, listed in Table 3, such as modularity or homophily are computed on each graph of the dynamic network to describe its properties, notably **P1**, **P2**, **P3**, **P4** and **P5** detailed in Sect. 2. The changes in these different measures on the sequence of

¹ http://perso.univ-st-etienne.fr/langeron/DANC_Generator/.

Algorithm 14: SplitCommunities

```

1  $C_{split} \leftarrow \arg \max_{C \in \mathcal{P}, |C| \geq 2} \text{diameter}(C)$ 
2  $V_{sorted} \leftarrow$  vertices in  $C_{split}$  sorted by decreasing degree
   //  $v_1$  is the vertex from  $V_{sorted}$  having the highest degree
3  $v_1 \leftarrow V_{sorted}[0]$ 
4  $v_2 \leftarrow \arg \max_{v \in V_{sorted}[1:\min(5, |C_{split}|-1)]} d_g(v_1, v)$ 
5  $C_a \leftarrow \{v_1\}$ 
6  $C_b \leftarrow \{v_2\}$ 
7 for  $v \in C_{split} \setminus \{v_1, v_2\}$  do
8   if  $d_g(v, v_1) \leq d_g(v, v_2)$  then  $C_a \leftarrow C_a \cup \{v\}$ 
9   else  $C_b \leftarrow C_b \cup \{v\}$ 
   // Remove between edges
10 for  $\{\{v'_1, v'_2\} \in \mathcal{E} \mid v'_1 \in C_a \wedge v'_2 \in C_b\}$  do
11   if  $\text{RandUni}([0, 1]) < P_{\text{removeEdgeSplit}}$  then  $\mathcal{E} \leftarrow \mathcal{E} \setminus \{\{v'_1, v'_2\}\}$ 
   // Insert within edges
12 for  $C \in \{C_a, C_b\}$  do
13   for  $v \in C$  do
14     while  $(deg_w(v) = 0 \vee deg_w(v) \leq deg_b(v) \wedge deg_w(v) < |C| - 1)$  do
15        $v' \leftarrow \text{RandEdgeWith}(C \setminus (\text{neighbour}(v) \cup \{v\}))$ 
16        $\mathcal{E} \leftarrow \mathcal{E} \cup \{\{v, v'\}\}$ 
17  $\mathcal{P} \leftarrow (\mathcal{P} \cup \{C_a, C_b\}) \setminus \{C_{split}\}$ 
18  $K \leftarrow K + 1$ 

```

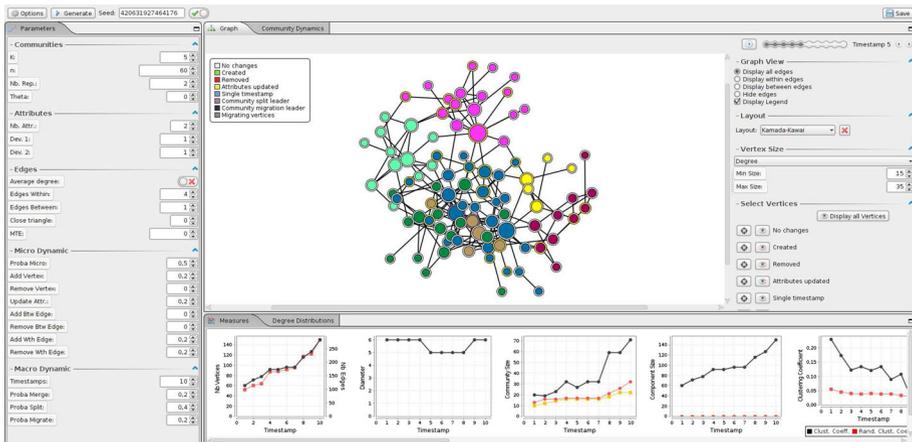


Fig. 9 User interface of the generator DANcer

graphs are presented at the bottom of the interface as Fig. 9 shows. The generated dynamic network, the parameters and the measures can be saved as a collection of files. Note that a seed is used for the random number generator. It can be saved to reproduce exactly the same first graph. The user interface is described in more detail in “Appendix 2.” Moreover, a set of predefined benchmark profiles is presented in “Appendix 3.” The parameter setting is given for each network as well as its characteristics: modularity, within inertia, micro- and macro-dynamicity.

Table 3 Measures computed on each graph \mathcal{G}_i

Measures
Number of vertices $ \mathcal{V}_i $ and number of edges $ \mathcal{E}_i $
Diameter of the graph \mathcal{G}_i
Size of the three largest communities in \mathcal{G}_i
Size of the connected components in \mathcal{G}_i
Clustering coefficient of the graph \mathcal{G}_i
Average degree in the graph \mathcal{G}_i
Modularity of the partition in \mathcal{G}_i
Number of communities in \mathcal{G}_i
Average shortest path measured on \mathcal{G}_i
Within and between inertia ratios for \mathcal{G}_i
Observed and expected Homophily measures for \mathcal{G}_i
Number of within and between edges in \mathcal{G}_i
Degree distribution in \mathcal{G}_i

4 Experiments

A set of experiments has been carried out to demonstrate how the parameters can be used to generate dynamic attributed networks with a community structure. In this section, we show our experimental results. Firstly, we study how the parameters for the micro- and macro-updates change the graph measures. In each experiment, we consider the impact of one parameter on one measure. Then, we show how it is possible to obtain different evolutions of the communities using the macro-parameters. Finally, we report the run times obtained for various sets of parameters.

Default parameters: Unless stated otherwise, all the experiments have been performed using a base setting with 1000 vertices (i.e., $N = 1000$), 10 communities having 10 representatives each (i.e., $K = 10$, $NbRep = 10$). Each vertex is assigned with two attributes having a standard deviation of 1 (i.e., $p = 2$ and $\mathcal{A} = \{1, 1\}$). The parameters E_{with}^{\max} and E_{btw}^{\max} have been, respectively, set to 6 and 3, and the minimum number of edges (MTE) is 5000. The parameter $P_{RandomCommunity}$ is set to 0 (i.e., no degradation on the attributes associated to a community). The probability of performing the micro-updates is fixed to 1 (i.e., the micro-updates operations are always performed). By default, the ratio for each micro-update operation is 0 as well as the probability for the macro-updates (i.e., they are not performed). The number of timestamps is 10.

For each set of parameters, ten dynamic graphs are generated, each one using a different random seed, in other words, the ten dynamic networks, each composed of ten graphs, are different even though they have been generated with the same set of parameters. The reported results correspond to the mean of the measures computed on these ten dynamic networks. The standard deviation is not given since the obtained results are stable.

4.1 Impact of the parameters on graph measures

The first set of experiments aims at studying the variation of several graph measures depending on the value of each micro- and macro-parameter considered independently. In particular, we focus on measures suited to evaluate the homophily and the graph structure, like the clustering

Fig. 10 Evolution of the average clustering coefficient for micro-update parameters ranging between 0 and 1

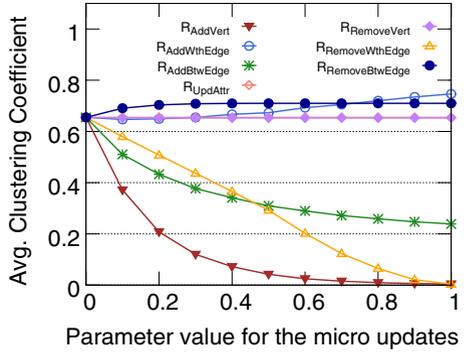


Fig. 11 Evolution of the average clustering coefficient for macro-update parameters ranging between 0 and 1

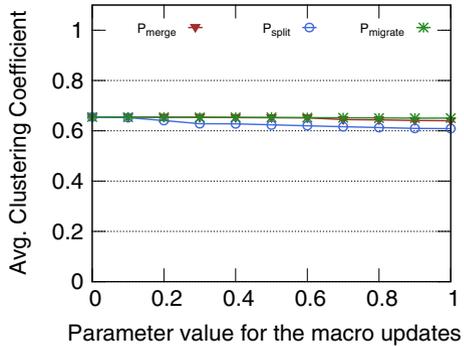
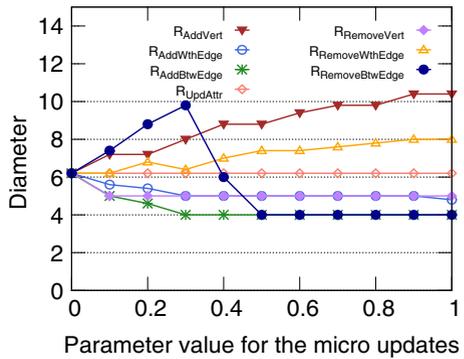


Fig. 12 Evolution of the diameter for micro-update parameters ranging between 0 and 1



coefficient average, the modularity and the within inertia. We also evaluate classical properties like the small world through the diameter and the average length of the shortest paths.

Figures 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22 and 23 are dedicated to the evolution of one measure (on the second axis) and presents the impact of the micro- and macro-parameters on the measure: Each curve corresponds to a parameter whose values are given on the abscissa, the others remaining stable.

Each figure presents the mean value computed on the ten dynamic networks at the last timestamp (i.e., the last graph of each sequence) except when considering parameter $R_{AddVert}$ where the diameter, the average length of the shortest paths, the number of within and between edges are computed on the graph generated at timestamp 5. Indeed, when $R_{AddVert} = 1$, the

generation of the graph is not a problem, but even though an efficient version of the shortest path computation has been implemented, this measure remains intractable for large graphs.

4.1.1 Average clustering coefficient

Figures 10 and 11, depict the evolution of the clustering coefficient [16]. These results demonstrate that our generator is able to build graphs having a community structure since the average clustering coefficient remains higher than 0.6 for most parameter settings.

When considering the micro-update operations (Fig. 10), we observe that three operations tend to decrease the clustering coefficient: adding vertices, removal of within edges and adding between edges. Adding vertices, and even with the additional edges our algorithm does, dilutes the density of the network; hence, the expected drop of the average clustering coefficient. Increasing $R_{removeWthEdge}$ seems to decrease the average clustering coefficient linearly. This behavior is coherent since most triangles are found within the communities; therefore, removing within edges is likely to remove these triangles.

Regarding the insertion of between edges, we note that there are only few edges between the communities at the first timestamp. Consequently, the first between edges inserted are unlikely to form new triangles. Thus, the measure decreases quickly for low values of the parameter. However, when more between edges are inserted, triangles are more likely to be created. Consequently, the drop tends to stabilize for larger parameter values. Conversely, the increase in parameters $R_{addWthEdge}$ or $R_{removeBtwEdge}$ tends to slightly improve the clustering coefficient. This is coherent since both operations improve the community structure.

Finally, the macro-update operations (Fig. 11) have little impact on the average clustering coefficient. Consequently, the proposed approach for the generation of dynamic social networks seems relevant since it does not degrade the community structure over time.

4.1.2 Diameter

Figures 12 and 13 present the evolution of the graph diameter, defined as the longest shortest path between any pairs of vertices. If the graph contains several disconnected components, the diameter is the longest shortest path computed in considering each component independently. The value of the diameter in the generated graphs remains lower than 11, which is in accordance with the small-world property observed in many real networks.

When parameter $R_{removeBtwEdge}$ increases (ratio of between edges removed), we can distinguish three phases. When it is between 0 and 0.3, the graph contains a large connected component composed of communities connected by only few edges (the higher $R_{removeBtwEdge}$, the fewer the number of between edges). Consequently the diameter increases. However, once sufficient between edges have been removed (i.e., $R_{removeBtwEdge} > 0.3$ in our experiments), a community can be completely disconnected. When it occurs, the diameter is computed on each connected component and cannot be greater than the diameter of the whole graph. This behavior explains the drop in the diameter when $0.3 < R_{removeBtwEdge} \leq 0.5$. When $R_{removeBtwEdge} > 0.5$, there are no more between edges and consequently the diameter remains stable.

Adding vertices tends to increase the diameter because inserted vertices may be connected to a vertex belonging to the previous longest shortest path; this leads the diameter being increased by one.

Conversely, the vertices from different communities are more likely to be connected when between edges are added; this tends to decrease the diameter. Note that removing vertices also decreases the diameter since removed vertices are those having a single neighbor.

Fig. 13 Evolution of the diameter for macro-update parameters ranging between 0 and 1

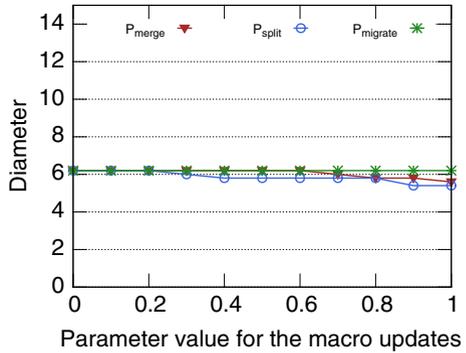


Fig. 14 Evolution of the average shortest path length for micro-update parameters ranging between 0 and 1

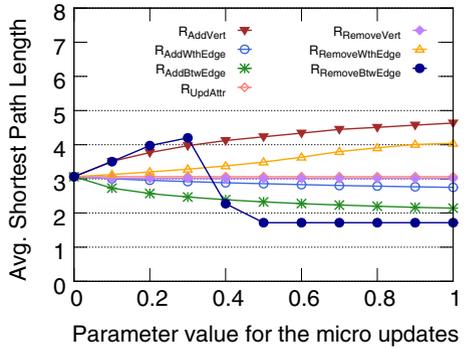
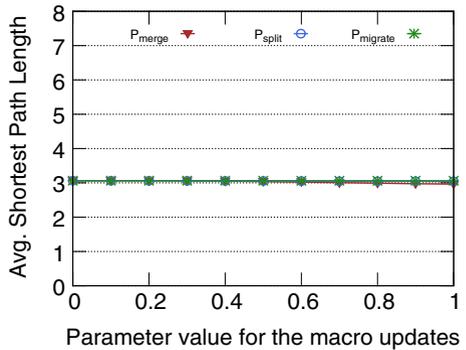


Fig. 15 Evolution of the average shortest path length for macro-update parameters ranging between 0 and 1

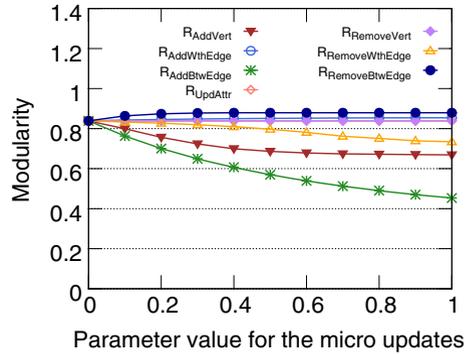


Finally, the macro-update operations have almost no impact on the diameter. This behavior underlines that the macro-operations maintain the small-world property.

4.1.3 Average shortest path length

The behavior of the average length of the shortest paths is depicted in Figs. 14 and 15. The average shortest path length corresponds to the mean length of the shortest path among each pair of vertices if such a path exists. The behavior of this measure is similar to the one obtained for the diameter since both are based on the length of the shortest path between pairs of vertices.

Fig. 16 Evolution of the modularity for micro-update parameters ranging between 0 and 1



4.1.4 Modularity

Figures 16 and 17 present the change in the modularity. The modularity aims at measuring if a graph has a community structure, i.e., if there is a partition of the vertices such that vertices within a community are more connected with vertices in the same community compared to vertices in the others. We used the modularity measure proposed by [29] computed as $\frac{1}{2m} \cdot \sum_{(i,j) \in \mathcal{V}^2} \left(A_{i,j} - \frac{\text{deg}(i) \cdot \text{deg}(j)}{2m} \right) \cdot \delta(i, j)$, where m is the number of edges in the graph, A is the adjacency matrix of the graph and $\delta(i, j)$ is the Kronecker function evaluated to 1 if i and j belong to the same community and 0 otherwise. A graph is considered to have a community structure if its modularity measure is above 0.3 [29].

We note that adding vertices tends to decrease the modularity slightly. This behavior is due to the high within density of the communities generated by adding extra edges in the graph at the first timestamp (recall that the parameter MTE is fixed to 5000). Consequently, adding new vertices tends to reduce the density within the communities slightly, leading to a lower modularity which stabilizes above 0.6. As expected, adding between edges or removing within edges also tends to decrease the modularity, whereas removing between edges has unsurprisingly the opposite effect.

Regarding macro-update operations, the modularity remains relatively stable for the migrate and split operations but drops to 0 for the merge operation. This is a known effect of the modularity since a graph composed of a single community has a modularity equal to 0. In our experiments, after 10 timestamps, if the merge operation occurs every time (i.e., $P_{merge} = 1$), the original communities are merged into a single community.

To conclude, these results demonstrate that unless extreme parameter values are used, the generated graphs have a good community structure with respect to modularity.

4.1.5 Within inertia ratio

The evolution of the within inertia is presented in Figs. 18 and 19. The within inertia is based only on the attribute values assigned to the vertices. It corresponds to the sum of the squared Euclidean distance of the vertices within a community C to the center of gravity of their community C . The within inertia ratio is simply the within inertia normalized by the total inertia (i.e., the sum of the squared Euclidean distance of all the vertices to the center of gravity of all the vertices). The lower the measure is, the better the homogeneity with respect to the attribute values. Note that this measure depends on the number of communities; in

Fig. 17 Evolution of the modularity for macro-update parameters ranging between 0 and 1

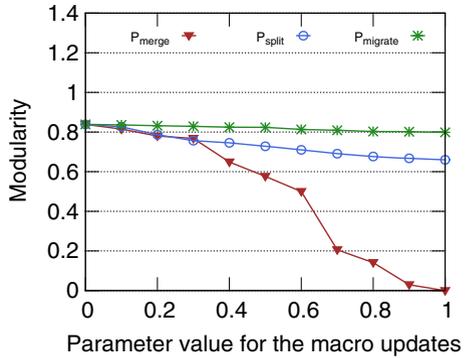
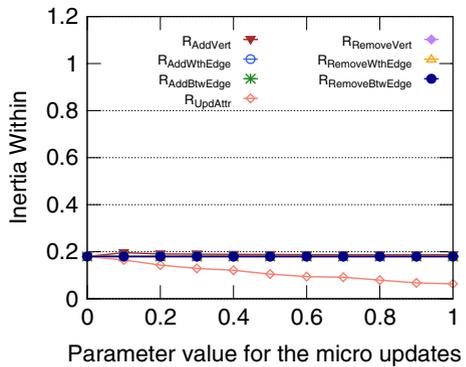


Fig. 18 Evolution of the within inertia ratio for micro-update parameters ranging between 0 and 1



particular when there is only one community, the within inertia is equal to the total inertia so the within inertia ratio is 1. This effect is observed in Fig. 19 for the merge operation.

The update attribute operation and consequently, the high values for parameter $UpdAttr$, tend to decrease the measure, confirming an improvement of the homogeneity. This behavior can be explained by the fact that the new attribute values are bounded by the attribute values of the vertices within the community modeling the notion of social influence. Consequently, after several iterations, the outliers with regard to the attributes are likely to become more similar or closer to the other vertices within the community.

The other operations have little impact on the measure since they focus on the transformation of the graph structure and not the attribute values.

4.1.6 Homophily

The homophily measure takes into account both the structure of the graph and the attribute values assigned to the vertices. The *observed homophily* is the number of pairs of vertices being similar with regard to the attribute values and being connected, normalized by the number of edges in the graph. It can be compared to the *expected homophily*, which is the number of pairs of similar vertices normalized by the number of pairs of vertices in the whole graph (i.e., not considering edges). In our experiments, we considered that two vertices v and v' are similar iff. $\exists A \in \mathcal{A}$ s.t. $|v_A - v'_A| < \frac{2\sigma_A}{K}$ where σ_A is the standard deviation parameter associated with the attribute A and K is the number of communities in the graph. If the

Fig. 19 Evolution of the within inertia ratio for macro-update parameters ranging between 0 and 1

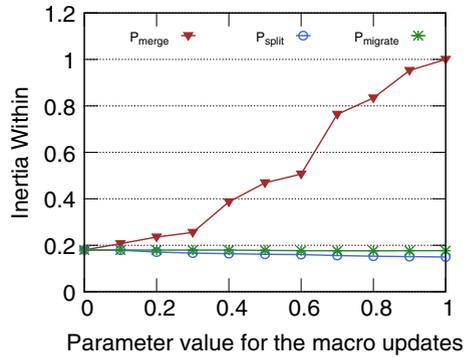
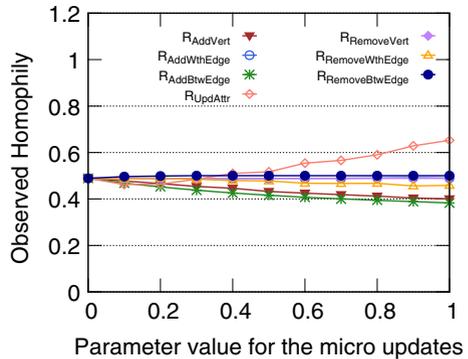


Fig. 20 Evolution of the observed homophily for micro-update parameters ranging between 0 and 1



observed homophily measure is above the expected homophily, we can conclude that similar vertices are more likely to be connected.

From Fig. 20, we observe that the homophily increases with $R_{UpdAttr}$, the ratio of update attributes. The reason for this behavior is the same as the one described for the within inertia measure. The vertices having an updated attribute value are more likely to be similar to other vertices in their community, and consequently, the homophily increases, while the expected homophily remains stable (Fig. 22). Conversely, adding between edges tends to decrease the homophily since these edges are more likely to connect vertices which are not similar.

For the macro-operations described in Fig. 21, we note that the homophily increases to its maximum when $P_{merge} = 1$ (i.e., when the merge operation is performed at each timestamp). This is due to the definition of the similarity threshold which takes into account the number of communities and to the fact that for $P_{merge} = 1$, the final graphs contain only one community. Indeed, since the attribute values are generated according to a normal law, given our similarity definition, the probability that two vertices are similar within the final community is 0.95. Nonetheless, the observed homophily always remains higher or equal to the expected homophily presented in Fig. 23.

4.2 Communities operations

The second set of experiments aims at illustrating different evolutions of a network over time. Figures 24, 25 and 26 are, respectively, dedicated to three dynamic networks generated with different values for parameters P_{merge} governing the merge operation and P_{split} governing

Fig. 21 Evolution of the observed homophily for macro-update parameters ranging between 0 and 1

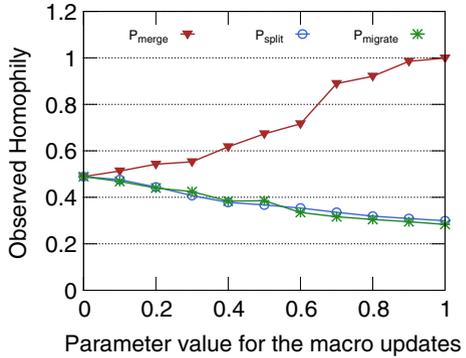


Fig. 22 Evolution of the expected homophily for micro-update parameters ranging between 0 and 1

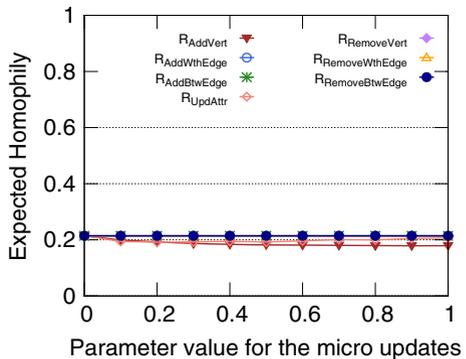
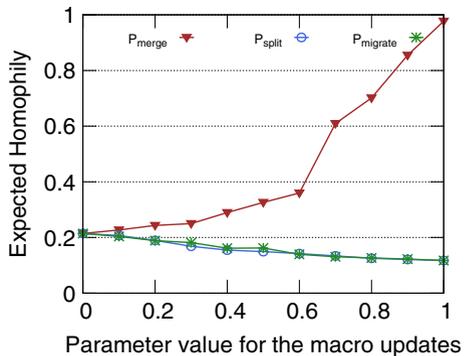


Fig. 23 Evolution of the expected homophily for macro-update parameters ranging between 0 and 1



the split operation. In the figures, each community corresponds to a color and a numeric identifier. Each column in the figures is a graph at a timestamp, and each row corresponds to a community evolution. A circle is a community, and an edge is the transition of a community from one timestamp to the next, illustrating merges and splits. We considered that when a community is split, it leads to two different communities. Similarly, when two communities are merged, the resulting community is considered as a new one. The edges and labels linking the communities indicate the number of vertices shared by two communities at consecutive timestamps.

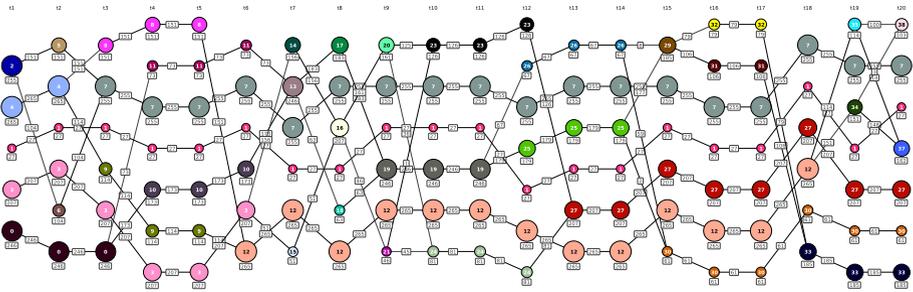


Fig. 24 Communities evolution when $P_{split} = 0.5$ (split), $P_{merge} = 0.5$ (merge) and $K = 5$

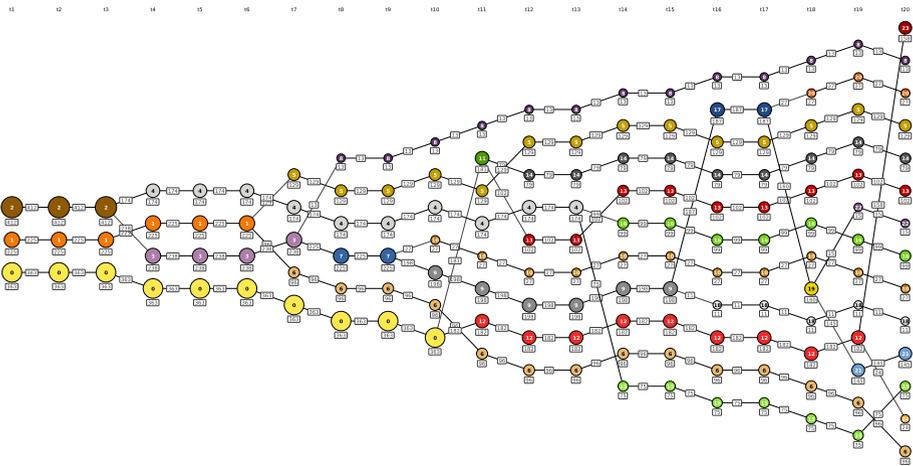


Fig. 25 Communities evolution when $P_{merge} = 0$ (merge), $P_{split} = 0.5$ (split) and $K = 3$

In Fig. 24, parameters P_{merge} and P_{split} are equal (0.5); consequently, the number of communities remains relatively stable over time. The number of communities is at most equal to 7 and at least equal to 5. During the 20 timestamps, 10 merge operations and 12 split operations occur. Note that at the timestamp 8, the community 16 comes from the merge of two communities (14 and 15) and is then split to build the communities 20 and 21 (timestamp 9).

In Fig. 25, only the split operation occurs (i.e., $P_{split} = 0.5$ and $P_{merge} = 0$). As expected, the number of communities increases over time, from 3 communities in the first graph to 14 communities in the last generated graph.

Conversely, in Fig. 26, only the merge operation occurs (i.e., $P_{split} = 0$ and $P_{merge} = 0.5$). This time, the number of communities decreases over time: The ten original communities are merged into a single community at the last timestamp. In this example, communities 0 and 2 remain very stable over time and are merged at the last two timestamps.

4.3 Runtimes

The last set of experiments presents the running times for the network generation depending on the parameters. The times include the generation of all the graphs corresponding to the network. Note, however, that the first graph which requires more time can be saved, and in a second phase, the dynamic updates with varying parameter settings can be performed. The

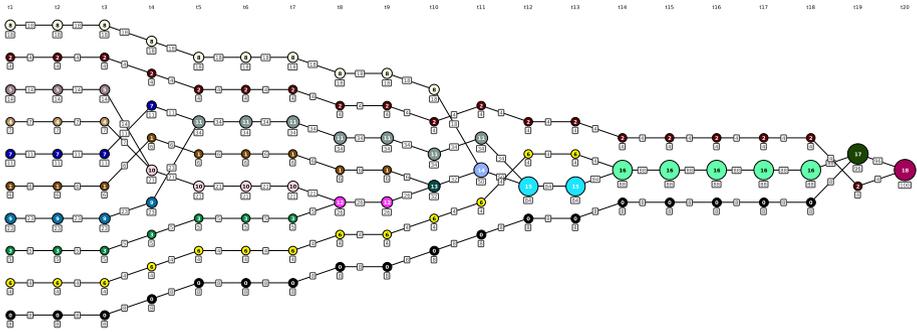


Fig. 26 Communities evolution when $P_{merge} = 0.5$ (merge), $P_{split} = 0$ (split) and $K = 10$

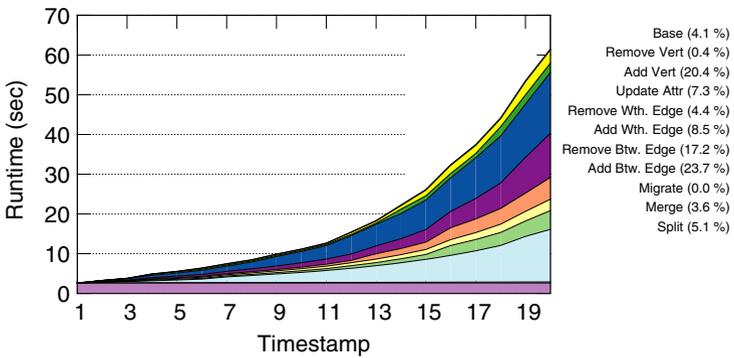
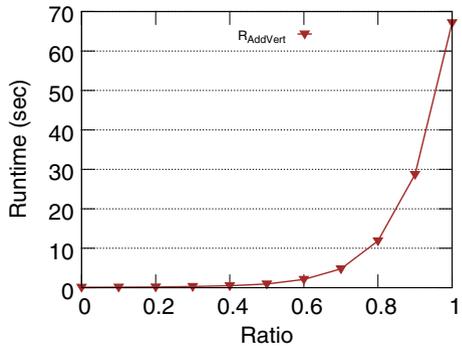


Fig. 27 Change in the runtime over 20 timestamps for each micro- and macro-operations. The values are cumulative, i.e., the reported value for an operation at a given timestamp includes the sum of the runtimes for this operation during the previous timestamps. The percentage corresponds to the ratio between the sum of the runtimes for an operation at each timestamp and the total time required to generate the dynamic graph

Fig. 28 Evolution of the runtime w.r.t. the ratio of vertices inserted



experiments were performed on a PC running GNU/Linux with an Intel Core i7 and 16 Gb of main memory.

Figure 27 presents the evolution of the runtimes for a parameter setting that may correspond to real-world graphs. All the micro- and macro-parameters were set to 0.2, the original number of vertices is 10,000 and the graph contains 100 communities at the first timestamp (i.e., $N = 10,000$ and $K = 100$). The final graph has 81,806 vertices and 328,016 edges. The

Fig. 29 Evolution of the runtime w.r.t. the micro-removal operations and the update attribute operation

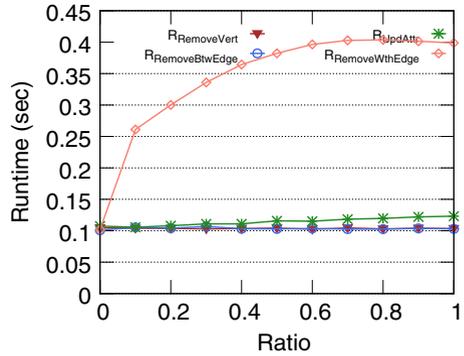


Fig. 30 Evolution of the runtime w.r.t. the edge insertion operations

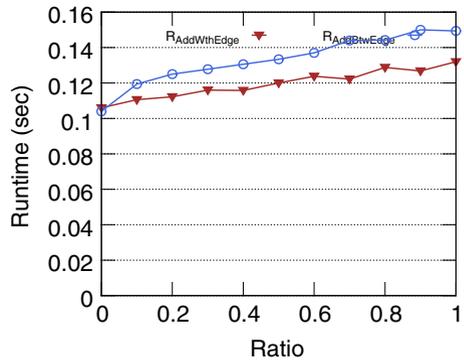
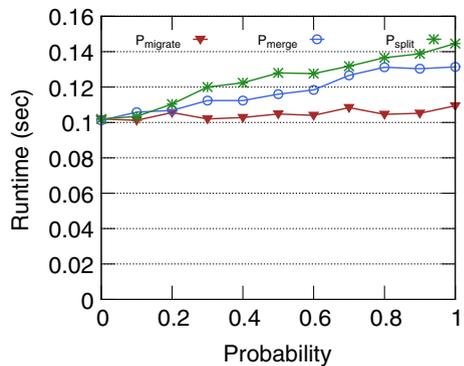


Fig. 31 Evolution of the runtime w.r.t. the macro-update operations



generation of this dynamic network composed of 20 graphs takes approximately 1 min. The most time-consuming operation is the insertion of between edges (23.7% of the total time) since it requires time to compute the Euclidean distance between a vertex and the representatives. Note that since the community representatives and the attribute values change over time, it is not possible to compute this distance in advance. The insertion of vertices is the second most time-consuming operation (20.4%) because it also requires processing time to insert between edges. Note that even though the split operation requires the diameter of the communities to be computed, since the size of the communities remains usually much smaller than the size of the whole graph, it only requires 5.1% of the runtime in our experiments.

Figures 28, 29, 30 and 31 present the evolution of the runtime considering each parameter independently. The parameter setting is the same as in Sect. 4.1. To better visualize the evolution of computational costs, the micro-updates operations are displayed in three distinct plots.

Figure 28 presents the evolution of the runtime with respect to parameter $R_{addVert}$. The shape is clearly polynomial, and the number of inserted vertices is also polynomial. Indeed, when $R_{addVert} = 1$, the number of vertices is doubled at each timestamp. Consequently, the insertion cost of a single vertex remains linear, and this operation is tractable for large graphs when used in conjunction with the vertex removal operation as demonstrated in Fig. 27.

Figure 29 depicts the runtimes for the removal and attribute update operations. The cost of the operation consisting in removing within edges is due to the update of the edges in a triangle, which must be performed after each deletion to ensure the connectivity inside each community. The runtime of the other operations remains almost constant with respect to their parameter.

Regarding the edge insertion presented in Fig. 30, the runtime slightly increases but remains tractable even for high parameter values.

Finally, the runtimes for the macro-update operations are presented in Fig. 31. Again, the runtime shows a slight increase, particularly for the split operation. This cost is mainly due to the computation of the communities diameter required to select which community to split.

To conclude this experimental section, the obtained results demonstrate that the proposed approach is able to generate large graphs (up to 250,000 vertices) in a reasonable time (about 70 s). Moreover, our approach allows the generation of dynamic graphs having a community structure, and the same properties as real graphs. In addition, these properties can be degraded by varying the parameter values.

5 Related works

Synthetic graphs are important for developing scenarios, extrapolations, and simulations, when real graphs are impossible to collect or expensive to obtain. This leads to design models and generators, which are very useful for assessing the quality of algorithms for tasks such as community detection or link prediction.

5.1 Static networks

There is a large bibliography regarding network generation for graphs that are not dynamic. Among the best known generative models, we cite the classic Erdős-Rényi (ER) model [10], which generates random graphs where edges are formed independently and with uniform probability or the Barabási-Albert (BA) model [5] that generates random scale-free networks. Other notable synthetic generators are the mathematical tractable models, such as the Stochastic Kronecker Graph model [24], and its generalization, the Multifractal Network model [6, 30]. With regard to the networks with community structure, the first generator was introduced by [11]. Several extensions of this GN benchmark have been proposed notably the **LFR benchmark** [18, 19]. However, all these generators synthesize static graphs without attributes. Concerning networks with attributes, the generators are based on the homophily hypothesis, and they suppose that vertices having the same characteristics are more likely to be connected [1, 12, 17, 33].

Recently, in [31], a framework which jointly models network structure and vertex attributes is presented. The framework learns the attribute correlations in the observed network and

uses a generative graph model, such as the Kronecker Product Graph Model (KPGM) [24] and Chung Lu Graph Model (CL) [7], to compute structural edge probabilities. The model combines the attribute correlations with the structural probabilities to sample networks conditioned on attribute values, while keeping the expected edge probabilities and degrees of the input graph model. Thus, the structural characteristics provided by the graph models are maintained when incorporating the attribute dependencies. This model generalizes and uses a common key structural assumption of generative network models, where the model incrementally selects an observed edge from all possible edges and inserts it into the generated network.

Finally, very few generators allow the construction of a network having both a community structure and attributes associated with the vertices. Dang presents a basic extension of the BA model [5], without experiments confirming the properties of the generated graphs [8]. The generator introduced in [20] is the static version of the model described in this article.

5.2 Dynamic networks

Very few generators allow a dynamic network composed of a sequence of evolving graphs to be constructed.

[25] empirically observed that real networks become denser over time, with the average degree increasing and the diameter decreasing in many cases. From these conclusions, they provided two models of graph generators. The first model, Community Guided Attachment (CGA), is a probabilistic generative model, which produces hierarchical graphs with a densification power law such that the linkage probability between nodes decreases as a function of their relative distance in the hierarchy. An extended version has been proposed in which nodes and their out-links are added over time. As shown by the authors, this dynamic version exhibits different behaviors according to the parameter setting: densification power law with heavy-tailed in-degrees, constant average degree with heavy-tailed in-degrees, constant in- and out-degrees with high probability.

Forest Fire, the second model of evolving graphs introduced in [25], seeks to capture the average degree increase as well as the shrinking diameter that real networks tend to have and that the Community Guided Attachment models do not exhibit.

Another recursive generative model, introduced in [23], satisfies the two properties of densification power law and shrinking diameters. The recursive process of graph construction is based on the Kronecker product of the adjacency matrix of current graph at time t with itself to generate the graph at time $t + 1$. This process generates self-similar Kronecker graphs (growing sequence of graphs) by iterating the Kronecker product, and it has been shown that the aforementioned properties of time-evolving networks are satisfied but with a staircase effect in the degree distribution. A stochastic version of Kronecker graphs was then introduced to overcome this issue.

In [2], the authors studied properties of several time-evolving weighted graphs. They observe that they exhibit a power law for the largest eigenvalue of the adjacency matrix and the number of edges over time. They also show that the weight of a given edge and weights of its neighboring two nodes are correlated. They proposed a generator satisfying these properties and based on the same principle as in [23]. The recursive process starts with a tensor representation of the initial graph and use *recursive tensor multiplications* to produce growing graphs over time.

In a recent work, [15] propose a model for generating simple dynamic benchmark graphs for community detection. The model, based on the classic stochastic block model, considers a periodic evolution, such that the same configuration of communities is repeated in cycles

and is invariant under time reversal. It generates three classes of dynamic benchmark graphs for time-evolving networks, such that at each snapshot, the partition into communities is well defined. In this benchmark, a network is divided into a number of subgraphs or communities. The cyclical evolution of the graph is modeled by two dynamic processes: growing/shrinking and merging/splitting of the communities. From this configuration, three benchmarks can be generated: The first one consists in communities which grow and shrink in size with a fixed total number of nodes in the network, while the second considers communities that merge and split. The third one is a mixed combination of the last operations.

In [14], the authors studied the generation of dynamic random graphs such that the graph changes dynamically by node and edge insertions or deletions and where the graph incorporates a clustering structure (communities), which also changes dynamically. They describe a random dynamic graph generator, which is based on the Erdős-Rényi model by adding mechanism of dynamics to the model accounting for evolving events on clustering structure. The model uses a custom graph generator based on the planted partition model that introduces dynamics by splitting and merging clusters in the ground truth clustering. In each time step, one edge or vertex is added or deleted according to the probabilities prescribed by the current ground truth clustering. The actual graph structure follows the ground truth clustering with some delay. They also provide an efficient implementation of this generator in [13].

The aforementioned generators allow the construction of dynamic networks, exhibiting or not a community structure but, none of them takes into account the characteristics of the vertices. The interest of discovering communities in dynamic networks where vertices are associated with attributes led us to develop this tool.

6 Conclusion

Interrelated data that are modeled in graphs are becoming ubiquitous in all disciplines that tend to amass digital information. These data, in the form of large networks, are complex and difficult to replicate. Real-world information networks have fundamental properties, such as power law degree distribution, the small-world phenomenon, homophily and social influence. Another central property is that information networks tend to organize according to an underlying modular structure called communities. The proliferation of complex information networks in diverse fields of application has led to the proposal of a panoply of approaches to analyze and discover relevant patterns in these networks. However, comparing and assessing these approaches on effectiveness and efficiency is a significant challenge. The challenge is compounded by the lack of large real networks with ground truth that could be easily and freely accessible to researchers. The alternative is to synthesize data, and many approaches have been suggested. However, while real-world information networks evolve continuously, very little has been proposed so far in terms of generation of dynamic complex networks that have attribute values attached to the nodes. In this paper, we proposed a generator, DANCer, for attributed dynamic graphs with embedded community structure. After introducing the fundamental properties of real-world attributed graphs, we presented a detailed algorithm that malleably, with adjustable parameters, can produce a sequence of evolved attributed graphs that abide by the mentioned properties. Our wide-ranging experiments demonstrate the effectiveness of our algorithm and its scalability to relatively large networks. We provide a freely available tool to efficiently generate dynamic attributed networks with community structure that closely obey the properties of real large information networks. Note that our generator can trivially be extended to produce multiplex networks, also called multilayered

or multi-level networks, where all nodes are omnipresent in all levels and intra-level edges connect the representations of a node from one level to the other. This conversion is possible by simply converting each timestamp graph into a layer of the multiplex network and adding the necessary intra-level edges. It remains that other less trivial extensions could also be beneficial, in particular the possibility to include categorical attribute values. Currently, DANCer can only generate attributed networks with numerical values. Another useful addition could be the possibility of overlapping communities. This can be done for the initial graph before the changes, as well as at the level of the macro-operations allowing nodes to join a new community while remaining in their original one. Lastly, one could also consider the generation of networks having communities with a hierarchical structures, a hierarchy to be considered in the dynamic operations.

Appendix 1: Additional functions

See Table 4.

Table 4 Additional functions used in the algorithms

Operations	Description
$deg(v)$	Degree of vertex v
$deg_w(v)$	Degree of vertex v , counting only edges connecting v to vertices in its community
$deg_b(v)$	Degree of vertex v , counting only edges connecting v to other communities
$neig(v)$	Set of neighbors of vertex v
$neig_{wth}(v)$	Set of neighbors of vertex v inside its community
$neig_{btw}(v)$	Set of neighbors of vertex v outside its community
$d(v, v')$	Euclidean distance between the attributes of the two vertices v and v'
$dist(v, C)$	distance of the vertex v to the community C defined by $\sum_{v' \in C} \frac{d(v, v')}{ C } + \min_{v' \in C} d(v, v')$
$Sample(S, L)$	Randomly sample L values from the set S
$Rand_{Uni}(S)$	Returns an element of the set S selected uniformly and randomly
$Rand_{PL}(m)$	Returns a natural number belonging to $\{1, \dots, m\}$ randomly selected using the density function $f : x \mapsto \frac{x^{-2}}{\sum_{i=1}^m i^{-2}}$
$Rand_{EdgeWth}(V)$	Returns a vertex u from V randomly selected according to the probability density function $f : u \mapsto \frac{deg(u)}{\sum_{u' \in V} deg(u')}$
$Rand_{EdgeBtw}(v, V)$	Returns a vertex u from V randomly selected according to the probability density function $f : u \mapsto \frac{d(v, u)^{-1}}{\sum_{u' \in V} d(v, u')^{-1}}$
$KMedoids(V, K)$	Performs clustering on the vertices of V using KMedoids method to build K clusters

Appendix 2: User manual

The software **DANCer** as well as a detailed user manual is available at http://perso.univ-st-etienne.fr/lageron/DANC_Generator/. The user interface of **DANCer** generator is formed by three views as shown in Fig. 9.

Graph parameters

The parameters are on the left panel. They correspond to the parameters of algorithms given in Table 2. They are detailed below.

Communities

- K : Number of communities in the first graph;
- n : Number of vertices in the first graph;
- *Nb. Rep.* : Number of representatives in each community. The higher is the value, the slower is the computation;
- Θ : Percentage of vertices assigned to a random community. The higher is this value, the less likely the community will be homogeneous w.r.t. the attributes.

Attributes

- *Nb. Attr.* : Number of real attributes associated with the vertices. Each attribute is distributed according to centered normal distribution with mean equals to 0;
- *Dev. i* : Standard deviation of the i th attribute.

Edges

- *Edges Within* : Maximum number of within community edges added to a newly inserted vertex;
- *Edges Between* : Maximum number of between community edges added to a newly inserted vertex
- *MTE* : Minimum number of edges in the resulting graph (up to a graph where communities are cliques).

Micro-dynamic

- *Proba Micro* : The probability to perform a micro-update operation;
- *Add Vertex* : The ratio of vertices created at each timestamp. When set to 1, the number of vertices is doubled at each timestamp;
- *Remove Vertex* : The ratio of vertices removed at each timestamp;
- *Update Attr.* : The ratio of vertices having their attribute values updated;
- *Add Btw. Edges* : The ratio of edges inserted connecting two vertices in different communities;
- *Remove Btw. Edges* : The ratio of edges removed connecting two vertices in different communities;
- *Add With. Edges* : The ratio of edges inserted connecting two vertices in the same communities;

- *Remove With. Edges* : The ratio of edges removed connecting two vertices in the same communities;

Macro-dynamic

- *Timestamps* : The number of timestamp (i.e., the number of single graphs generated to form the dynamic network);
- *Proba Merge* : The probability to perform a merge operation at a single timestamp (i.e., merging two communities into a single one);
- *Proba Split* : The probability to perform a split operation at a single timestamp (i.e., split one community into two)
- *Proba Migrate* : The probability to perform a migrate operation at a single timestamp (i.e., migrate vertices from a community to either a new or an existing community).

Network reproduction

- *Seed* parameter : A seed is used for the random number generator. It allows to reproduce exactly the same network.

Graph visualization and manipulation

The central part of the user interface as shown in Fig. 9 allows to display the generated network and the changes in its communities at each time step. Each graph in the sequence can be viewed separately in the **Graph** tab. The sequence of graphs can also be visualized through the timestamp scrollbar at the right side of the panel.

For each graph plotted, in the *Graph View* tab, we can set different options (see Fig. 32) allowing, for example, to hide or display the edges and vertices through the *Graph View* section at the right side panel. The graph can then be displayed with different layout options (*kamada-kawai*, *fruchtmann-reynolds* or *self-organizing map*) where the sizes of the plotted vertices are chosen according to their degree, age or community membership. Moreover, we can select or filter the displayed vertices according to their different events, as described in the micro-dynamic operations, from the *Select Vertices* panel.

In the plotted graph, vertices of the same color are member of the same community. The user can then interactively select or manipulate a vertex (respectively a group of vertices) using the cursor. The informations for each node (id, degree, attributes) are displayed when a vertex is pointed.

The community dynamics (see Figs. 24, 25, 26) are available through the *Community Dynamics* tabs, in the central part of the user interface. It displays the size and the evolution of the different communities in the sequence of graphs according to the macro-dynamic operations (split, merge and migrate).

Graph measures

Several measures, listed in Table 3, such as modularity or homophily are computed on each graph of the dynamic network to describe its properties, notably **P1**, **P2**, **P3**, **P4** and **P5** detailed in Sect. 2. The changes in these different measures on the sequence of graphs is presented at the bottom of the interface as Fig. 9 shows.

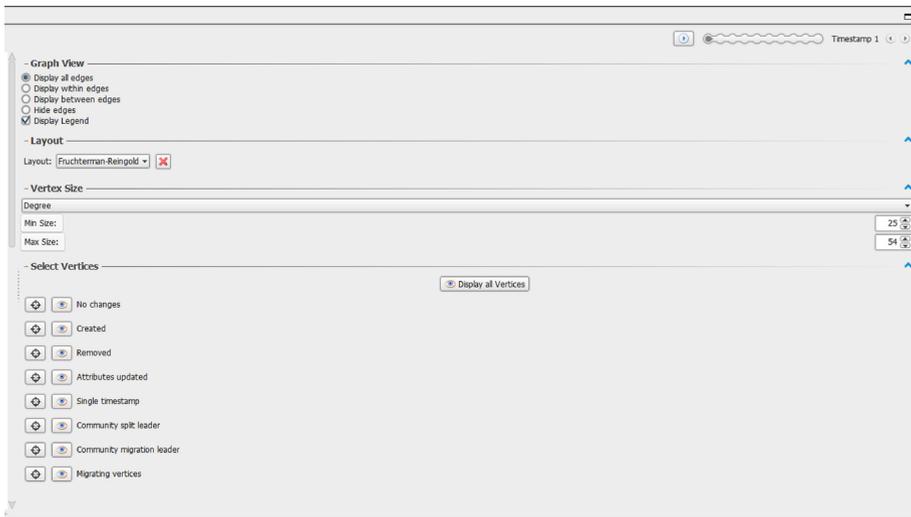


Fig. 32 Graph options panel

Attribute measures

- *Observed homophily* : Ratio of edges connecting similar vertices w.r.t. their attribute values;
- *Expected homophily* : Ratio of pair of similar vertices among all possible pairs of vertices;
The difference between the expected and observed homophily allows to measure if similar vertices according to the attributes tend to be more connected than dissimilar vertices (cf. **P5**);
- *Within inertia* : Measure of the dispersion of the attribute values inside the communities (cf. **P4**). A low within inertia indicates that the communities are highly homogeneous with regard to the attribute values;

Structural measures

- *Modularity* : gives the partition modularity measure as defined by [28] (cf. **P3**);
- *Average clustering coefficient* : is given as an indication of the transitivity of connections in the network [32];
- *Random clustering coefficient* : gives the clustering coefficient in a Erdős–Renyi random graph having the same number of vertices and edges;
The network average clustering coefficient is a measure of the clustering tendency of the network (cf. **P3**). This observed value can be compared with the expected value computed on a random graph having the same vertex set: An observed value higher than the expected value confirms the community structure;
- *Average degree* : the average number of neighbors of the vertices (cf. **P1**);
- *Average shortest path length* : the average minimum number of hops required to reach two arbitrary vertices (cf. **P2**). It is not computed when the graph is formed by several disconnected components (i.e., $E_{btw}^{\max} = 0$);
- *Diameter* : length of the longest shortest path between any pair of vertices (cf. **P2**);

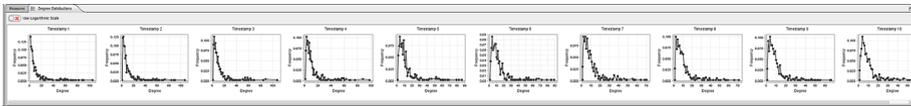


Fig. 33 Degree distribution panel

- *Nb. edges between* : number of edges connecting two vertices belonging to different communities;
- *Nb. edges within* : number of edges connecting two vertices belonging to the same community (cf. **P3**);
- *Nb. edges* : total number of edges in the graph, i.e., \mathcal{E} .

Degree distribution

The bottom of the user interface includes also a panel displaying the distribution of vertex degrees on each graph of the sequence as shown in Fig. 33.

Table 5 Predefined benchmark profiles

Parameter setting	Link-based communities	Attribute-based communities
<p>Configuration 1</p> <p>$P_{RandomCommunity} = 0$</p> <p>$p = 2, \sigma_1 = 20, \sigma_2 = 20$</p> <p>$E_{wth}^{max} = 20$</p> <p>$E_{btw}^{max} = 2$</p> <p>$P_{micro} = 0.1$</p> <p>$R_{removeVert} = 0.1$</p> <p>$R_{addVert} = 0.1$</p>	<p>Strong</p> <p>(Modularity = 0.64 ± 0.008)</p>	<p>Strong</p> <p>(Within inertia = 0.36 ± 0.01)</p>
<p>Configuration 2</p> <p>$P_{RandomCommunity} = 0.8$</p> <p>$p = 2, \sigma_1 = 2, \sigma_2 = 2$</p> <p>$E_{wth}^{max} = 20, E_{btw}^{max} = 2$</p>	<p>Strong</p> <p>(Modularity = 0.66 ± 0.005)</p>	<p>Weak</p> <p>(Within inertia = 0.98 ± 0.001)</p>
<p>Configuration 3</p> <p>$P_{RandomCommunity} = 0$</p> <p>$p = 2, \sigma_1 = 20, \sigma_2 = 20$</p> <p>$E_{wth}^{max} = 20, E_{btw}^{max} = 19$</p> <p>$P_{micro} = 0.3$</p> <p>$R_{addBtwEdge} = 0.7$</p>	<p>Weak</p> <p>(Modularity = 0.30 ± 0.03)</p>	<p>Strong</p> <p>(Within inertia = 0.37 ± 0.01)</p>
<p>Configuration 4</p> <p>$P_{RandomCommunity} = 0.8$</p> <p>$p = 2, \sigma_1 = 2, \sigma_2 = 2$</p> <p>$E_{wth}^{max} = 20, E_{btw}^{max} = 19$</p> <p>$P_{micro} = 0.3$</p> <p>$R_{addBtwEdge} = 0.7$</p>	<p>Weak</p> <p>(Modularity = 0.32 ± 0.06)</p>	<p>Weak</p> <p>(Within inertia = 0.98 ± 0.003)</p>

Output files

The generated dynamic network can be saved as a collection of files, one for each time step, under the **out** directory located in the same working directory as the generator. For each graph of the sequence, the file with the extension “.graph” indicates the composition of the graph (vertices and edges), and the “parameters” file enumerates all the parameters used by the generator.

- **Parameters** : The parameters are output in a separated file. Each line starts by the parameter name and its value.
- **Vertices** : In the graph file, the vertices section starts with the line *# Vertices*. Each consecutive line describes a vertex. A line consists of an integer corresponding to the vertex *id*, the list of its attribute values separated by “;” and an integer corresponding to the vertex community *id*.
- **Edges** : This section starts with the line *# Edges*. Each consecutive line corresponds to an edge. A line is composed of two vertex ids separated by a “;”.
- **Measures** : the measures are saved in a separated file. Each line gives the measure name and its consecutive values at each time step.

Appendix 3: Benchmark profiles

Table 5 presents a first network (Configuration 1 obtained with parameters given in Table 6), having a good community structure according to the relationships and the attributes and then three other networks in which the link-based structure or the attribute-based structure or the both are weakened. Table 7 presents modifications related to the dynamicity of the first network. The parameter setting is given for each network as well as its characteristics (modularity and within inertia).

Table 6 Default parameters

Default parameters

$seed = 60713489427403$
 $K = 5, N = 2000, NbRep = 10, T = 10$
 $P_{randomCommunity} = 0$
 $p = 2, \sigma_1 = 20, \sigma_2 = 20$
 $E_{wth}^{max} = 20, E_{brw}^{max} = 2, MTE = 2000$
 $P_{micro} = 0.1$
 $R_{removeVert} = 0.1, R_{addVert} = 0.1$
 $R_{updatedAttributes} = 0$
 $R_{removeWithEdge} = 0, R_{addWithEdge} = 0$
 $R_{removeBrwEdge} = 0, R_{addBrwEdge} = 0$
 $P_{migrate} = P_{merge} = P_{split} = 0$

Table 7 Benchmark profiles derived from configuration 1 by changing dynamicity

	Dynamic micro	Dynamic macro
<i>Configuration 1</i>		
Link-based communities strong	Weak	Weak
$Modularity = 0.64 \pm 0.008$	$P_{micro} = 0.1$ $R_{removeVert} = 0.1$	$P_{migrate} = 0$ $P_{merge} = 0$
Attribute-based communities strong	$R_{addVert} = 0.1$ $R_{updatedAttributes} = 0$ $R_{addWthEdge} = 0$ $R_{removeWthEdge} = 0$ $R_{removeBtwEdge} = 0$ $R_{addBtwEdge} = 0$	$P_{split} = 0$
$Within Inertia = 0.36 \pm 0.01$		
<i>Configuration 1.2</i>		
Link-based communities strong	Strong	Strong
$Modularity = 0.63 \pm 0.01$	$P_{micro} = 0.5$ $R_{removeVert} = 0.3$	$P_{migrate} = 0.3$ $P_{merge} = 0.3$
Attribute-based communities strong	$R_{addVert} = 0.2$ $R_{updatedAttributes} = 0.3$ $R_{addWthEdge} = 0.3$ $R_{removeWthEdge} = 0.1$ $R_{removeBtwEdge} = 0.3$ $R_{addBtwEdge} = 0.1$	$P_{split} = 0.3$
$Within Inertia = 0.38 \pm 0.02$		
<i>Configuration 1.3</i>		
Link-based communities strong	Weak	Strong
$Modularity = 0.63 \pm 0.01$	$P_{micro} = 0.1$ $R_{removeVert} = 0.1$	$P_{migrate} = 0.3$ $P_{merge} = 0.3$
Attribute-based communities strong	$R_{addVert} = 0.1$ $R_{updatedAttributes} = 0$ $R_{addWthEdge} = 0$ $R_{removeWthEdge} = 0$ $R_{removeBtwEdge} = 0$ $R_{addBtwEdge} = 0$	$P_{split} = 0.3$
$Within Inertia = 0.42 \pm 0.06$		
<i>Configuration 1.4</i>		
Link-based communities (strong)	Strong	Weak
$Modularity = 0.68 \pm 0.04$	$P_{micro} = 0.5$ $R_{removeVert} = 0.3$	$P_{migrate} = 0$ $P_{merge} = 0$
Attribute-based communities strong	$R_{addVert} = 0.2$ $R_{updatedAttributes} = 0.2$ $R_{addWthEdge} = 0.3$ $R_{removeWthEdge} = 0$ $R_{removeBtwEdge} = 0.3$ $R_{addBtwEdge} = 0$	$P_{split} = 0$
$Within Inertia = 0.36 \pm 0.01$		

References

1. Akoglu L, Faloutsos C (2009) RTG: a recursive realistic graph generator using random typing. *Data Min Knowl Discov* 19(2):194–209
2. Akoglu L et al (2008) RTM: laws and a recursive generator for weighted time-evolving graphs. In: Eighth IEEE international conference on data mining, 2008 (ICDM'08). IEEE, pp 701–706
3. Albert R, Barabási A-L (2002) Statistical mechanics of complex networks. *Rev Mod Phys* 74(1):47–97
4. Amaral LAN et al (2000) Classes of small-world networks. *Proc Natl Acad Sci* 97(21):11149–11152
5. Barabási A-L, Albert R (1999) Emergence of scaling in random networks. *Science* 286(5439):509–512
6. Benson AR et al (2014) Learning multifractal structure in large networks. In: Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining. ACM, pp 1326–1335
7. Chung F, Lu L (2002) The average distances in random graphs with given expected degrees. *Proc Natl Acad Sci* 99(25):15879–15882
8. Dang TA (2012) Analysis of communities in social networks. Ph.D. thesis, Université Paris 13
9. Easley D, Kleinberg J (2010) *Networks, crowds and markets: reasoning about a highly connected world*. Cambridge University Press, Cambridge
10. Erdős P, Rényi A (1960) On the evolution of random graphs. *Publ Math Inst Hung Acad Sci* 5:17–61
11. Girvan M, Newman ME (2002) Community structure in social and biological networks. *Proc Natl Acad Sci* 99(12):7821–7826
12. Gong NZ et al (2012) Evolution of social-attribute networks: measurements, modeling, and implications using Google+. In: ACM conference on internet measurement conference (IMC). ACM, pp 131–144
13. Görke R et al (2012) An efficient generator for clustered dynamic random networks. Springer, Berlin
14. Görke R, Staudt C (2009) A generator for dynamic clustered random graphs. Tech. rep., ITI Wagner, Department of Informatics, Universität Karlsruhe. Informatik, Uni Karlsruhe, TR 2009-7
15. Granell C et al (2015) A benchmark model to assess community structure in evolving networks. *CoRR arXiv:1501.05808*
16. Holland PW, Leinhardt S (1971) Transitivity in structural models of small groups. *Comp Group Stud* 2:107–124
17. Kim M, Leskovec J (2012) Multiplicative attribute graph model of real-world networks. *Internet Math* 8(1–2):113–160
18. Lancichinetti A, Fortunato S (2009) Benchmarks for testing community detection algorithms on directed and weighted graphs with overlapping communities. *Phys Rev E* 80(1):016118
19. Lancichinetti A et al (2008) Benchmark graphs for testing community detection algorithms. *Phys Rev E* 78(4):046110
20. LARGERON C et al (2015) Generating attributed networks with communities. *PLoS ONE* 10(4):e0122777
21. Lazarsfeld PF, Merton RK (1954) Friendship as a social process: a substantive and methodological analysis. *Freedom Control Mod Soc* 18(1):18–66
22. Leskovec J et al (2008) Microscopic evolution of social networks. In: ACM SIGKDD international conference on knowledge discovery and data mining (KDD), pp 462–470
23. Leskovec J et al (2005a) Realistic, mathematically tractable graph generation and evolution, using kronecker multiplication. In: Knowledge discovery in databases: PKDD 2005. Springer, Berlin, pp 133–145
24. Leskovec J et al (2010) Kronecker graphs: an approach to modeling networks. *J Mach Learn Res* 11:985–1042
25. Leskovec J et al (2005b) Graphs over time: densification laws, shrinking diameters and possible explanations. In: Proceedings of the eleventh ACM SIGKDD international conference on Knowledge discovery in data mining. ACM, pp 177–187
26. McPherson M et al (2001) Birds of a feather: homophily in social networks. *Annu Rev Sociol* 27(1):415–444
27. Milgram S (1967) The small-world problem. *Psychol Today* 2:60–67
28. Newman ME (2006) Finding community structure in networks using the eigenvectors of matrices. *Phys Rev E* 74(3):036104
29. Newman MEJ, Girvan M (2004) Finding and evaluating community structure in networks. *Phys Rev E* 69(2):026113
30. Palla G et al (2010) Multifractal network generator. *Proc Natl Acad Sci* 107(17):7640–7645
31. Pfeiffer JJ III et al (2014) Attributed graph models: modeling network structure with correlated attributes. In: Proceedings of the 23rd international conference on World Wide Web. ACM, pp 831–842
32. Watts DJ, Strogatz SH (1998) Collective dynamics of ‘small-world’ networks. *Nature* 393(6684):440–442
33. Wong LH et al (2006) A spatial model for social networks. *Phys A Stat Mech Its Appl* 360(1):99–120



C. Largeton is Professor at Jean Monnet University, and she is member of the Data intelligence group at the Hubert Curien Laboratory. She received her Ph.D. in computer science from Claude Bernard University (Lyon—France) in 1991 and then her HDR from Jean Monnet University in 2004. Her main interests include data mining and information retrieval, and her current research focuses on developing methods to efficiently deal with data such as XML documents or social networks. She has served as program committee member of international conferences, and she has been invited as reviewers by several journals (KAIS, Pattern Recognition Letters, etc).



P. N. Mougel holds a Ph.D. in Computer Science from INSA-Lyon (2012). His research interests are mainly related to data mining and its applications, in particular constraint-based approaches for network analysis.



O. Benyahia received a Master degree from Jean Monnet University of Saint-Étienne, France, in 2012. He is currently a Ph.D. student, under the supervision of Christine Largeton, at Hubert Curien Laboratory and Jean Monnet University. His research interests include data mining, graph mining, social network analysis, community detection.



O. R. Zaiane is a Professor in Computing Science at the University of Alberta, Canada, and Scientific Director of the Alberta Innovates Centre for Machine Learning (AICML). He obtained his Ph.D. from Simon Fraser University, Canada, in 1999 under the supervision of Dr. Jiawei Han. He has research interests in data analytics, namely novel data mining algorithms, web mining, text mining, image mining, social network analysis, data visualization and information retrieval with applications in Health Informatics, e-Learning and e-Business. He has published more than 200 papers in refereed international conferences and journals and taught on all six continents. Osmar Zaiane received the ICDM Outstanding Service Award in 2009 and the 2010 ACM SIGKDD Service Award.