A Methodology for Generating Time-Varying Complex Networks with Community Structure

Sandy Porto[1] and Marcos G. Quiles[2]

[1] National Institute for Space Research (INPE), São José dos Campos, Brazil
sandyporto@gmail.com
[2] Institute of Science and Technology, Federal University of São Paulo (UNIFESP), São José dos Campos, Brazil
quiles@unifesp.br

Abstract. There is a demand of benchmark networks for testing community detection algorithms in dynamic scenarios. For generating these benchmarks is necessary to have a methodology able to create controlled networks that simulate the natural behavior of communities over time. This work aims to fill this gap by presenting a methodology for generating dynamic complex networks with community structure. Computer experiments show that our methodology can, starting from an initial network, evolve its communities over time while the overall modular structure of the network is preserved.

Keywords: Dynamic networks, time-varying networks, community detection, benchmark networks.

1 Introduction

A network can be seen as a collection of points connected in parts by lines [1], in which points are called nodes or vertices, and lines named edges or links. Although the study of networks can be traced back to Euler's solution of the Königsberg bridges problem in the XVIII century [2], only recently research in complex networks became a focus of attention giving rise to a new research field named *Network Science* [3,4].

In contrast to the former studies in graph theory, which mainly considered static graphs with regular topologies, the new network science field is more concerned with real networks and their dynamics. Due to their flexibility and unlimited capacity for representing connectivity in real systems, these networks have been used as common framework to model real problems ranging from sociology to physics [5].

Among several topological features that can be extracted from a network, the community structure is a very important one. There is a great effort applied to detect and analyze its community structure. Communities, or modules, can be defined as groups of nodes which are more densely connected with each other, when compared to the rest of the network [6,7]. Detecting such modular structure, as well as their evolution over time, is essential to understanding the network dynamics and also the complex system it represents [8,9].

Due to its importance, several community detection algorithms have been proposed albeit most of them take only static network into account (for a review see [10]). However, real-world networks are not static, but they constantly change their structure over time, thus, those community detection algorithms cannot be straightly applied to these networks [8].

Recently, some community detection models have been proposed to address this limitation (See [11] and Sec. XIII in [10]). However, a new problem has emerged: How to test the accuracy and performance of dynamic algorithms? There are well defined benchmarks, such as the GN [6] and LFR [12] networks, to the static scenario, but there is no published benchmark to evaluate dynamic community detection algorithms.

To fill this gap, here we propose a methodology for generating time-varying networks with community structure. By using our methodology one can build a network in which its structure evolves over time, it means, new communities can rise, several communities can be merged, or even the extinction of a former community. It is worth noting that the process does not occurs instantly, but, according to the parameters setup, the changes are carried out step-by-step dynamically, which is similar to the natural evolution of real networks. In summary, these networks can be used to create a benchmark to test dynamic community detection algorithms in a well controlled scenario. Moreover, these networks can also be used to evaluate general graph-based machine learning algorithms in dynamic situations.

This paper is organized as follows. Section 2 presents our methodology for generating time-varying complex networks with community structure. Computer experiments are shown in Section 3. Finally, Section 4 draws some conclusions about this work.

2 The Methodology

A simple form to evaluate an algorithm regarding to its performance is by using a benchmark. A benchmark can be defined as reference problem in which the expected solution is known a priori. Thus, by benchmarking the algorithm one can straightly compare the algorithm's outcome with the expected result.

In community detection, a benchmark is a graph with a clear community structure that should be recovered by the algorithm being evaluated [10].

In 2008, a methodology for generating static networks with communities of different sizes and with degree distributions that follows a power law was proposed [12]. This methodology, named LFR benchmark, provides a realist scenario to evaluate community detection algorithm using static networks. Thus, here we assume a LFR network as the initial model in which the evolution defined in our methodology takes place.

The original parameters of the LFR benchmark is also used in the proposed methodology: the initial size of the network n, the average degree of the network $\langle k \rangle$, the maximum degree k_{max}, the mixing parameter $\mu \in [0, 1]$, which

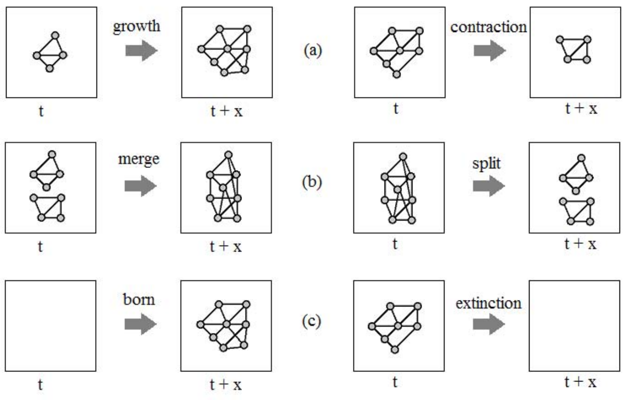

Fig. 1. Possible changes that a community can suffer in a complex dynamic network

controls the fraction of links that a node shares with nodes belonging to other communities. The minimum and maximum size of the communities, s_{min} and s_{max}, respectively. Finally, the exponents τ_1 and τ_2 which controls the degree distribution and size of the communities in the network.

Our methodology aims to create a structure that simulates the behavior of a dynamic network. For this simulation to be successful one must keep in mind the features that we want to explore. For instance, we expect to simulate behaviors commonly observed in real networks, such as social networks.

Studies in social networks have investigated the behavior of communities in dynamic networks. In [13], blogs and bloggers were tracked and the evolution of formed communities were analyzed. In [14], community evolution was investigated in a music application. Networks of scientific publications and links between mobile phones were analyzed in [15].

These three studies have observed the formation of new communities, as well as changes in existing communities. Moreover, their results also seem to form of a consensus on the possible transformations that communities can suffer.

Basically there are three types of transformations, which are illustrated in Fig. 1: (a) one-to-one, which involves the growth or contraction of the community; (b) one-to-many or many-to-one, that are related to the splitting or merging phenomena, i.e. one community can give rise to several or several communities can merge into a single one; (c) one-to-zero or zero-to-one, represented by the birth or termination of a community.

Once identified the possible changes a community can undergo during its evolution in a dynamic network, we propose a methodology composed of six algorithms that simulate these behaviors, named: born, growth, extinction, contraction, merge, and split.

Here, the born function deals with the rising of a new community. The extinction function treats the termination of an existing community in the network. The growth function triggers the growth of an existing community.

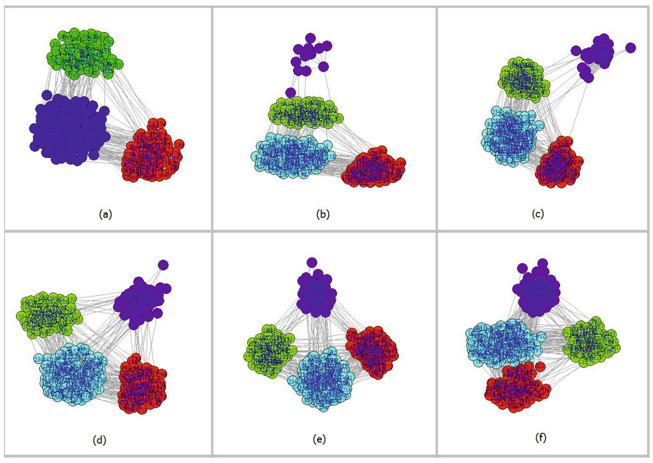

Fig. 2. (Color online) Born Function

The contraction function causes the contraction of a community. Finally, the merge function handles the union of two or more communities in one while the split function does the opposite by dividing of a community into two or more other communities.

From an initial network generated by using the LFR benchmark algorithm, the transformations can be applied sequentially with no overlap, forming a list of graphs representing the evolution of the network over time. At every action of the functions, a copy of the current state of the graph is stored in a list that serves as a timeline of the transformation within the network and will be the output parameter of all functions. Thus, in order to simulate real scenarios, one can define the sequence of actions that will be performed into the network. Moreover, by selecting a single function, an algorithm can be tested over specific situations, i.e. the behavior of the algorithm when the communities start growing or when new communities arises.

All the functions are briefly explained in the following sections. The complete algorithms implemented in R are freely available by request.

2.1 Born Function

The first parameter of the born function is the definition of the new community size. The community size can be explicitly set or automatically defined by the algorithm according to original network characteristics. After that, for each new vertex added to the graph, new edges are formed according to the degree set to this vertex. There are two options while adding new edges: *in* or *out*, *in* for inside edges, or edges linking vertices from the same community; and *out* for outside edges, or edges connecting vertices from distinct communities. The switching between the *in* and *out* modes are controlled by the parameter μ. Finally, after all vertices have been added, additional links are added internally to the community with the purpose of increasing its connection density until it reaches the average connection density of the network. The Born function is summarized in Algorithm 1 depicted in Fig. 4. The $Density(x)$ function, in the algorithm, represents the average density connection in the community or graph x.

Figure 2 shows some steps of the function acting on a network. The original network is shown in (a). In (b) it is possible to observe the initial steps (in purple), when the vertices are not well connected and the new community is still not formed. In (c) and (d) one can notice the early stages of the new community. Finally, Figure (f) depicts the completely rising of the new community.

2.2 Growth Function

The first action is to define which community will undergo the transformation. It can be explicitly informed or randomly defined by the algorithm. Next, a new community size is chosen and new vertices are added to the community until the new size is reached. At each new vertex, edges are also added according to the chosen vertex degree. The edges are added following the same methodology described in the born function by using the *in-out* procedures controlled by the parameter μ. Finally, after all vertices have been created, additional edges are added internally to the community with the purpose of increasing the community connection density. This function is summarized in Algorithm 2 shown in Fig. 4. The $Size(x)$ functions returns the number of vertices in x.

Figure 3 shows some snapshots of the transformation acting on the network shown in (a). In (b)-(d) early stages of new vertices on the community in blue color can be observed. In Figures (e)-(f), extra edges are added albeit hardly observed in the figures.

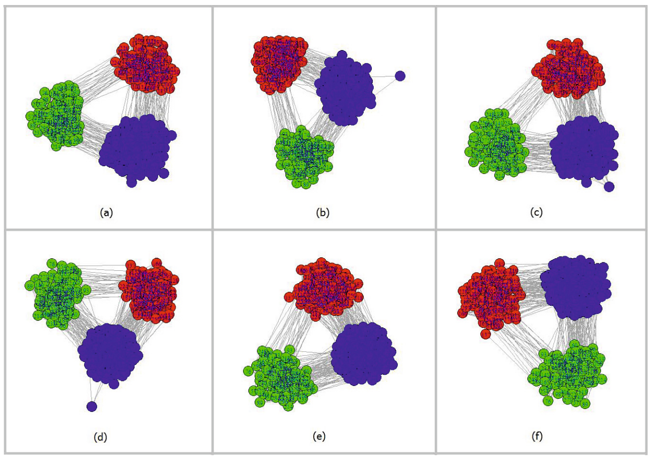

Fig. 3. (Color online) Growth Function

Algorithm 1. Born

```
Input  : Initial graph $g$, $s_{min}$, $s_{max}$, & $\mu$
Output: Graph list: $G_{list}$
begin
    $s_{com}$ = value between $s_{min}$ and $s_{max}$;
    $id_c$ = new community index;
    $G_{list} = \{\}$;
    for $i \leftarrow 1$ to $s_{com}$ do
        if $i == 1$ then
            Add $v_1$ to $g$;
            Assign $v_1$ to community $id_c$;
            $G_{list} = G_{list} + g$;
            Select $v_2$ from $g$ and add $(v_1, v_2)$;
            $G_{list} = G_{list} + g$;
        else
            Add $v_1$ to $g$;
            Assign $v_1$ to community $id_c$;
            $G_{list} = G_{list} + g$;
            $k$ = value between 2 and $i$;
            for $j \leftarrow 1$ to $k$ do
                $link$ = $in$ or $out$ according to $\mu$;
                if $link == out$ then
                    Select $v_2 \notin id_c$
                else
                    if $link == in$ then
                        Select $v2 \in id_c$;
                    end
                end
                Add $(v1, v2)$ to $g$;
                $G_{list} = G_{list} + g$;
            end
        end
    end
    while $Density(id_c) < Density(g)$ do
        Select $v1$ and $v2 \in id_c$;
        Add $(v1, v2)$ to $g$;
        $G_{list} = G_{list} + g$;
    end
    return $(G_{list})$;
end
```

Algorithm 2. Growth

```
Input  : Initial graph $g$, $s_{max}$, $\mu$, & $id_c$
Output: Graph list: $G_{list}$
begin
    $G_{list} = \{\}$;
    if $id_c == 0$ then
        $id_c$ = select a random community from $g$;
    end

    $s_{start} = Size(id_c)$;
    $s_{end}$ = value between $s_{start}$ and $s_{max}$;

    for $i \leftarrow 1$ to $s_{end} - s_{start}$ do
        Add a new $v1$ to $g$;
        Assign $v1$ to $id_c$;
        $G_{list} = G_{list} + g$;
        $k$ = value between 2 and $s_{start} + i$;
        for $j \leftarrow 1$ to $k$ do
            $link$ = $in$ or $out$ according to $\mu$;
            if $link == out$ then
                Select $v_2 \notin id_c$
            else
                if $link == in$ then
                    Select $v2 \in id_c$;
                end
            end
            Add $(v1, v2)$ to $g$;
            $G_{list} = G_{list} + g$;
        end
    end

    while $Density(id_c) < Density(g)$ do
        Select $v1$ and $v2 \in id_c$;
        Add $(v1, v2)$ to $g$;
        $G_{list} = G_{list} + g$;
    end
    return $(G_{list})$;
end
```

Fig. 4. Functions Born and Growth

2.3 Extinction Function

In the extinction function, the first action is to choose the community that will undergo the transformation. After that, each vertex of the community is deleted in a random order along with their edges, until all vertices belonging to the community no longer exits. The process is conducted step-by-step in order to simulate a real scenario. Algorithm 3 in Figure 7 summarize this function.

In Fig. 5 is possible to observe the behavior of the network during the transformation. (a) shows the original network. In (b)-(d), it is already possible to perceive a lower density of vertices in the green community, which is the

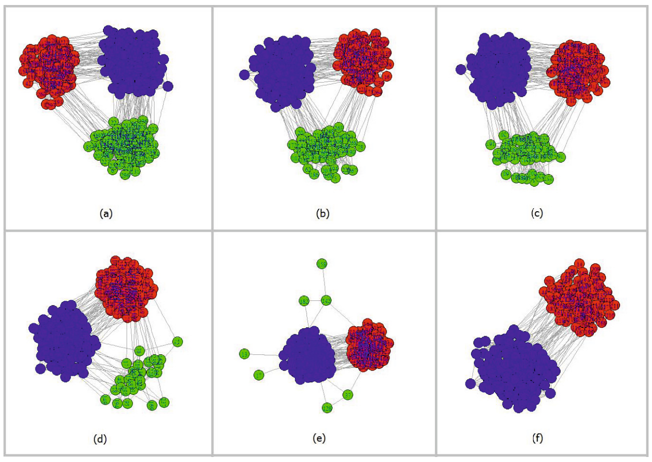

Fig. 5. (Color online) Extinction Function

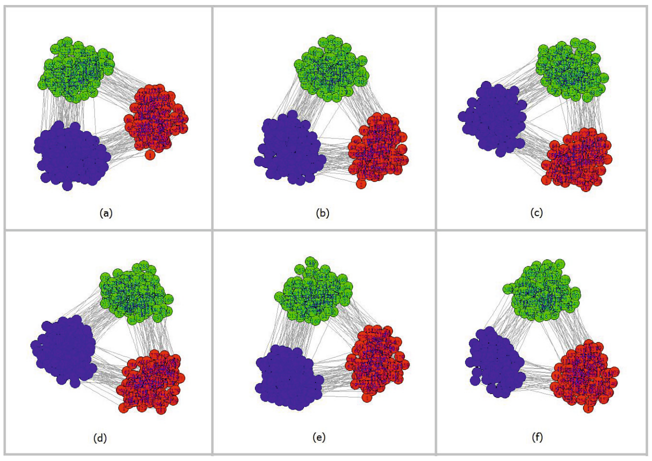

Fig. 6. (Color online) Contraction Function

community undergoing the action. In (e) there is a division of the community into parts due to exclusion of edges connecting them, in some cases the vertices may even become isolated. Finally, in Fig. (f) we can see the complete dissolution of the green community.

2.4 Contraction Function

In the contraction function, once chosen the community that will undergo the action, a new community size is defined. Then, step-by-step, vertices are deleted from the community until the new size is reached. For each deleted vertex and its respectively edges, new internal edges are added to the community to

Algorithm 3. Extinction

```
Input  : Initial graph $g$ & $id_c$
Output: Graph list: $G_{list}$
begin
    $G_{list} = \{\}$;
    if $id_c == 0$ then
        $id_c$ = select a random community from $g$;
    end

    $s_{com} = Size(id_c)$;

    while $s_{com} > 0$ do
        Select a vertex $v$ from community $id_c$;
        Delete all links from $v$;
        Delete vertex $v$;
        $G_{list} = G_{list} + g$;
        $s_{com} = s_{com} - 1$;
    end

    return ($G_{list}$);
end
```

Algorithm 4. Contraction

```
Input  : Initial graph $g$, $s_{min}$, & $id_c$
Output: Graph list: $G_{list}$
begin
    $G_{list} = \{\}$;
    if $id_c == 0$ then
        $id_c$ = select a random community from $g$;
    end
    $s_{start} = Size(id_c)$;
    $s_{end}$ = value between $s_{min}$ and $s_{start}$;
    for $i \leftarrow 1$ to $s_{start}$-$s_{min}$ do
        Select and Delete $v \in id_c$
        $G_{list} = G_{list} + g$;
        while $density\ of\ id_c < g$ do
            Select $v1$ and $v2 \in id_c$;
            Add $(v1, v2)$ to $g$;
            $G_{list} = G_{list} + g$;
        end
    end
    return ($G_{list}$);
end
```

Fig. 7. Functions Extinction and Contraction

maintain the original connection density of the network. The Contraction function is summarized in Algorithm 4 shown in Fig. 7.

In Fig. 6, the transformation steps are displayed. Despite not having any clear movement, one can observe a slight difference in the density of the blue community between steps (a) and (f). There is also a slight decrease in the diameter of community representation, stating that there was indeed a deletion of vertices of the community.

2.5 Merge Function

The selection of the communities that will take part of the merging process is the first step of this function. After, edges are added or redirected into the new community. Two edges that are internal to the former groups of vertices are selected and are switched; in such a manner that after the switching the edges will connect the merging groups. Edges can also be added by simply connecting vertices belonging to different communities. These two options are repeated until the density of the community reaches a threshold consistent with the average density of the network. Algorithm 5 depicted in Fig 10 presents the Merge function. There, the $Links(x)$ function, returns the number of links inside the community or graph x.

Some snapshots of this function are presented in Fig. 8. In Fig. (a) despite the blue community is considered unique, it is clear to the observer that there are actually two groups of vertices. In Figures (b)-(d), edges are added or redirected among those blue groups making the density of connection between them to

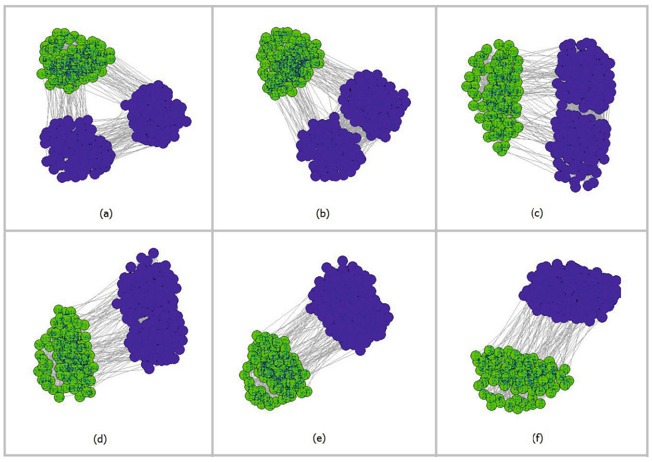

Fig. 8. (Color online) Merge Function

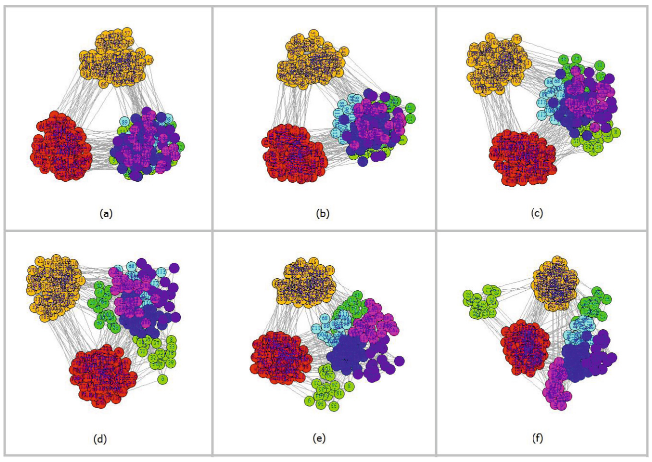

Fig. 9. (Color online) Split Function

approach their intra-connections density. Finally, in Figs. (e)-(f) one can observe a unique community.

2.6 Split Function

The split function is responsible for dividing a single community into different smaller communities. The first step lies in selecting the community that will undergo the process. Second, we need to define in how many parts the original community will be divided. These two parameters: which community and the number of new communities can be defined by the user or automatically set by the algorithm. The vertices of the former community are randomly distributed

Algorithm 5. Merge

```
Input  : Initial graph $g$, merging
         communities $com_{ids}$, & $probR$
Output: Graph list: $G_{list}$
begin
    $G_{list} = \{\}$;
    if $com_{ids}$ is empty then
        $ncom$ = rand() between 2 and
        max communities;
        $com_{ids} = ncom$ communities
        randomly chosen;
    else
        $ncom = Size(com_{ids})$;
    end
    $id_c$ = select an index in $com_{ids}$;
    Assign $id_c$ to all $v \in \{\cup com_{ids}\}$;
    $R_t = probR * Links(id_c)$;
    $D_t = (1 - probR) * Density(g)$;
    $r = 0$;
    $d = Density(id_c)$;
    $stop = (r \leq R_t) and (d \leq D_t)$;
    while $!stop$ do
        $type$ = add or redirect according
        to $probR$;
        if $type == add$ then
            Select $v_1$ and $v_2 \in id_c$;
            Add link $(v1, v2)$ to $g$;
            $G_{list} = G_{list} + g$;
            $d = Density(id_c)$;
        else
            Select two links $(v_1, v_2)$ and
            $(v_3, v_4)$ belonging to
            distinct former communities
            in $com_{ids}$ ;
            Delete $(v1, v2)$ & $(v3, v4)$;
            Add $(v1, v3)$ & $(v2, v4)$;
            $G_{list} = G_{list} + g$;
            $r = r + 1$;
        end
        $stop = (r \leq R_t) and (d \leq D_t)$;
    end
    return $(G_{list})$;
end
```

Algorithm 6. Split

```
Input  : Initial graph $g$, $id_c$, $parts$, $\mu$, &
         $probR$
Output: Graph list: $G_{list}$
begin
    $G_{list} = \{\}$;
    if $id_c = 0$ then
        $id_c$ = select a random
        community;
    end
    if $parts < 2$ then
        $parts$ = value between 2
        $Size(id_c)/3$;
    end
    $A_t = Links(id_c)$;
    Set new communities indexes
    $\{idcom_1, idcom_2, \cdots, idcom_{parts}\}$;
    Assign each $v \in id_c$ to a new
    community randomly;
    $A_c$ = number of links connecting
    distinct new communities;
    $stop = A_c/A_t < \mu$;
    while $!stop$ do
        $type$ = delete or redirect
        according to $probR$;
        Select two communities $idcom_x$
        e $idcom_y$;
        if $tipo == delete$ then
            Delete a link between
            $idcom_x$ e $idcom_y$;
        else
            Select two links $(v_1, v_2)$ &
            $(v_3, v_4) | v_1, v_3 \in idcom_x$ &
            $v_2, v_4 \in idcom_y$;
            Delete these links;
            Add links $(v_1, v_3)$ & $(v_2, v_4)$
            to $g$;
        end
        $G_{list} = G_{list} + g$;
        Update $A_c$;
        $stop = A_c/A_t < \mu$;
    end
    return $(G_{list})$;
end
```

Fig. 10. Functions Merge and Split

among the new communities. Then edges are deleted or redirected until the distribution of edges is satisfactory according to the parameter μ. Two edges connecting two different communities are switched in such way that, after the switching, they connect vertices of the same community. Edges connecting distinct communities can also be deleted to reach an expected separation according to μ or to reach the average density observed in the network. This function is summarized in Alg. 6 presented in Figure 10.

Figure 9 shows some steps of the splitting transformation acting on the network. In Fig. (a) it is possible to see how the vertices in cyan, green, blue, and pink colors, are distributed in new communities randomly. Those vertices formerly represent a single community in the original network. In Figs. (b)-(e) it is observed the separation of the new communities. In Figure (e), it is already possible to observe a separation of the new communities albeit still close to each other due to the high density of inter-connections. Finally, Figure (f) depicts the final state.

3 Experiments

As stated in Sec. 2 the outcome of the functions is a list of graphs that form a timeline of changes occurring in the network. Each item in this list is an instant representation of the network state, or a snapshot. The size of this list indicates the number of steps in each transformation.

Then to evaluate the influence of changes in community detection, longitudinal application on successive snapshots was taken into account [15,16,17]. This approach consists of applying an algorithm to detect communities in each graph on the timeline. Here, to evaluate our methodology, several networks and their respective transformations were used as input to four state-of-art community detection algorithms: the Fast Greedy method [18], the Infomap [19,20], the Label Propagation method [21], and the Walktrap [22]. The Normalized Mutual Information (NMI) [7] was used to evaluate the detection accuracy provided by each algorithm. The purpose of these experiments is twofold: 1) to analyze whether the community structure is preserved after the transformations; and 2) to test some state-of-art algorithms using our networks.

In the experiments, the functions of the methodology are applied successively in the following order: born, extinction, growth, contraction, merge and split. All the figures follow the same visual pattern: each function is identified by an specific color: red color represents the function born, yellow indicates the extinction function, the growth function is depicted in green color, cyan color symbolizes the contraction function, blue color indicates the merge function, and finally, magenta color shows the split function.

Four experiments were performed with distinct parameters setup. The following parameters were held constant in all simulations: $\langle k \rangle = 30$, $k_{max} = 60$, $\tau_1 = 2$, $\tau_2 = 1$, $s_{min} = 50$ and $s_{max} = 100$. The size of the network n and the mixing parameter μ were set for each simulation. Specially in the experiments conducted with a low mixing parameter μ, we expect to observe a high accuracy from all the algorithms. Whether a high accuracy is observed it means that the community structure is well recovered by the technique. Thus, we can conclude that the transformations performed by the functions preserve the community structure of the network.

In the first experiment, the following values for the input parameters were used: $n = 300$ and $\mu = 0.05$. This configuration was set in order to generate networks with few and very isolated communities. The NMI calculated from the outcome of the four community detection algorithms is depicted in Figure 11. All algorithms capture the changes in the community structure of the network by providing high NMI values. It can be observed that only the Infomap and the Label Propagation algorithms reach the maximum value after each transformation. The functions merge and split (blue and magenta) were the ones, which

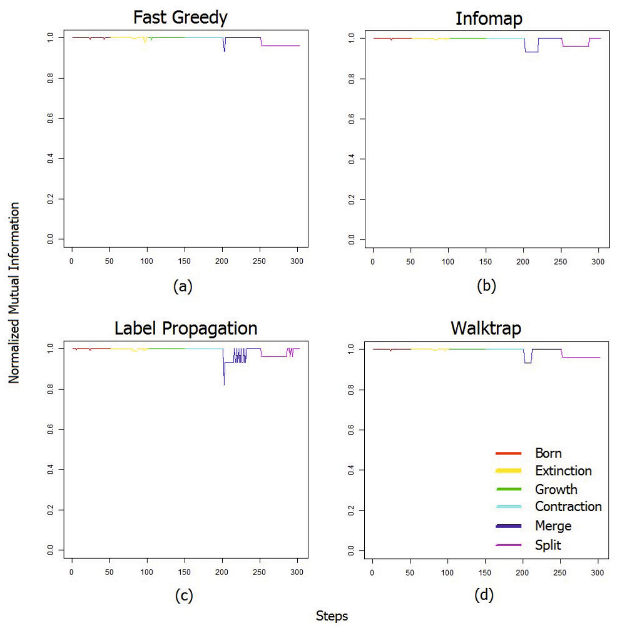

Fig. 11. (Color online) Experiment with $n = 300$ and $\mu = 0.05$

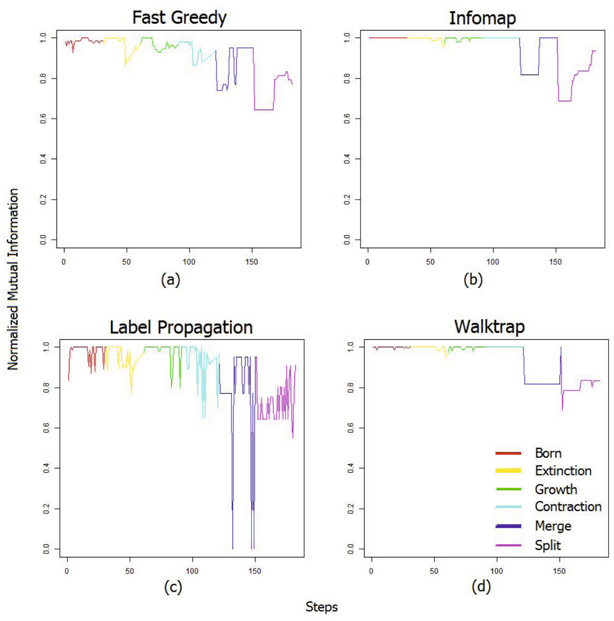

Fig. 12. (Color online) Experiment with $n = 300$ and $\mu = 0.2$

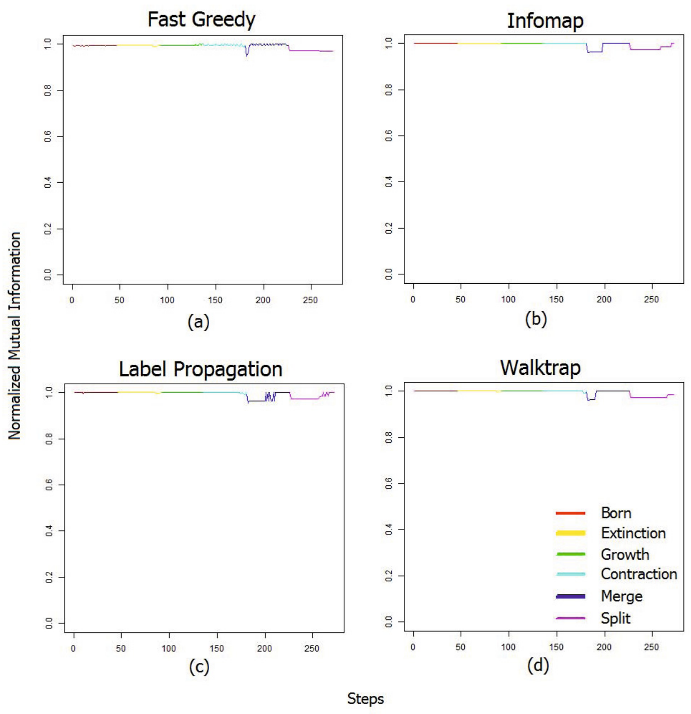

Fig. 13. (Color online) Experiment with $n = 500$ and $\mu = 0.05$

introduced the strongest changes in the network structure, thus, the hardest to be recovered by the community detection algorithm. The performance of the four algorithms was similar, with some slight differences, such as a faster recovery at the merge function in Fast Greedy, and the higher accuracy achieved by the Infomap.

The second experiment assumed $n = 300$ and $\mu = 0.2$ as initial parameters, with the intention of forming a network with a few communities albeit not well isolated from each other. Fig. 12 shows the results of this experiment. Due to the change in the network structure caused by the increase of the mixing parameter μ, the results were quite different from the previous experiment. It means that the network and their respective transformation are harder to be recovered by the algorithms. One can see that the Label Propagation present more unstable results. Again the merge and the split functions are the ones who cause the greatest impact, which can be observed by the variations in the NMI values. The most accurate algorithm in this experiment was the Infomap, which obtained the maximum NMI values during transformations, except in the split function.

The following experiments was conducted with networks set with $n = 500$ and $\mu = 0.05$. Here we evaluate how the community structure is preserved in a larger network with isolated communities. The results of this experiment are depicted

Table 1. Average NMI values achieved by the algorithms in the four experiments

Alghoritm	Fast Greedy	Infomap	Label Propagation	Walktrap
Experiment 1	0.9925355	0.9912771	0.9894647	0.9909344
Experiment 2	0.9069037	0.9484890	0.8853985	0.9366107
Experiment 3	0.9919887	0.9943170	0.9933527	0.9946506
Experiment 4	0.9449553	0.9843080	0.9828750	0.9848928

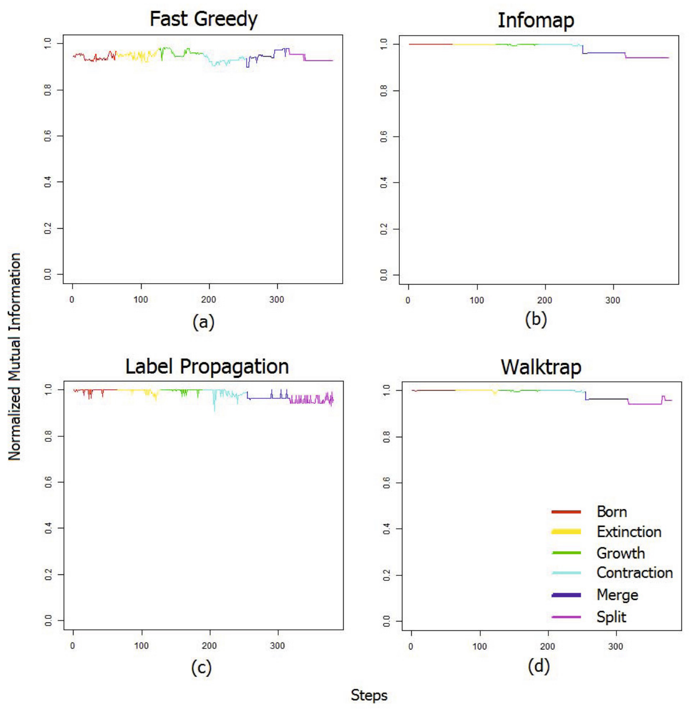

Fig. 14. (Color online) Experiment with $n = 500$ and $\mu = 0.2$

in Figure 13. The results are quite similar to those obtained in Experiment 1. Again, only Infomap and Label Propagation algorithms have reached the maximum value at the end of the transformations.

The last experiment was conducted with networks of $n = 500$ vertices and the mixing parameter set to $\mu = 0.2$. With this setup, networks with several communities not well separated were generated. Figure 14 shows the principal results obtained in this final experiment. One can observe that after the applications of the marge and split functions, all algorithms we taken into account were not able to achieve the maximum NMI. Moreover, in contrast to the previous experiments, here, the Label Propagation and the Walktrap methods were the ones that achieved the highest accuracy.

Finally, in Table 1, the average normalized mutual information of each algorithm in each experiment is shown. In the first experiment, the Fast Greedy algorithm achieved the best average performance. In the second experiment, the best performance was reached by the Infomap algorithm. In the third and fourth experiments the best results were provided by the Walktrap algorithm.

4 Conclusion

In this paper we have introduced a methodology for generating time-varying complex networks. Our methodology, starting from an initial LFR network, can evolve the network structure by simulating the natural evolution of communities observed in real networks. Specifically, our methodology is composed of six functions responsible for evolving the network, they are: the born, the extinction, the growth, the contraction, the merge, and the split functions.

The main motivation behind this work was the absence of a well-defined benchmark for testing dynamic community detection algorithms. Thus, by using our methodology, one can define a specific set of networks to evaluate and compare, in a controlled scenario, those algorithms.

The experiments performed with four state-of-art algorithms have shown that our methodology, besides evolving the network over time, also kept its community structure. Thus, this work presented a further contribution to the problem of how to evaluate dynamic community detection algorithms.

As a future work we intend to use a set of networks generated with our methodology to test several dynamic community detection algorithms.

Acknowledgments. This research was supported by the São Paulo Research Foundation (FAPESP) and by the Brazilian National Research Council (CNPq).

References

1. Newman, M.E.J.: Communities, modules and large-scale structure in networks. Nature Physics 8(1), 25–31 (2011)
2. Barabási, A.L.: Linked: The New Science of Networks. Perseus Publishing (2002)
3. Newman, M.: Networks: An Introduction. Oxford University Press, USA (2010)
4. Barabási, A.L.: Network Science. Barabasi Lab: on-line (2014)
5. Costa, L.F., Oliveira Jr., O.N., Travieso, G., Rodrigues, F.A., Villas Boas, P.R., Antiqueira, L., Viana, M.P., Rocha, L.E.C.: Analyzing and modeling real-world phenomena with complex networks: a survey of applications. Advances in Physics 60(3), 329–412 (2011)
6. Girvan, M., Newman, M.E.J.: Community structure in social and biological networks. PNAS 99(12), 7821–7826 (2002)
7. Danon, L., Duch, J., Arenas, A., Díaz-guilera, A.: Comparing community structure identification. Journal of Statistical Mechanics: Theory and Experiment 9008, 9008 (2005)
8. Bansal, S., Bhowmick, S., Paymal, P.: Fast community detection for dynamic complex networks. In: da F. Costa, L., Evsukoff, A., Mangioni, G., Menezes, R. (eds.) Complex Networks. CCIS, vol. 116, pp. 196–207. Springer, Heidelberg (2011)

9. Newman, M.E.J., Girvan, M.: Finding and evaluating community structure in networks. Phys. Rev. E 69(026113), 1–15 (2004)
10. Fortunato, S.: Community detection in graphs. Phys. Rep. 486, 75–174 (2010)
11. Quiles, M.G., Zorzal, E.R., Macau, E.E.N.: A dynamic model for community detection in complex networks. In: The International Joint Conference on Neural Networks (IJCNN), pp. 1–8 (2013)
12. Lancichinetti, A., Fortunato, S., Radicchi, F.: Benchmark graphs for testing community detection algorithms. Phys. Rev. E 78(4), 046110 (2008)
13. Lin, Y.R., Sundaram, H., Chi, Y., Tatemura, J., Tseng, B.L.: Blog community discovery and evolution based on mutual awareness expansion. In: Web Intelligence, pp. 48–56. IEEE Computer Society (2007)
14. Schlitter, N., Falkowski, T.: Mining the dynamics of music preferences from a social networking site. In: Proceedings of the International Conference on Advances in Social Network Analysis and Mining (2009)
15. Palla, G., Barabási, A.L., Vicsek, T., Hungary, B.: Quantifying social group evolution. Nature 446 (2007)
16. Asur, S., Parthasarathy, S., Ucar, D.: An event-based framework for characterizing the evolutionary behavior of interaction graphs. TKDD 3(4) (2009)
17. Kim, M.S., Han, J.: A particle-and-density based evolutionary clustering method for dynamic networks. Proc. VLDB Endow. 2(1), 622–633 (2009)
18. Clauset, A., Newman, M.E.J., Moore, C.: Finding community structure in very large networks. Phys. Rev. E 70, 066111 (2004)
19. Rosvall, M., Axelsson, D., Bergstrom, C.T.: The map equation. The European Physical Journal Special Topics 178, 13–23 (2009)
20. Rosvall, M., Bergstrom, C.T.: Maps of random walks on complex networks reveal community structure. Proceedings of the National Academy of Sciences 105(4), 1118–1123 (2008)
21. Raghavan, U.N., Albert, R., Kumara, S.: Near linear time algorithm to detect community structures in large-scale networks. Phys. Rev. E 76(036106), 1–11 (2007)
22. Pons, P., Latapy, M.: Computing communities in large networks using random walks. J. Graph Algorithms Appl. 10(2), 191–218 (2006)