Benchmark Generator for Dynamic Overlapping Communities in Networks

Neha Sengupta
Department of Computer Science
Indian Institute of Technology Delhi
New Delhi, India
Email: neha.sengupta@cse.iitd.ac.in

Michael Hamann
Department of Informatics
Karlsruhe Institute of Technology
Karlsruhe, Germany
Email: michael.hamann@kit.edu

Dorothea Wagner
Department of Informatics
Karlsruhe Institute of Technology
Karlsruhe, Germany
Email: dorothea.wagner@kit.edu

***Abstract*—We describe a dynamic graph generator with overlapping communities that is capable of simulating community scale events while at the same time maintaining crucial graph properties. Such a benchmark generator is useful to measure and compare the responsiveness and efficiency of dynamic community detection algorithms. Since the generator allows the user to tune multiple parameters, it can also be used to test the robustness of a community detection algorithm across a spectrum of inputs. In an experimental evaluation, we demonstrate the generator's performance and show that graph properties are indeed maintained over time. Further, we show that standard community detection algorithms are able to find the generated community structure.**

To the best of our knowledge, this is the first time that all of the above have been combined into one benchmark generator, and this work constitutes an important building block for the development of efficient and reliable dynamic, overlapping community detection algorithms.

***Index Terms*—Clustering; Social networks;**

I. Introduction

Many graph based applications require identifying densely linked subsets of nodes in the network, commonly known as communities. There has been a large and diverse amount of work in the area of community detection on graphs [1], [2], [3], [4]. However, a large portion of the existing literature overlooks at least one of two key aspects of a multitude of real world graphs, (a) their dynamic nature, i.e. edges and nodes keep getting added and deleted and (b) the often observed highly overlapping and complex structure of such networks [5], [6].

Recently proposed approaches have identified the above problems and provide solutions for detecting overlapping communities in temporal graphs [7], [8], [9]. However, it is difficult to empirically evaluate and compare these methods due to the lack of a realistic and fast benchmark network data generator or real-world data sets with reliably labeled, dynamic ground truth communities. While real-world data sets might provide important insights, most of the time such data is either unavailable e.g. for privacy reasons, or does not contain reliable ground truth data to compare the found communities against. Benchmark graph generators allow to evaluate the behavior of community detection algorithms on graphs with different, predefined properties and thus test the robustness of the algorithm against a well-defined set of ground truth communities.

The requirements of a benchmark graph generator in such a scenario are diverse. Existing generators for static benchmark graphs such as LFR [10] and CKB [11] replicate properties of real-world networks like that node degrees and community sizes follow power law distributions. CKB additionally ensures that the number of communities a node belongs to follows a power law distribution and has a positive correlation with the node degrees. Our goal is to extend the CKB model with a dynamic component that simulates changes in the community structure while *maintaining* these graph properties.

It is commonly agreed on [12], [13], [14], [5], that the evolution of communities can be characterized by the fundamental events birth, death, merge, split, expansion and contraction. The challenge is that such community scale events also affect the properties of the graph. For example when a new community appears, many edges need to be added to the nodes of the community which might distort the degree distribution. An analogous problem comes up when communities cease to exist, two communities merge into a single community or one community splits into two individual communities [5]. Moreover, since nodes leave or join communities, it also has the potential to affect the community size distribution. Further, we want to allow fine-grained control over the rapidity with which events take place in the generated graph as most communities in a real-world setting evolve gradually over time [15]. A community detection algorithm must thus be able to follow smaller changes in order to detect large-scale changes in the community structure. This can be used to evaluate the sensitivity of different community detection algorithms. It has been shown that all of these events can also be observed in real-world networks [5]. Therefore, any dynamic community detection algorithm that is supposed to work on non-trivial real-world graphs needs to be able to detect at least these basic events.

A. Our Contribution

In this work we extend the CKB benchmark graph generator for overlapping community structures to generate communities that are evolving over time. Our generator simulates community events like birth, merge, split, death, expansion,

2374-8486/17 $31.00 © 2017 IEEE
DOI 10.1109/ICDM.2017.51

IEEE computer society

Authorized licensed use limited to: KIT Library. Downloaded on October 19,2021 at 08:29:20 UTC from IEEE Xplore. Restrictions apply.

and contraction which result in node and edge insertions and deletions that gradually change the graph. A set of parameters allows to control various properties of the graph as well as the speed of the dynamic process. In our model, at every time step, each community event can occur with a certain probability. However, our generator can also be easily adapted to follow a different pattern of events.

We show using empirical analysis that the graph generator is fast and produces graphs that actually maintain the properties of the CKB model over time, i.e., the node degrees, the number of communities per node and the community sizes follow power law distributions. Further, we show that we achieve a realistic average local clustering coefficient over time. Also, we show using standard, existing community detection algorithms that our generator produces graphs whose link structure reflects its ground truth community assignment over time and that it is capable of differentiating between different community detection algorithms.

Section II describes the prior work, while Section III summarizes the notations used in this paper. Section IV describes the algorithm in detail and Section V presents the experimental evaluation of the generation. The code for the benchmark generator is publicly available at https://github.com/senguptaneha/DynamicBenchmarkGenerator.

II. Related Work

Studying random graph models has a long tradition starting with the simplest model introduced by Gilbert [16] where every edge is present with the same uniform probability, commonly known as Erdős-Rényi graphs. Closely related to this model is the planted partition model [17], where additionally a disjoint ground truth community structure is generated and edges within communities are present with a higher probability than the remaining edges. These simple models lack many properties of real-world networks like a power law degree distribution, though. As an attempt to overcome these limitations, the LFR generator has been introduced [10], [18]. It features a power law degree distribution and power law community sizes and has variants for generating overlapping communities as well as directed and weighted graphs. Today, the LFR benchmark is widely used to evaluate the performance of community detection algorithms.

Concerning overlapping communities, the model of LFR has been criticized though that overlaps between communities are sparser than non-overlapping parts as for real word networks it has been shown overlaps are dense and that the number of communities per node follows a power law distribution [4], [6]. The proposed AGM model [4], [6] reflects this by modeling each community as an Erdős-Rényi random graph which naturally leads to dense overlaps between communities. They show that from such an overlapping community structure naturally many properties like clustering coefficients are similar to real-world networks. The CKB generator [11] is based on the AGM model, however instead of assuming a given community structure it presents a random model for generating them such that the number of communities per node as well as the community sizes follow power law distributions. Moreover, the observation in [6] of larger communities being less tightly knit than smaller communities is accounted for in the graphs generated by this algorithm. As a result, we selected this approach for our static graph generation module.

For evolving networks, there is no widely used model available and those available are limited to non-overlapping communities. It is widely agreed on [12], [13], [14], [5] though, that the evolution of dynamic communities can be characterized by the following fundamental events: *birth, death, merging, splitting, expansion,* and *contraction*. In [5], they describe an algorithm for tracking the evolution of a given community in a dynamic graph. For the empirical evaluation, they generate synthetic LFR graphs [18] along with embedded community events of each of the above types that are applied in rapid changes of the graph. In [19], a different model is used for the evaluation of a dynamic community detection algorithm where nodes are initially randomly assigned to a set of k communities and some nodes change their membership in each time step. Based on the planted partition model, [20] describes an algorithm for generating a dynamic graph with non-overlapping clusters that also features the above-mentioned event types, they are slowly applied using random changes over several time steps. In a more recent work, [21] introduces another benchmark model again based on the planted partition model but only considering grow-shrink and merge-split operations. They propose three benchmarks that feature either one or both of the operations.

III. Notations

The dynamic graph generator generates a dynamic graph G over $T+1$ time steps, $G_0 \dots G_T$. Each entity in the graph G, i.e. nodes, edges, and communities store the period during which it is active in G. CommunityList(G_t) is the set of communities in graph G_t. We use C to denote the set of communities in graph G over all time steps. For community $c \in C$, n_c is the number of nodes in community c. The set of nodes and edges in c are $c.nodeList$ and $c.edgeList$ respectively. Each community c stores its nodes in a certain (possibly random) order. For a node u belonging in c, $pos(u, c)$ determines the position of u in the list of nodes in c. A community can die only once due to a death, merge, or split event. $0 \leq join(u, c) \leq T$ and $0 \leq leave(u, c) \leq T$ signify the time steps at which a node u joins and leaves c respectively.

$V(G_t)$ is the set of nodes in G at time t. For $u \in V(G)$, x_u is the number of communities that node u belongs to. $u.start$ and $u.end$ are the start and end times of node u. The set of edges incident to u is $adj(u)$, while $comm(u)$ is the set of communities that u belongs to. Similar to communities, nodes may not re-appear after being deleted in G.

An edge in the undirected graph G between nodes u and v is denoted as (u, v). In a graph with overlapping communities, a pair of nodes u and v may be part of many different communities, and may therefore be deleted and re-inserted multiple times due to different community events. The operation $Insert(u, v, c, t)$ generates the edge event that

Authorized licensed use limited to: KIT Library. Downloaded on October 19,2021 at 08:29:20 UTC from IEEE Xplore. Restrictions apply.

inserts edge (u, v) in G at time t as a part of community c only if the edge (u, v) doesn't already exist in G_t. Similarly, the operation $Delete(u, v, c, t)$ generates an edge event that deletes (u, v) from G_t if it is a part of community c.

$PL(x_1, x_2, \beta)$ is the power law distribution with minimum value x_1, maximum value x_2, and exponent β. For a random variable X, $E(X)$ denotes the expected value of X, and $X \sim PL(x_1, x_2, \beta)$ signifies that X is instantiated by drawing a value at random from the corresponding power law distribution. $f(z; \lambda)$ denotes an instantiation of the random variable z from the exponential distribution with coeffecient λ.

IV. Algorithm

We generate a simple, unweighted graph with an overlapping, evolving reference community structure following the model of the CKB generator [11] with parameters as summarized in Table I. In each of the T discrete time steps we maintain its properties:

The number of overlapping communities a node is part of follows a power law distribution $PL(x_1, x_2, \beta_1)$. Community sizes also follow a power law distribution $PL(n_{\min}, n_{\max}, \beta_2)$. Every community is modeled as an Erdős-Renyi graph, i.e., every edge has the same probability of existence [22]. The edge probability is decreasing with increasing community size according to the parameters α and γ. To add noise to the graph, a so-called *Epsilon Community* containing all nodes with edge probability ϵ is added.

Initially, in time step 0, such a graph with N nodes is generated (see section IV-A). In each time step $t \in [1, \ldots, T]$, a *community event*, where a community is born, dies, is merged or split, happens with probability p. Further, with probability p', a *node event*, where individual nodes are added to or removed from the graph, happens. Each community event triggers several *edge events* that are spread over t_{effect} discrete time steps. Further, random edge events change the global epsilon community over time. At each time step, one community event and one node event as well as several edge events may take place (Note that it is straight-forward to extend the generator described here for multiple community events per time step. We impose the limit of one community event per time step only for simplicity of analyses and evaluation). Communities participating in a community event at time t may not be part of another community event until time $t + t_{effect}$, i.e., till all triggered edge events have taken place.

As output, the algorithm produces an initial graph G_0, a stream of node and edge updates and a ground truth community assignment for each time step. From these events, a full graph G_t can be obtained for every time step $t \in [1, \ldots, T]$.

A. Static Graph Generation

Our static graph generation algorithm adapts the algorithm described in [11]. Table I is also an extended version of the list of parameters used in their work. The first part of this table lists the set of parameters along with their recommended values used for the static graph generation. Except for the value of n_{min}, the recommendation for all other parameters is the same as that in [11].

Param.	Meaning	Recomm. Value
N	Number of nodes in G_0	-
T	Number of time steps	-
x_{min}	Minimum node membership	1
x_{max}	Maximum node membership	$N/10$
β_1	Community Membership Exponent	2.5
n_{min}	Minimum size of community	$min(N/100, 20)$
n_{max}	Maximum size of community	$N/10$
β_2	Community Size Exponent	2.5
α	Intra Community Edge Prob. $= \frac{\alpha}{n^\gamma}$	2
γ	Intra Community Edge Prob. $= \frac{\alpha}{n^\gamma}$	0.5
ϵ	Inter Community Edge Probability	$2N^{-1}$
p	Community Event Probability	0.1
p'	Node Event Probability	10^{-2}
λ	Community event sharpness	0.2
t_{effect}	Time steps for community events to take effect	-

TABLE I: Parameters used in the Graph Generator

The value of n_{min} is the size of a majority of the communities in the graph (since community sizes follow the power law distribution). Note that the probability of an intra-community edge is α/n^γ, where n is the number of nodes in the community. For the recommended values for α and γ, the intra-community edge probability evaluates to $2/\sqrt{n}$. For $n \leq 4$, this amounts to a clique. Therefore, n_{min} must be large enough to avoid the presence of a large number of small cliques in the graph qualifying as communities.

a) Node-Community Bigraph: The node-community bigraph is a bipartite graph with vertices corresponding to the nodes and communities in G_0. An edge between vertices u and c in the node-community bigraph indicates that the node u is a member of community c in G_0. The static graph generator begins by generating the node community bigraph. For each node u, we draw x_u, the number of communities that u is a part of, from $PL(x_{min}, x_{max}, \beta_1)$. Let $M_0 = E(M)$, where $M \in PL(n_{min}, n_{max}, \beta_2)$, and $X_0 = E(X)$, where $X \in PL(x_{min}, x_{max}, \beta_1)$, then the number of communities $N_c = \frac{N * X_0}{M_0}$. N_c community sizes are generated using $PL(n_{min}, n_{max}, \beta_2)$. However, the community sizes and node memberships thus generated must satisfy $\sum_u x_u = \sum_c n_c$. We decrement or increment the size of randomly selected communities (without violating the constraints of n_{min}, n_{max}) to satisfy the above. The bi-graph is generated using the configuration model.

b) Intra-Community Edge Generation: The methodology described in [11] for generating the edges within a community involves drawing the number of edges per node to insert from a binomial distribution with success probability $p_c = \frac{\alpha}{{n_c}^\gamma}$ where n_c is the size of the community c, and thereafter using the configuration model. We use the Batagelj Brandes model to more efficiently generate an Erdős-Renyi graph for each community and erase multiple edges.

c) Epsilon Community: The final step in the static graph generator is the generation of global edges. The global edge

Authorized licensed use limited to: KIT Library. Downloaded on October 19,2021 at 08:29:20 UTC from IEEE Xplore. Restrictions apply.

density is given by the parameter ϵ. Therefore, we generate ${}^{N}C_2 \times \epsilon$ random node pairs, and insert a global edge between them (if not already existing).

B. Dynamic Graph Generation

Once the static graph (G_0) has been created, for every timestep t in $1 \leq t \leq T$, a community event is generated with probability p. Effectively, a coin toss with probability of head $= p$ decides if any community event is generated at t, which in turn may result in many different edge addition or deletion events. The community event generated at t is said to start at t. The edge events resulting from it are spread over timesteps t to $t+t_{effect}$. There may be other community events generated until $t + t_{effect}$. As a result, at every timestep, edge events corresponding to different community events can take place. However, at any given timestep, at most one community event is generated.

Dynamic Graph Generation Parameters: The second part of Table I denotes the parameters used in the generation of dynamic events in G. As described above, p denotes the probability with which *any* community event happens at a given timestep. p' denotes the probability with which any node in the graph is added or deleted at any given timestep. λ and t_{effect} are the parameters that control how sharply communities rise and fall. For example, consider what happens when a community is born. In keeping with our intuition that a community can appear slowly, we simulate nodes gradually joining the community. Therefore, if a community is spawned at time t, then the last node to join it does so at time $t+t_{effect}$. For a node u joining a community c at time $t + t'$, where $t' < t_{effect}$, edges joining u with nodes already in c appear slowly starting from $t + t'$.

1) Community Events: We consider 4 types of community events - birth, death, merge, and split. Other events considered in the literature such as community expansion and contraction are subsumed in the events birth and death as described in the following sections. Nevertheless, it is straightforward to add these as separate community events in our model.

- Community Birth: When a community is born, a set of nodes become part of the new community, and the density of edges among these nodes significantly increases. Edges newly inserted due to the birth event are said to be present due to the existence of this community.
- Community Death: When a community 'dies' or dissolves, edges in the graph present due to the existence of only this community are deleted and the edge density among the nodes of this community decreases.
- Community Merge: Two communities (possibly overlapping) may merge at a given timestep. This implies that the density of edges across the two communities increases, until the two communities are not distinguishable individually, but form one larger community.
- Community Split: Analogously to merge, the nodes of a single community may split into two separate ones. The edges in the community that cut across the split boundary are deleted until the original community does not have the required overall edge density to be considered a community.

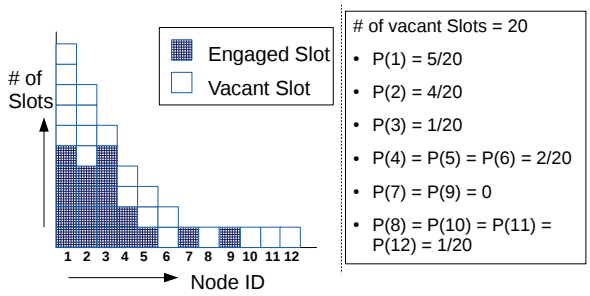


Fig. 1: Slot System for maintaining Node Community Membership distribution

2) Maintaining the Node Membership Distribution: To maintain the power law distribution of node memberships, we employ a *slot system*. Each node u has an associated value X_u, which is the number of communities it may belong to, or the number of slots alloted to node u. In general, we use $X_u = l \times x_u$ where $1 < l < 2$. Over all nodes, the values of X_u follow a power law distribution. Therefore, with each node u, we have X_u slots, of which x_u are filled, where x_u is the number of communities node u actually belongs to. When a node joins a community, an additional slot is occupied, and when a node leaves a community, one of its slots is freed (Figure 1). The number of free slots for u is $X_u - x_u$, which influences the likelihood with which u joins new communities. In subsequent sections, we describe how these slots are used to maintain the desired power law distribution of node community memberships for individual community event types.

After generating the bigraph in the static generator described in IV-A, for each node u we set $X_u = 1.2 \times x_u$. This ensures that at $t = 1$, many nodes are eligible to join at least one community and such new communities may be born while the values of x_u still follow a power law distribution.

A side-effect of using the slot system is that as the number of community events grow, many nodes may emerge that are 'orphaned', in that they are not a part of any community. If n_1 is the number of single-community nodes in G_0, then to ensure that the number of orphaned nodes does not grow with time, one possibility is to *over-sample* such nodes as in [23], i.e. we allocate $l * n_1$ nodes with $X_u = l$ in the slot data structure. In expectation, at each timestep t, there would be n_1 nodes that belong to 1 community, while all nodes orphaned at t would not be a part of G_t. Note that we do not implement this scheme in the experiments for this paper.

3) Community Birth: To spawn a new community c at time t with desired size $n_c \sim PL(n_{min}, n_{max}, \beta_2)$, n_c nodes are sampled from the set of nodes $V(G_t)$, where each node u is sampled with probability proportional to $max(0, X_u - x_u)$ (over the course of time, due to merge and split events, it is

Authorized licensed use limited to: KIT Library. Downloaded on October 19,2021 at 08:29:20 UTC from IEEE Xplore. Restrictions apply.

possible that $x_u > X_u$). For each node u thus sampled, x_u is incremented. The intra-community edges are generated using the Batagelj Brandes model with edge probability $p_c = \frac{\alpha}{{n_c}^{\gamma}}$ [24]. The community birth event is immediately followed by a community expansion phase in this generator where nodes gradually join the initial core.

The first $k \times n_c$ of the nodes forms the core of the community (in our experiments, $k = 0.1$). All nodes other than the core nodes join c *after* time t. For a node i not belonging to the core of the community, the *join time* of i in c ranges from t to $t + t_{effect}$. While edges within the core are inserted at time t, all other edges (u, v) are inserted depending on the join times of u and v. We compute $m(u, v) = max(join(u, c), join(v, c))$, the time by which both u and v have joined c and insert the edge at timestep $(m, v) - f(z; \lambda)$. A formal description of the community birth event is in Algorithm 1.

Parameters λ and t_{effect} can be used to test the sensitivity of a community detection algorithm. For instance, setting $t_{effect} = 1$ would mean that a community birth event causes a large number of edges to be inserted within a single time step. However, if the value of t_{effect} is large, then a sensitive community detection algorithm would be able to detect the community expansion phase in response to the addition of corresponding edges.

Algorithm 1: Community Birth

Algorithm `BirthCommunity(`t`)`
- $c \leftarrow newCommunity()$;
- $n_c \leftarrow draw(PL(n_{min}, n_{max}, \beta_2))$;
- **for** $u \in V(G_t)$ **do**
  - $p(u) \leftarrow max(0, X_u - x_u)$;
- Normalize $p(.)$;
- $c.nodeList \leftarrow$ Sample n_c nodes with $p(.)$;
- $c.edgeList \leftarrow$ Generate edges with prob. $p_c = \alpha / {n_c}^{\gamma}$;
- **for** $u \in c.nodeList$ **do**
  - **if** $pos(u, c) \leq k * n_c$ **then**
    - $join(u, c) = t$;
  - **else**
    - $join(u, c) = t + \frac{pos(u)}{n_c} t_{effect}$;
- **for** $(u, v) \in c.edgeList$ **do**
  - **if** $join(u, c) = t$ *and* $join(v, c) = t$ **then**
    - $Insert(u, v, c, t)$;
  - **else**
    - $m(u, v) \leftarrow max(join(u, c), join(v, c)$;
    - $Insert(u, v, c, m(u, v) - f(z; \lambda))$;

4) Community Death: Symmetrical to the birth event, the community death event is preceded by a contraction phase, where nodes gradually leave the community due to edge deletions until only a core of the community remains.

Algorithm 2: Community Death

Algorithm `DeathCommunity(`c,t`)`
- **for** $u \in c.nodeList$ **do**
  - **if** $pos(u, c) \leq k * n_c$ **then**
    - $leave(u, c) = t$;
  - **else**
    - $leave(u, c) = t + \frac{pos(u,c)}{n_c} t_{effect}$;
- **for** $(u, v) \in c.edgeList$ **do**
  - **if** $leave(u, c) = t$ *and* $leave(v, c) = t$ **then**
    - $Delete(u, v, c, t)$;
  - **else**
    - $m(u, v) \leftarrow min(leave(u, c), leave(v, c))$;
    - $Delete(u, v, c, m(u, v) + f(z; \lambda))$;

For a node i not belonging to the core of c, the $leave$ time of i from c is ranges from t to $t + t_{effect}$. The node i leaving community c is manifested by deleting edges that are incident on i and other nodes in c. Given time at which either u or v leaves c, $m(u, v) = min(leave(u, c), leave(v, c))$, the edge (u, v) is deleted z time steps after $m(u, v)$, where z is determined similarly as in community birth. Edges within the core are all deleted at time t.

5) Community Split: The Community Split event is simulated as the consequence of a gradually vanishing overlap between two communities. Given an existing community c to split into communities c_1 and c_2 at time t, the split algorithm starts by randomly choosing a *split point* s (Figure 2), such that both resulting communities have at least n_{min} nodes.

When the split event begins at time t, both c_1 and c_2 include all nodes that were in c and overlap entirely. Until $t + t_{effect}$, nodes to the left of s leave c_2, and nodes to the right of s leave c_1, thereby simulating a vanishing overlap. In Figure 2, x_1 is the first node and y_1 the last node to leave c_2. Converting the vanishing overlap to edge events, if the edge (x_1, y_2) exists, it must be deleted first since they are the first nodes to 'leave' the overlap between c_1 and c_2, while an edge (y_1, x_2) if existing, must be deleted last. In the case that node u ends up in c_1 and v in c_2, we define $m(u, v)$ as $(leave(u, c_2) + leave(v, c_1))/2$ and the edge (u, v) is deleted at time $m(u, v) + f(z; \lambda)$.

While the above procedure suffices to split the nodes of the community c into c_1 and c_2, there remains the difference in the intra-community edge probabilities. In particular, since $n_{c_1} < n_c$ and $n_{c_2} < n_c$, $p_{c_1} > p_c$ and $p_{c_2} > p_c$.

For community c_1, $p_1 = \alpha / n_1^{\gamma}$. Let $p_0 = |E(c_1)| / {}^{n_1}C_2$, where $E(c_1)$ is the set of edges in c_1 after the split. We must generate edges in c_1 such that edge probability in c_1 is p_1. To account for already existing edges, we generate edges in c_1 with edge probability $p_{new} = (p_1 - p_0)(1 + p_0)$.

6) Community Merge: The merge event follows an almost reverse process to that of the split event, whereby it is simulated as a consequence of a growing overlap between two communities. Given communities c_1 and c_2 to merge at time t,

Authorized licensed use limited to: KIT Library. Downloaded on October 19,2021 at 08:29:20 UTC from IEEE Xplore. Restrictions apply.

Algorithm 3: Community Split

Algorithm `SplitCommunity(c,t)`
- Generate split point s;
- **for** $u \in c.nodeList$ **do**
  - $c_1.nodeList.add(u)$;
  - $c_2.nodeList.add(u)$;
  - $join(u, c_1) = join(u, c_2) = leave(u, c) = t$;
  - **if** $pos(u, c) \leq s$ **then**
    - $leave(u, c_2) = t + \left(\frac{pos(u,c)}{n_c}\right) * t_{effect}$;
  - **else**
    - $leave(u, c_1) = t + \left(\frac{n_c - pos(u,c)}{n_c}\right) * t_{effect}$;
- **for** $(u, v) \in c.edgeList$ **do**
  - **if** *($pos(u, c) \leq s$ and $pos(v, c) > s$)or($pos(u, c) > s$ and $pos(v, c) \leq s$)* **then**
    - $m(u, v) \leftarrow (leave(u, c) + leave(v, c))/2$;
    - $Delete(u, v, c, m(u, v) - f(z; \lambda))$;
- Generate extra edges for c_1 and c_2;

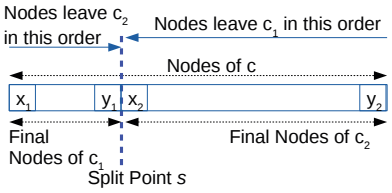


Fig. 2: Splitting community c into c_1 and c_2

the nodes of c_1 start joining c_2, while the nodes of c_2 join c_1. The community c is born at time $t + t_{effect}$, when the overlap between c_1 and c_2 has grown to its full extent. To generate the corresponding edge events, for each pair of nodes (u, v) such that before time t u existed only in c_1 and v only in c_2, we insert an edge with probability p', where p' is the edge probability required to achieve an internal edge probability of α/n^{γ} in c, where $n = n_{c_1} + n_{c_2}$. Each newly generated edge (u, v) is inserted according to the join times of the nodes in the other community. Figure 3 contrasts the process of growing and shrinking overlap for the Split and the Merge events.

7) Dynamics in the Epsilon Community: The epsilon community edges, also called the global edges in the graph undergo some deletions and additions in our model. Our goal is thereby to keep the overall number of global edges in the graph the same while also generating deletion and insertion events of these edges at randomly selected points in time. For this, we double the amount of edges generated for the epsilon community in G_0 (see section IV-A). We split these edges into two parts and pair every edge of the first part with a unique edge of the second part. Of each such pair (e_1, e_2), we sample a random switch time $t_s \in 1, \ldots, T$ at which e_1 is deleted and e_2 is inserted.

Algorithm 4: Community Merge

Algorithm `MergeCommunity(c1,c2,t)`
- $L = n_{c_1}$;
- $R = n_{c_2}$;
- **for** $u \in c_1.nodeList$ *and* $pos(u, c_1) \leq L$ **do**
  - $c.nodeList.add(u)$;
  - $c_2.nodeList.add(u)$;
  - $leave(u, c_1) = leave(u, c_2) = join(u, c) = t + t_{effect}$;
  - $join(u, c_2) = t + \left(\frac{pos(u,c_1)}{L}\right) * t_{effect}$;
- `/* Repeat for` $u \in c_2.nodeList$ `*/`
- $p_m \leftarrow$ Compute Merge Edge Probability(c_1, c_2);
- **for** $u \in c_1.nodeList$ *and* $pos(u, c_1) \leq L$ **do**
  - **for** $v \in c_2.nodeList$ *and* $pos(v, c_2) \leq R$ **do**
    - Generate Edge (u, v) with probability p_m;
    - **if** *Edge* (u, v) *not active* **then**
      - $m(u, v) \leftarrow (join(u, c_2) + join(v, c_1))/2$;
      - $Insert(u, v, c, m(u, v) + f(z; \lambda))$

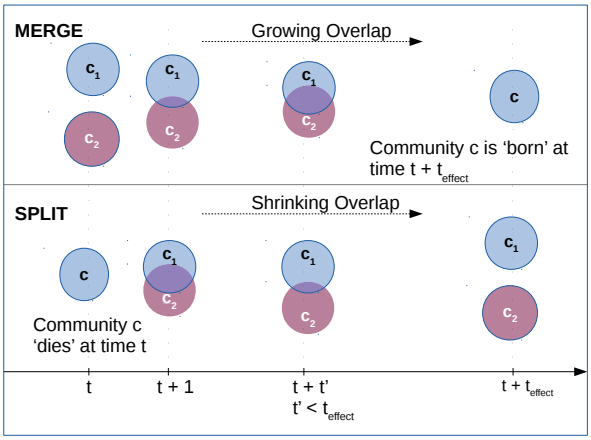


Fig. 3: Split and Merge of Communities

8) Node Events: Node events, i.e. add nodes and remove nodes, are generated with a smaller probability p'.

a) Add Node: When a node u is added to G at time t, we set $X_u = 1.2 \times X$, where $X \sim PL(x_{min}, x_{max}, \beta_1)$ and node u with X_u vacant slots is added to the set of slots (Section IV-B). Finally, ϵ edges incident on u are generated.

b) Delete Node: When a node u is to be deleted from G at t, we set $X_u = x_u = 0$. u leaves each active community it belongs to. If the number of vertices in a community c falls below n_{min} due to this operation, then c dies altogether. Finally, each active edge incident on u is deleted.

9) Conversion of G_T to G_0 and Graph Stream: Once the graph G_T has been generated, the algorithm must convert it into the initial graph G_0 and the associated stream of edge and node events. G_0 is the set of nodes and edges that are active at time 0. For each node u in G, if u is not active

Authorized licensed use limited to: KIT Library. Downloaded on October 19,2021 at 08:29:20 UTC from IEEE Xplore. Restrictions apply.

in G_0 (or in G_T), we must append the corresponding node add (or delete) event to the stream. Similarly, for each edge (u, v) in G, we collect the edge events generated for (u, v). Since the same edge may be added or deleted due to events pertaining to different, overlapping communities, we resolve conflicts as follows: (i) For two edge events $Delete(u, v, c, t)$ and $Insert(u, v, c', t)$, we remove the events generated by both, and (ii) For two edge events $Insert(u, v, c, t_1)$ and $Insert(u, v, c, t_2)$, $(t_1 < t_2)$ with no $Delete$ event between t_1 and t_2, we remove the event generated by the second insert. Analogously, we remove events generated by consecutive $Delete$ operations. When each node and edge in G has been thus processed, the stream of events is sorted by the time step.

C. Time Complexity

In this section, we analyze the cost of generating the dynamic events in the graph. At each time step t, we generate a random community event with probability p, and a random node event with probability p'. With $\epsilon = 2N^{-1}$, $\mathcal{O}(N)$ global edges are generated, which incurs only $\mathcal{O}(N)$ cost.

a) Node Events: Adding nodes incurs only constant work, thus resulting in only $\mathcal{O}(T)$ work over all time steps. Removing nodes only removes previously added edges and we can attribute its costs to their insertion.

b) Community Events: The time taken for generating the birth and death events of community c is dominated by the generation of insert and delete edge events. Using the Batagelj-Brandes model in the birth event, the time taken to generate edges is linear in the number of edges generated. Since $p_c = \alpha/n_c^{\gamma}$, the cost of the birth/ death event is $\mathcal{O}(n_c^{2-\gamma})$. Observing that $n_c \leq n_{max}$, the cost of the birth or death events is $\mathcal{O}(n_{max}^{2-\gamma})$. The split event of community c into c_1 and c_2 requires setting the end times of all edges that cross c_1 and c_2, and the number of such edges $\leq \alpha {n_c}^{2-\gamma}$. The cost for generating the additional internal edges in both c_1 and c_2 can be bounded by $\mathcal{O}({n_{c_1}}^{2-\gamma} + {n_{c_2}}^{2-\gamma}) = \mathcal{O}({n_c}^{2-\gamma})$, as $n_{c_1} + n_{c_2} = n_c$. On the other hand, in merging communities c_1 and c_2 into c, the overall cost is dominated by the cost of lines 21–33 in Algorithm 4, which is $\mathcal{O}(n_{c_1} \times n_{c_2}) = \mathcal{O}(n_{max}^2)$.

The time taken to convert G_T into G_0 and the event stream is dominated by the time taken to sort the stream. Note that at each time step, at most one community event takes place and the number of edges affected at one time step is bounded by the number of edges in any community. The size of the stream is therefore at most $\mathcal{O}(T{n_{max}}^{2-\gamma})$. For large values of T, the time to sort the stream is $\mathcal{O}(T{n_{max}}^{2-\gamma} \log(T{n_{max}}^{2-\gamma}))$.

V. Experiments

We present the results of several experiments conducted with the dynamic graph generator. The empirical evaluation of the generator has been done along three main categories.

1) Running Time: We measure the efficiency of the generator itself in constructing graphs with increasing graph size and event probability. The goal of these experiments is to illustrate that the generator scales well when generating large, dense, or highly dynamic graphs.
2) Graph Properties: As the dynamic graph evolves over time due to the edges and nodes changed by the generator, properties of the graph are measured at different time steps. This category of experiments is targeted at showing that the generator is capable of *maintaining* graph properties while generating community scale events.
3) Community Detection Algorithm: Finally, we use existing community detection algorithms, OSLOM [25] and MOSES [3], to detect communities on the generated datasets. The two selected community detection algorithms are two of the best performing algorithms for the CKB benchmark, as evaluated and reported in [26]. We show using these experiments that the link structure of the generated dynamic graph follows the ground truth even as the graph evolves. Moreover, we use the dynamic variant of OSLOM on the datasets to show how it reacts to changes in the community assignments.

Our primary contribution is that we add dynamic events to an existing static graph generator for overlapping communities. The goal of our experiments is, therefore, to show that generating community events on the initial graph (G_0), generated by the static generator, does not adversely affect the graph properties that G_0 exhibits. The static graph model used for generating G_0 in this work, CKB, has been shown to resemble real world graphs in terms of graph properties in [11]. We show that, in spite of the dynamic events generated on the original static graph, the same graph properties are followed and the link structure indicated by the community assignment is maintained. This implies that the dynamic generator model we propose, is able to generate a graph with evolving overlapping communities, realistic graph properties, and community events at *each* time step - forming a suitable input to a dynamic community detection algorithm.

A. Setup

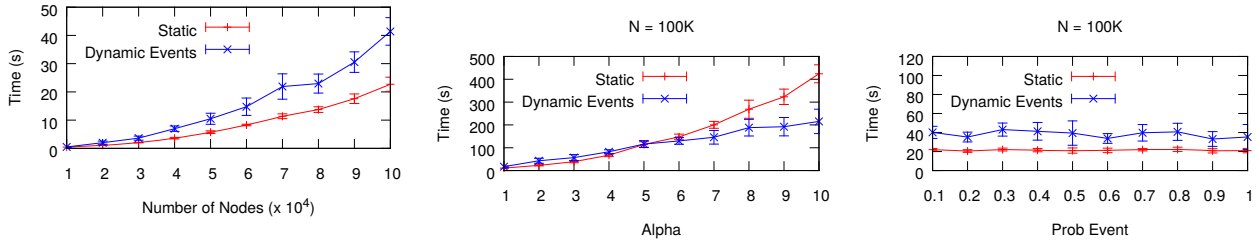
Our experiments were conducted on a system with 32 GB of RAM, 4 cores with 2 threads per core. The parameters experimented with include N, the starting number of nodes in the graph, p, the probability of an event occurring, α, which affects the intra cluster edge density, and ϵ, which affects the density of inter-community or global edges. Other parameters are set to their values in Table I, and the value of t_{effect} is set to a random number in $[5, 15)$ for each community event.

B. Running Time

Figure 4 plots the time taken by the static graph generation module (see Section IV-A) that generates a static graph at time step 0 (called the starting graph), and the time taken by the dynamic graph generator (see Section IV-B) that generates events spanning time steps $t = 1 \ldots T$.

Figure 4a shows the time taken for varying number of nodes in the starting graph and $T = 1000, p = 0.1, \alpha = 2$, and $\epsilon = 2$. As the value of N increases, the size of the starting graph increases. This results in larger static graph generation time at $t = 0$, as well as longer event generation times for the dynamic module, since a greater number of nodes and edges participate

Authorized licensed use limited to: KIT Library. Downloaded on October 19,2021 at 08:29:20 UTC from IEEE Xplore. Restrictions apply.



(a) Varying starting number of nodes (b) Varying intra cluster edge density (c) Varying event probability

Fig. 4: Time taken to generate initial static graph and dynamic events

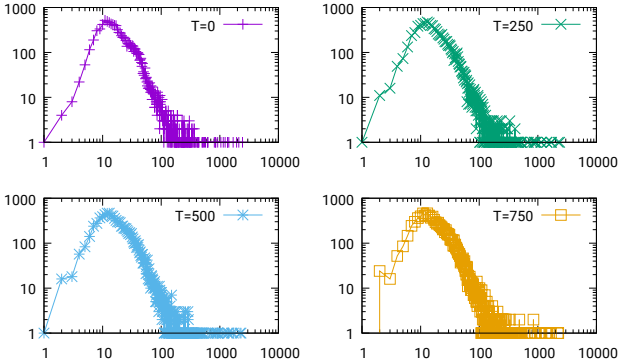


Fig. 5: Degree Distribution

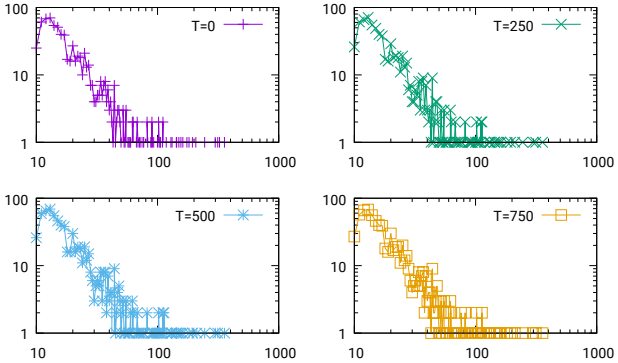


Fig. 6: Community Size Distribution

in each community event. For these experiments, n_{max} and x_{max} are set to $N/10$, while $n_{min} = N/1000$ and $x_{min} = 1$. Figure 4b shows the individual running times for varying values of α and $T = 1000, N = 10^5, p = 0.1$, and $\epsilon = 2$. Since α directly influences intra-community edge probability, a larger value of α implies a denser starting graph, and therefore more expensive starting graph and dynamic event generation. Figure 4c shows the time taken for varying probabilities of dynamic events and $T = 1000, N = 10^5, \alpha = 2$, and $\epsilon = 2$. As p goes to 1, more events take place, resulting in more changes to the graph at each time step. We note that the frequency of dynamic events to the graph has a much smaller effect on the overall graph generation time, implying that the running time of the generator is dominated by the number of nodes/ edges that are affected by each event rather than the number of events that occur. All points reported are the mean and standard deviation over 10 separate instances.

C. Graph Properties

In Figures 5, 6, and 7, the graph at $t = 0$ is generated by the static graph generation module [11], which has been shown to generate graphs with properties close to that of real world graphs. The static graph generator is run with parameters $N = 10000$, $n_{min} = 10$, $n_{max} = x_{max} = N/10$, $x_{min} = 1$, $p = 0.1$, $\alpha = 2$, and $\epsilon = 2$. Subsequently, the dynamic graph generation module generates events for the time steps $t = 1 \ldots 1000$. We plot graph properties at equal-sized intervals of time $t = 250$, $t = 500$, and $t = 750$. Clearly, the *tail* of the degree distribution remains a power law, as in real world graphs. Most importantly, we see that even as the graph structure evolves to incorporate community scale events, the various distributions remain mostly unchanged.

In Figure 8, the dynamic graph generator is invoked with different values of α. For each dynamic graph thus generated, we construct its snapshots from $T = 1$ to $T = 1000$ and measure the clustering coefficient of each, using NetworKit [27]. For $\alpha = 2$ for instance, the clustering coefficient of the graph remains stable at around 0.2, which is close to the clustering coefficient of many real world graphs [28].

D. Community Detection Algorithm

To evaluate community detection algorithms, the most often used quality measure is the Normalized Mutual Information, which has been shown to suffer from the selection bias [29]. As a result, we choose to use the *average weighted F1 score* [30]. Let C_G and C_D be the ground truth and detected set of clusters respectively. We define the weighted F1 as

$$F_w(C_G, C_D) = \frac{1}{\sum_{C \in C_G} |C|} \sum_{C \in C_G} |C| \max_{C' \in C_D} F1(C, C')$$

The average weighted F1 score is the average of $F_w(C_G, C_D)$ and $F_w(C_D, C_G)$.

The dynamic graph generated by our generator is evaluated against two well-known community detection algorithm

Authorized licensed use limited to: KIT Library. Downloaded on October 19,2021 at 08:29:20 UTC from IEEE Xplore. Restrictions apply.

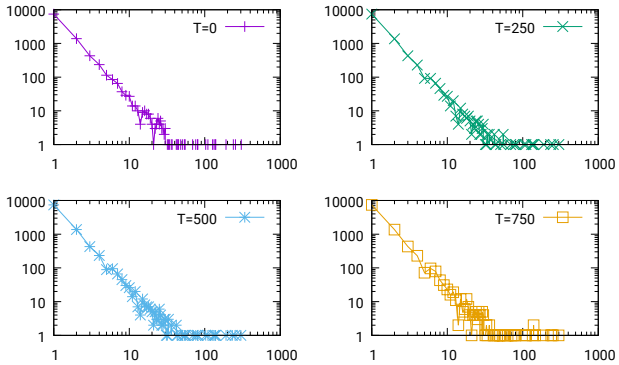


Fig. 7: Node Membership Distribution

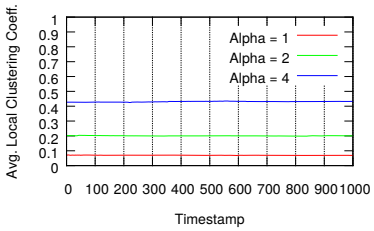


Fig. 8: Clustering Coefficient

MOSES ([3]) and OSLOM ([25]). While OSLOM has a dynamic variant that is able to update community assignments for a dynamic graph, MOSES can only detect communities for a static graph. Our graph generator generates a static graph at time step 0 along with a set of graph updates spanning time steps $1 \ldots T$. The starting graph and set of graph updates are input to the dynamic variant of OSLOM, and the weighted F1 is measured at each time step (OSLOM-D). For evaluating our generated data against the static algorithms, the graph updates from $t = 1 \ldots T$ are unrolled to create a graph snapshot for each individual time step. The static community detection algorithms (OSLOM-S and MOSES) are then run for each individual graph snapshot and the weighted F1 score at each time step is measured independently.

For Figures 9 and 10, the dynamic graph generator is invoked for $N = 10,000$ nodes, $T = 1000$ timesteps, $n_{min} = 10$, $n_{max} = 1000$, $x_{min} = 1$, $x_{max} = 1000$, $p = 0.1$, with the default values for α and ϵ both set to 2. Figure 9 shows the performance of the algorithms MOSES and OSLOM (static and dynamic) as the dynamic graph evolves. As the value of ϵ is varied, the 'global edge' probability increases, thereby blurring the community boundaries. MOSES is able to distinguish the correct communities even for $\epsilon = 10$, while OSLOM-S performs slightly worse. Notably, while both static community detection algorithms are able to distinguish the correct communities throughout the T time steps, the dynamic variant of OSLOM, OSLOM-D, drops in performance considerably early. This shows that the link structure of the generated graph follows the ground truth faithfully over time, and that communities can still be distinguished by the static version of the same algorithm. However, the dynamic variant of OSLOM is unable to detect the communities in the same dataset, implying that detecting overlapping communities efficiently and reliably in a dynamic graph remains a challenge.

Similarly, Figure 10 shows the performance of the two algorithms with time and varying values of α. As the value of α increases, the intra-community edge probabilities increase, implying that communities are more easily distinguishable. Interestingly, the dynamic version of the OSLOM algorithm does not have high error until 700 time steps for $\alpha = 4$. The reason for this is that for a high value of α, communities have large cliques and high internal edge density, and even as community events occur, the large intra-community edge density is maintained. Consider a merge event between communities c_1 and c_2, merged into community c_3. If the value of α is small, then the required intra-community edge density of c_3 is small, and so a small number of edges are inserted between the nodes of c_1 and c_2. On the contrary, if the value of α is large, then a large number of edge insertions occur. Clearly, the larger the number of edge insertions, the easier it is for the dynamic algorithm to detect the merge event, and distinguish c_3 from the rest of the graph. This observation reinforces the intuition that dynamic community detection algorithms work better when community events are large scale (give rise to large number of edge/ vertex events) and when communities themselves have a higher internal edge density.

VI. Conclusion

We have introduced the first benchmark graph generator for dynamic overlapping community detection. The node degrees, the community sizes, and the number of communities per node all follow power law distributions as commonly observed in real-world networks. In the experimental evaluation we show that these properties and a realistic local clustering coefficient are not only present in the initial graph but also maintained over time. Further, we demonstrate that our generator is capable of generating graphs with 100,000 nodes and 1000 time steps in under a minute. Our generator is therefore the ideal starting point for an extensive evaluation of existing and novel dynamic community detection algorithms. Further, we show that existing community detection algorithms are capable of finding a community structure similar to the ground truth structure but that maintaining it over time remains a challenge. An obvious direction for future work is therefore the development of novel algorithms for detecting overlapping communities in dynamic networks.

Acknowledgment

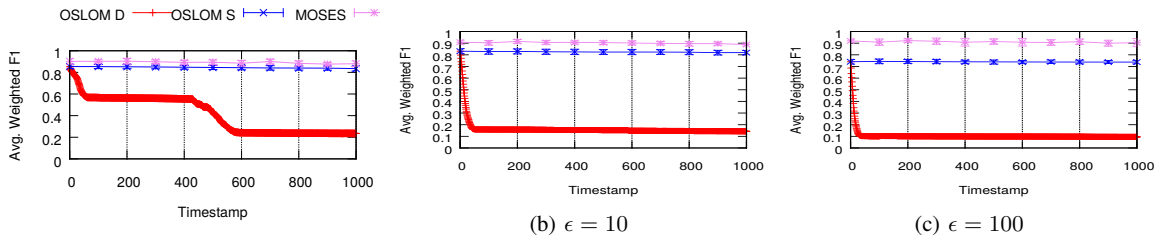
This work was supported by the DFG under grant WA 654/22.

References

[1] D. Duan, Y. Li, R. Li, and Z. Lu, "Incremental k-clique clustering in dynamic social networks," *Artificial Intelligence Review*, 2012.

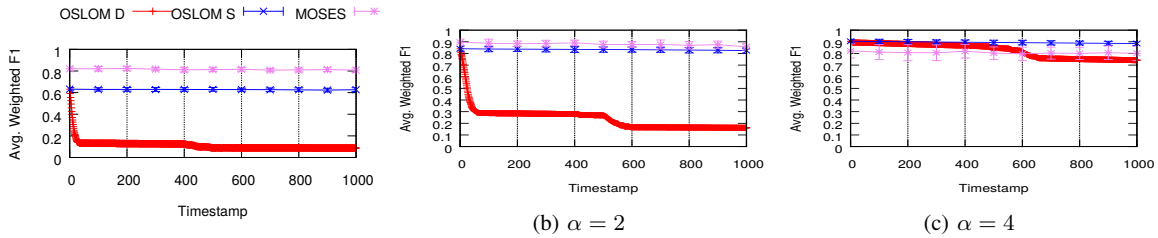
[2] C. Lee, F. Reid, A. McDaid, and N. Hurley, "Detecting highly overlapping community structure by greedy clique expansion," *arXiv preprint arXiv:1002.1827*, 2010.

Authorized licensed use limited to: KIT Library. Downloaded on October 19,2021 at 08:29:20 UTC from IEEE Xplore. Restrictions apply.



(a) $\epsilon = 1$ (b) $\epsilon = 10$ (c) $\epsilon = 100$

Fig. 9: Performance of OSLOM and MOSES algorithm with varying ϵ



(a) $\alpha = 1$ (b) $\alpha = 2$ (c) $\alpha = 4$

Fig. 10: Performance of OSLOM and MOSES algorithm with varying α

[3] A. F. McDaid and N. J. Hurley, "Using model-based overlapping seed expansion to detect highly overlapping community structure," *arXiv preprint arXiv:1011.1970*, 2010.

[4] J. Yang and J. Leskovec, "Community-Affiliation Graph Model for Overlapping Network Community Detection," in *Proceedings of the 2012 IEEE International Conference on Data Mining*. IEEE Computer Society, 2012.

[5] D. Greene, D. Doyle, and P. Cunningham, "Tracking the evolution of communities in dynamic social networks," in *Advances in social networks analysis and mining (ASONAM)*, 2010.

[6] J. Yang and J. Leskovec, "Structure and Overlaps of Ground-Truth Communities in Networks," *ACM Transactions on Intelligent Systems and Technology*, 2014.

[7] N. P. Nguyen, T. N. Dinh, S. Tokala, and M. T. Thai, "Overlapping communities in dynamic networks: their detection and mobile applications," in *Proceedings of the 17th annual international conference on Mobile computing and networking*, 2011.

[8] Y. Chen, V. Kawadia, and R. Urgaonkar, "Detecting overlapping temporal community structure in time-evolving networks," *arXiv preprint arXiv:1303.7226*, 2013.

[9] A. Amelio and C. Pizzuti, "Overlapping community discovery methods: a survey," in *Social Networks: Analysis and Case Studies*, 2014.

[10] A. Lancichinetti, S. Fortunato, and F. Radicchi, "Benchmark graphs for testing community detection algorithms," *Physical review E*, 2008.

[11] K. Chykhradze, A. Korshunov, N. Buzun, R. Pastukhov, N. Kuzyurin, D. Turdakov, and H. Kim, "Distributed generation of billion-node social graphs with overlapping community structure," in *Complex Networks V*, 2014.

[12] G. Palla, A. Barabási, and T. Vicsek, "Quantifying social group evolution," *Nature*, vol. 446, 2007.

[13] T. Berger-Wolf, D. Kempe, and C. Tantipathananandth, "A Framework For Community Identification in Dynamic Social Networks," in *Proceedings of the 13th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. ACM Press, 2007.

[14] S. Asur, S. Parthasarathy, and D. Ucar, "An event-based framework for characterizing the evolutionary behavior of interaction graphs," *ACM Transactions on Knowledge Discovery from Data*, vol. 3, no. 4, December 2009.

[15] L. Backstrom, D. Huttenlocher, J. M. Kleinberg, and X. Lan, "Group formation in large social networks: membership, growth, and evolution," in *Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. ACM Press, 2006.

[16] E. N. Gilbert, "Random Graphs," *The Annals of Mathematical Statistics*, 1959.

[17] A. Condon and R. M. Karp, "Algorithms for Graph Partitioning on the Planted Partition Model," *Randoms Structures and Algorithms*, 2001.

[18] A. Lancichinetti and S. Fortunato, "Benchmarks for testing community detection algorithms on directed and weighted graphs with overlapping communities," *Physical Review E*, 2009.

[19] C. Tantipathananandth and T. Berger-Wolf, "Finding Communities in Dynamic Social Networks," in *Proceedings of the 2011 IEEE International Conference on Data Mining*. IEEE Computer Society, 2011.

[20] R. Görke, R. Kluge, A. Schumm, C. Staudt, and D. Wagner, *An efficient generator for clustered dynamic random networks*, 2012.

[21] C. Granell, R. K. Darst, A. Arenas, S. Fortunato, and S. Gomez, "Benchmark model to assess community structure in evolving networks," *Physical Review E*, 2015.

[22] P. Erdos and A. Rényi, "On the evolution of random graphs," *Publ. Math. Inst. Hung. Acad. Sci*, 1960.

[23] T. G. Kolda, A. Pinar, T. D. Plantenga, and C. Seshadhri, "A scalable generative graph model with community structure," *CoRR*, vol. abs/1302.6636, 2013.

[24] V. Batagelj and U. Brandes, "Efficient generation of large random networks," *Physical Review E*, 2005.

[25] A. Lancichinetti, F. Radicchi, J. Ramasco, and S. Fortunato, "Finding statistically significant communities in networks," *PloS one*, 2011.

[26] N. Buzun, A. Korshunov, V. Avanesov, I. Filonenko, I. Kozlov, D. Turdakov, and H. Kim, "Egolp: Fast and distributed community detection in billion-node social networks," in *2014 IEEE International Conference on Data Mining Workshop (ICDMW)*. IEEE Computer Society, 2014.

[27] C. L. Staudt, A. Sazonovs, and H. Meyerhenke, "Networkit: A tool suite for large-scale complex network analysis," *Network Science*, vol. 4, no. 4, 2016.

[28] J. Leskovec and A. Krevl, "SNAP Datasets: Stanford large network dataset collection," http://snap.stanford.edu/data, 2014.

[29] A. Amelio and C. Pizzuti, "Is normalized mutual information a fair measure for comparing community detection methods?" in *Advances in Social Networks Analysis and Mining (ASONAM), 2015 IEEE/ACM*, 2015.

[30] J. Yang and J. Leskovec, "Defining and evaluating network communities based on ground-truth," *Knowledge and Information Systems*, 2015.

Authorized licensed use limited to: KIT Library. Downloaded on October 19,2021 at 08:29:20 UTC from IEEE Xplore. Restrictions apply.