*S*treaming *C*ommunity *R*andom *E*volution *A*lgorithm for *M*odularity (*SCREAM*)

Adam Amos-Binks Randall P. Dahlberg

9 January 2014

Abstract

Current graph partitioning and community detection algorithms rely on static data analysis (data-at-rest) to identify meaningful groupings within graphs. While this approach is an important step in understanding underlying structure of past observations, it is limited in quantifying the value of new data, the temporal relationships between data, and community behavior in a dynamic streaming environment. Addressing these limitations with a streaming graph partitioning algorithm would yield greater situational awareness, uncover inter-community dynamics, identify community roles of interest, and provide new insights into the value of data observed in a data stream, but has gone unaddressed in current research. This paper introduces a new streaming community detection algorithm, *SCREAM*, which can receive a data stream consisting of time-stamped edges and output community structure and various measurements associated with the dynamics of community membership over time. The *SCREAM* algorithm makes use of efficient modularity updates inspired by the Louvain algorithm, and modified by the standardization process introduced by Bader and McCloskey [BM10]. However, in contrast to the original and standardized versions of Louvain, *SCREAM* is a local algorithm in the sense that with the arrival of a new edge, only those communities containing nodes incident to the new edge are updated. In this paper, we give examples showing the inter-community dynamics observed over time, and how the order in which edges are observed in a data stream has a significant effect on the communities created. Finally we introduce a simple graph generation algorithm which produces a timestamped stream of edges exhibiting various levels of cohesive community structure over time.

Contents

1. Introduction

Community detection and graph partitioning offer a convenient way to decompose a large graph into more manageable components, allowing more focused questions and adding additional community dynamic information. According to Papadopoulos et al. [PKVS12], "the problem that community detection attempts to solve is the identification of groups of vertices that are more densely connected to each other than to the rest of the network." Lancichinetti and Fortunato [LF09] have stated that detecting communitites in networks is a fundamental problem in computer science. Once communities have been defined, roles within communities can be defined. Work by Scripps et al. [STE07] identifies four such community roles (ambassadors, big fish, loners, and bridges) that may prove useful in further discovering and interpreting communities.

The use of graphs in previous work has been primarily concerned with static analysis of data-at-rest to identify meaningful groupings (communities) within graphs (Lancichinetti & Fortunato [LF09], and Lancichinetti et al. [LFR08]), to find suspicious domains associated with botnets (Jiang et al. [JCLZ10], and to compress the graph by collapsing communities into "supernodes" (Rosvall and Bergstrom [RB08], Blondel et al. [BGLL08]). While this approach is an important step in understanding underlying structure of static graphs, it is limited in quantifying the value of new data in dynamically evolving graphs, the temporal relationships between the data, and community behavior in a dynamic streaming environment. Addressing these limitations with a streaming graph partitioning algorithm would yield greater situational awareness, uncover inter-community dynamics, identify community roles of interest, and provide new insights into the value of data observed in a data stream.

Recently, efforts on streaming algorithms and data-in-motion have accounted for the temporal dimension of edges, and the evolution of the connectedness of a graph over time. Analyzing data-in-motion accounts for this temporal dimension of data. A small but tightly temporally related community within a large graph may be grouped into a larger community in a static analysis, but may be identified as its own community in a streaming algorithm due to the strong temporal relationship between its intra-community edges. A goal of this paper is to account for the temporal dimension of graphs as they form into communities, as it has largely been ignored in the past in the literature. It is important to remember throughout the paper that direct comparisons between static and streaming results are not meaningful.

In a community detection review paper by Papadopoulos et al. [PKVS12], it was noted that the "possibility for incremental updates of already identified community structure" has not been adequately addressed in related survey articles. The main goal of this paper is to explore this

possibility by introducing the *SCREAM* algorithm. *SCREAM* receives a data stream consisting of time-stamped edges and outputs community structure and various measurements associated with the dynamics of community membership over time. The *SCREAM* algorithm makes use of efficient modularity updates inspired by the Louvain algorithm and modified by the standardization process introduced by McCloskey in [BM10]. However, in contrast to the original and standardized versions of Louvain, *SCREAM* is a local algorithm in the sense that with the arrival of a new edge, only those communities containing nodes incident to the new edge are updated. In this paper we give examples showing the inter-community dynamics observed over time, and how the order in which edges are observed in a data stream have a significant effect on the communities created. While the *SCREAM* algorithm accounts for the temporal ordering of new edges in its community assignments, it is limited to making assignments for nodes present in an initial starting graph. A second version, *SCREAM 2* has been developed that allows for new communities with new nodes to be handled in the stream of new edges. *SCREAM 2* will be detailed in a second paper. Finally, we introduce a simple graph generation algorithm which produces a time-stamped stream of edges exhibiting various stages of community structure over time.

2. Background

In this section we review some of the important concepts in community detection in graphs/networks, introduce the streaming framework provided by the InfoSphere Streams [IBM12] programming environment, and provide some detail on how the *SCREAM* algorithm updates community structure with the arrival of each edge.

2.1. Modularity

The modularity used in this paper is due to Newman and Girvan [NG04] with standardization modifications by McCloskey[BM10]. Although modularity can be defined for all kinds of graphs (see Fortunato [For10], here we only consider undirected graphs $G = (V, E), |V| = n, |E| = m$, together with a weight function on the edge set, $wt : E \longrightarrow \mathbb{R}^+ \equiv [0, \infty)$. If $e = (v_1, v_2) \in E$, we say v_1 and v_2 are *incident* to the edge e. For a vertex $v \in V$, let $E_v = \{e \in E \mid v \text{ is incident to } e\}$ and, more generally, for a subset $S \subset V$, let E_S denote the set of all edges whose incident vertices lie in S. A partitioning of the vertex (or node) set V into communities $V_1, \ldots, V_K$, is a decomposition $V = \cup_{k=1}^{K} V_k$ where $V_k \cap V_l = \emptyset$ for $k \neq l$. For a community $V_k \subset V$, denote by $G(V_k)$ the subgraph of G induced by the vertices in V_k, i.e. $G(V_k) = (V_k, E_{V_k})$ where $E_{V_k} = E \cap (V_k \times V_k)$.

The most basic definition of *modularity* is for an unweighted undirected graph $G = (V, E)$ with symmetric adjacency matrix $A = (A_{ij})_{i,j=1}^{n}$, consisting of K communities as defined above. In this case, the weight function takes values in $\{0, 1\}$. Our discussion here is based on Section III.C.2 of the survey paper of Fortunato [For10]. Define the community index function $C : V \longrightarrow \{1, 2, \ldots, K\}$ by $C_i = k$ if node i belongs to community V_k. The modularity of this network is defined to be

$$Q = \frac{1}{2m} \sum_{i,j} (A_{ij} - M_{ij})\, \delta(C_i, C_j), \tag{2.1}$$

where M_{ij} represents the expected (mean) number of edges between nodes i and j in the null model of the graph, and δ is the Kronecker delta function, i.e. $\delta(C_i, C_j) = 1$ if $C_i = C_j$; otherwise, $\delta(C_i, C_j) = 0$. This definition can also be applied to weighted graphs provided that the weight of an edge, $wt(v_i, v_j)$, represents the number of times a communication between nodes v_i and v_j has been observed.[1] In this case, $A_{ij} = wt(v_i, v_j) = wt(v_j, v_i)$ and $m = \frac{1}{2}\sum_{i,j} A_{ij}$ as in equation (1) of Blondel et al. [BGLL08].

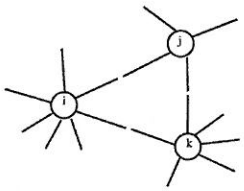


Figure 1: Half-edges (stubs) for nodes i and j.

The null model is that of Newman and Girvan [NG04] which consists of a randomized version of the original graph with edges reconnected at random under the constraint that the expected degree of each node matches the degree of the node in the original graph. The way this is done is by thinking of each edge as consisting of two stubs or half-edges (see Figure 1). If there are m edges, then there will be $2m$ stubs. To form an edge between nodes i and j we need to pick two stubs, incident to i and j respectively. There will be d_i stubs incident to node i where d_i is the degree of node i. If edges are picked randomly and independently, then the probability of an edge between i and j is $d_i d_j / 4m^2$. Hence under this null model, $M_{ij} = 2m\, d_i d_j / 4m^2 = d_i d_j / 2m$.

Since the only contributions to Q in equation (2.1) come from nodes that lie in the same community V_k, we need only consider sums for each community of the form

$$Q_{V_k} \;=\; \frac{1}{2m} \sum_{i,j \in V_k} \left[A_{ij} - \frac{d_i d_j}{2m} \right]. \tag{2.2}$$

for $k = 1, \ldots, K$. Now $\sum_{i,j \in V_k} A_{ij} = 2l_k$ where $l_k = |E_{V_k}|$ is the number of edges between nodes in V_k, and $\sum_{i,j \in V_k} d_i d_j = d_{V_k}^2$ where $d_{V_k} = \sum_{i \in V_k} d_i$.[2] Thus we have

$$Q \;=\; \sum_{k=1}^{K} Q_{V_k} = \frac{1}{2m} \sum_{k=1}^{K} \left[2l_k - \frac{d_{V_k}^2}{2m} \right] = \sum_{k=1}^{K} \left[\frac{l_k}{m} - \left(\frac{d_{V_k}}{2m} \right)^2 \right], \tag{2.3}$$

which is equation (14) for modularity in Section III.C.2 of Fortunato [For10].

We can make use of equation (2.3) to derive equation (2) in Blondel et al. [BGLL08] for the change in modularity after moving an isolated node i into a community V_r. Using our notation

[1] According to McCloskey, the A_{ij} must be non-negative integers due to the implied use of statistical independence in the computation of the M_{ij}. However, in the literature, real-valued non-integer A_{ij} have been used, perhaps incorrectly. See the long survey paper by Fortunato [BGLL08] for examples.

[2] We have not ruled out self-loops in our graphs so every ij-th entry of A has a small probability of being positive. Let $|V_k| = n_k$. Then $\sum_{i,j \in V_k} d_i d_j = \sum_{i=1}^{n_k} d_i^2 + 2\sum_{1 \le i < j \le n_k} d_i d_j = \left(\sum_{i=1}^{n_k} d_i\right)^2 = d_{V_k}^2$.

above and letting $l_{i,V_r} = |\{(i,j) \in E \mid j \in V_r\}|$, i.e. the number of edges incident to i and nodes in V_r, the modularity before the move is

$$Q_{before} = \sum_{k \neq i,r} Q_{V_k} + Q_{V_r} + Q_{\{i\}} = \sum_{k \neq i,r} Q_{V_k} + \frac{2l_r}{2m} - \left(\frac{d_{V_r}}{2m}\right)^2 + \frac{A_{ii}}{2m} - \left(\frac{d_i}{2m}\right)^2, \quad (2.4)$$

and after the move,

$$Q_{after} = \sum_{k \neq i,r} Q_{V_k} + \frac{2l_r + l_{i,V_r} + A_{ii}}{2m} - \left(\frac{d_{V_r} + d_i}{2m}\right)^2. \quad (2.5)$$

Hence the gain in modularity, $\Delta Q = Q_{after} - Q_{before}$, is

$$\Delta Q = \left[\frac{2l_r + l_{i,V_r}}{2m} - \left(\frac{d_{V_r} + d_i}{2m}\right)^2\right] - \left[\frac{2l_r}{2m} - \left(\frac{d_{V_r}}{2m}\right)^2 - \left(\frac{d_i}{2m}\right)^2\right]. \quad (2.6)$$

We recover equation (2) in Blondel et al. [BGLL08] by making the substitutions: $\sum_{in} = 2l_r$, $k_{i,in} = l_{i,V_r}$, $\sum_{tot} = d_{V_r}$, and $k_i = d_i$. Note that (2.6) simplifies to

$$\Delta Q = \frac{l_{i,V_r}}{2m} - \frac{2d_{V_r}\, d_i}{4m^2} = \frac{1}{4m^2}\left[(2m)l_{i,V_r} - 2d_{V_r}\, d_i\right]. \quad (2.7)$$

We can also recover the numerator of the McCloskey standardized Louvain update formula (see equation (3.1) in Section 3.1 of this paper) by substituting $N = 2m$, $X_{i,V_r} = l_{i,V_r}$, $X_{V_r+} = d_{V_r}$, and $X_{i+} = 2d_i$. The *SCREAM* algorithm makes use of this update when a new edge is processed. Using the same arguments, we can compute the change in modularity when moving a node out of one community and into another (see Appendix A).

2.2. Louvain Algorithm

The Louvain algorithm for community detection has been described as "the state-of-the-art technique" by De Meo et al. [MFFP11], and has demonstrated its superiority in comparative analyses by Lancichinetti and Fortunato [LF09], and Papadoupoulos et al. [PKVS12]. The Louvain algorithm refers to a community detection algorithm by Blondel et al. [BGLL08] based on modularity maximization that recursively compresses a graph by building a hierarchy of communities. Each recursive pass of the algorithm consists of two phases. The first is an agglomerative clustering process that combines isolated nodes into communities until a local maximum is achieved for modularity. The second phase compresses the graph by collapsing each community found in the first phase into a *supernode*. Weights of links between these new supernodes are computed by summing the weights of the individual links between the communities. In addition, a self-link is created for each supernode with weight equal to the sum of the weights of links between members of the community from the first phase. The algorithm is then applied to the resulting compressed graph. This process continues as long as a positive increase in modularity can be obtained. A standardized version of the Louvain algorithm was introduced by McCloskey in [BM10] which employes a standardized version of modularity that allows determination of statistically significant increases in modularity. **In light of improvements demonstrated by the use of the McCloskey**

standardization of modularity, the version of Louvain that we actually employ in this paper makes use of standardized modularity.

The Louvain algorithm is designed to handle hierarchical community structure in a graph since it produces a dendrogram much like a hierarchical clustering algorithm. Although higher-level passes of the algorithm suffer from the resolution limit problem, as does any modularity-based algorithm, the Louvain algorithm deals with this problem through the multi-resolution nature of the output, i.e. the nested community structure produced by the various passes of the algorithm. However, this introduces the same difficulty as with any hierarchical clustering algorithm: at what level does one "cut" the dendrogram to determine the number of communities or, in the case of Louvain, which pass do we use to present the results? Fortunato in his survey paper on community detection [For10] makes some additional criticisms.

> However, closing communities within the immediate neighborhood of vertices may be inaccurate and yield spurious partitions in practical cases. So, it is not clear whether some of the intermediate partitions could correspond to meaningful hierarchical levels of the graph. Moreover, the results of the algorithm depend on the order of the sequential sweep over the vertices.[3]

Fortunato's first point needs to be examined for real data. On the otherhand, as we will see below, since the SCREAM algorithm processes each edge as it arrives and only makes update to communities in the neighborhood of nodes incident to the new edge, the only randomness involved is that of edge arrival. The computation of the standardized modularity for the communities affected by the arrival of a new edge is deterministic.

The Louvain algorithm is extremely fast with most of the time spent in the first phase of the first pass. In fact, it has been pointed out that the algorithm is more limited by storage demands than by computational time ([BGLL08], [For10]). The efficiency of the algorithm is based in part on the ease in computing the change in modularity using equation (2.7) for moving an isolated node into a community, and equation (A.7) for moving a node out of one community and into another. Later passes deal with far fewer nodes since these nodes represent a compression of the communities discovered in the previous pass. Variants of the Louvain algorithm have been proposed including one based on edge centrality [MFFP11].

2.3. Features

The following definitions are used in the computation of features listed in Table 2.1. Here Q_C^t refers to the modularity (plain or standard) of community C at time t.

- adjacency weight (transition count) of a vertex v: $\text{adjWeight}_v = \sum_{e \in E_v} wt(e)$
- adjacency weight of a community C: $\text{adjWeight}_C = \sum_{v \in C} \text{adjWeight}_v = 2 \sum_{e \in E_C} wt(e)$
- $\text{intraCommunityEdgeWeight}(C) = \sum_{e=(v_1,v_2) \in E, v_1, v_2 \in C} \text{wt}(e)$

[3] Fortunato [For10], p. 29.

Index	Field	Comments
1	*nodeA*	first vertex of current edge
2	*nodeB*	second vertex of current edge
3	*calc* (calculation)	two operations and description - **sub**: remove nodeID from communityID, **add**: add nodeID to communityID; **edge**: current edge
4	*nodeId*	node of current edge used in calc
5	*communityId*	community used in calc
6	*dateTime*	timestamp from incoming edge
7	*incQ*	incremental (plain or standardized) modularity contributed by current edge
8	*usedGraphWeight*	sum of edge weights of nodes in the graph
9	*communityQ*	current value of community's modularity
10	*communityCumulativeQ*	sum of (plain or standardized) modularity contributions of edges since last full louvain
11	*communityAbsCumulativeQ*	sum of (plain or standardized) modularity contributions of edges since last full louvain
12	*communityWeights*	community's current importance, as measured by the proportion of edges within the community
13	*communityNodeCounts*	current size of the communities
14	*communityAdjWeights*	sum of edge weights of the nodes in the community
15	*communityAdjacencyMap*	list showing inter- and intra-community connectivity

Table 2.1: Field descriptions.

- incremental community modularity: $\text{inc}\, Q_C = Q_C^t - Q_C^{t-1}$
- absolute value of the incremental modularity: incAbs $|Q_C| = |\text{inc}\, Q_C| = \left|Q_C^t - Q_C^{t-1}\right|$

The algorithm outputs four files. Two of the files contain all of the feature information outlined in the table for the SCREAM algorithm. The output uses the standardized modularity update shown in equation (3.1) of Section 3.1. The other two output files are summarized versions of the complete feature information and are used as input to various R statistical programs used by the authors.

2.4. Streaming Platform

IBM's InfoSphere Streams [IBM12] was developed to address stream computing, an emerging paradigm that is rapidly increasing in scope and scale. Stream computing strives to build context, gain intelligence, and make decisions on massive amounts of data that are infeasible or impractical to analyze postmortem. Traditional data analytics are performed in batch format or in an *ad hoc* manner on a large amount of historical data. While this approach has a great deal of applications, for example the great increase in popularity of Hadoop and Map Reduce frameworks over the past

few years, incoming data that may be useful for the next hour is constrained to being analyzed in the next batch job. This is problematic as the batch job's scheduled start time may be perhaps several hours from the data's arrival time and outside its window of usefulness.

Streams is a software framework with its own language, SPL (Streams Processing Language), to construct stream computing applications. SPL can be best described as a merging of two classic programming paradigms: object-oriented (e.g. C++, Java, etc.) and declarative (eg. SQL, MATLAB, etc). At its core, SPL is used to write applications which operate on data *in situ*, and pass the output on to other applications, retaining very little context about what has been observed. To provide an architectual overview, the five relevant infrastructure components will be discussed below. For a more complete description of each component and discussion of the SPL grammar, please refer to the IBM Language Specification [IBM12].

2.4.1. Tuple

Tuples are the principal data container of SPL used to pass data between application components. Tuples are akin to a record in a database or a row in a spreadsheet whose columns are referred to as tuple members in SPL. Tuple members can be typed as strings, real numbers, integer numbers, or collections. An example of a tuple declaration (SampleTuple) can be observed in Line 1 of Figure 2 below. Modification and manipulation of tuples can be done within a stream (see next subsection) using the ":" modifier, an example of which is shown on line 12.

2.4.2. Stream

A stream is the mechanism over which tuples are passed between operators. A stream is defined by the tuple type which is both the input and output. A stream does not need to be used by another operator in SPL, and tuples can simply be dropped. Similarly, a stream can be sent to multiple operators to process, and be merged again at a later date. In Figure 2, lines 9 and 14 both contain stream declarations (SampleStream and the null stream, respectively). SampleStream is declared to be of type SampleTuple and connects SampleOperator and SampleFileWriter together, whereas the null stream simply defines there to be no output stream (as declared by the () brackets in line 14) for SampleFileWriter.

2.4.3. Operator

Operators are where the "work" happens on a tuple. Operators can accept multiple streams as input, perform some analysis on the observations, and then output a combination of the resulting tuple and original tuples over a number of streams. There are no callbacks in SPL, resulting in an operator knowing nothing of what happens to a tuple once sent to a stream. The SPL standard Toolkit provides some built-in operator types for common operations such as Filter, Aggregate, and Merge. Use of the built-in operators Functor and FileSink (Figure 2, lines 9 and 14) display the declaration of Standard Toolkit operators.

```
1 type SampleTuple = rstring tupleString, int32 tupleInt;
2
3 public composite SampleComposite()
4 {
5 param
6      expression<rstring> $parameterString    :     "newValue";
7      expression<int32> $parameterInt         :     1;
8 graph
9     (stream <SampleTuple> SampleStream) as SampleOperator = Functor()
10    {
11     output
12     SampleStream : tupleString = $parameterString, tupleInt = $parameterint;
13     }
14    () as SampleFileWriter = FileSink(SampleStream)
15    {
16    param
17          file    : "sampleOutput.txt";
18           format  : csv;
19   }
20 }
```

Figure 2: Sample SPL Code

2.4.4. Composite

In line with Object Oriented principles, a high degree of code reuse is enabled with the use of the composite operator. The Composite operator is a special type of operator composed of other operators connected together by streams, along with all other tuple definitions and helper functions. Composite operators can then be submitted as stand-alone jobs to run indefinitely within an instance. Line 3 of Figure 2 declares a Composite operator with no inputs or outputs.

2.4.5. Instance

In declarative languages, there is a workbench where all variables exist and can be combined together in calculations. SPL has an analogous facility called an Instance. Often in Streams installations, there is a shared or public instance where users submit their jobs (in the form of composite operators), so they may interact together, share an incoming data feed, or merge outputs. Private instances are an important arena for research and testing, as they can be created for individuals to mitigate experimental code failures.

3. *SCREAM* Implementation

The *SCREAM* algorithm has a Louvain operator written in SPL, the native language in InfoSphere Streams, and packaged in a toolkit for re-use (see for example, [IBM12]). The principal idea behind the *SCREAM* Louvain operator is that an initial start graph is provided, on which the full Louvain algorithm identifies initial partitions. Following the initial partition determined by the Louvain algorithm, the operator will accept a stream of edges. For this version of *SCREAM*, these edges can only contain vertices from the initial Louvain partition. In a recent updated version, *SCREAM 2*, the edge stream can contain edges on new vertices and new communities can be formed in a burst of edges. For each edge observed in the stream, the operator computes each node's community assignment using equation (3.1), and update state information maintained on the communities and graph as a whole. The following section explains the Louvain operator's input and output ports, as well as state information maintained.

3.1. Operator ports

An InfoSphere Streams operator's inputs and outputs are defined by ports. The Louvain operator described here has two input ports and two corresponding output ports. While it is not a requirement by the SPL language to have the input and output ports coupled, it is a convenient way to program this specific operator and an intuitive way to describe them.

3.1.1. Edge Stream Ports

The edges observed on the edge stream input port can be thought of as three different cases. The first is where neither vertex in the edge has been previously observed. In this case, a new community is assigned to the vertices incident upon this new edge.[4]

The second case is when both vertices of an edge have been previously assigned communities. The edge is added to the graph, and each node i of the edge is assigned the community which maximizes the change in the standardized modularity of McCloskey resulting from adding node i to community C given in equation (3.1). This function is called on each of the node's neighboring communities C, and by accepting the maximum value of this function, a node is assigned a community which maximizes modularity at a given time t. It should be noted that the numerator of equation (3.1) is equivalent to the function used in the original Louvain paper [BGLL08], except for the addition of the t superscripts, necessary when computing on streaming data.

$$\Delta Q^t_{i\to C} = \frac{N^t X^t_{iC} - X^t_{C+} X^t_{i+}}{\sqrt{N^t X^t_{C+} X^t_{i+}}} \tag{3.1}$$

The Louvain operator is configurable with equation (3.1) at runtime. The parameter N^t is the overall weight of the graph or sample size of the contingency table associated with the graph's matrix, and defined as the sum of the weights (i.e. the degrees or transition counts) of the vertices.

[4]This case will be explored and documented in more detail after the implemention of a second version of our algorithm, *SCREAM 2*, is on-line.

We may interpret X_{iC}^t as the volume of communication from node i to community C at time t, X_{C+}^t is the weight of a community defined by the sum of the weights of all edges incident to its member vertices at a given time t. Finally, X_{i+}^t is the sum of the weights of all the edges incident on node i at time t.

The third and final case is when one node from an incoming edge has already been assigned a community and one has not. The operator is designed to accept a degree threshold on a node before assigning a community to it. The degree threshold allows nodes with low degree to be filtered out of a graph and removed as a factor in determining community definitions. This becomes helpful when there are many leaf nodes, and it is undesirable for a community to be defined as a grouping of or include leaf nodes.

After the edge is processed and communities are assigned to each of its vertices, an output tuple summarizing the activity is submitted to the edge stream output port. Included in the output port is the pertinent information about the edge being processed (nodeA, nodeB, time, etc.), effects on the partitions modularity score (deltaQ, communityCumulativeQ, communityAbsCumulativeQ), and the community assignments for both nodes. The state information concerning the graph and communities is output on the command port.

3.2. Operator State Variables

In addition to an adjacency map to represent the observed graph structure, state information about the graph, nodes, and communities is also maintained within the Louvain operator. The state information is used to quantify an edge's contribution to modularity.

The Louvain algorithm strives to optimize the objective modularity function, and here via the Louvain operator, it is adapted to optimize it at a given time. The modularity change in a given partition after observing a new edge is defined in equation (3.2), where Q_c^t is the McCloskey standardized modularity after the new edge observation, and Q_c^{t-1} is the modularity previous to the new edge. Only those communities affected by the new edge need their modularity scores updated.

$$cumulativeIncQ_c = \sum_{t=1}^{T} Q_c^t - Q_c^{t-1}, \tag{3.2}$$

$$cumulativeIncAbsQ_c = \sum_{t=1}^{T} \left| Q_c^t - Q_c^{t-1} \right|. \tag{3.3}$$

Equation (3.2) accounts for the net change in modularity of a partition, and as a result does not account for overall activity that a partition is involved in. To evaluate the overall activity new edges have had on the original partition, equation (3.3) simply takes the cumulative absolute value of an edge change. This metric is updated as part of a community's state information in the same manner as equation (3.2)

4. Results

We present some results on five examples. The first two are simple hand-crafted examples. The remaining three were constructed using a random graph generator program. Some details about this program are given in Appendix C. In all of the examples, we start with a graph having predefined community structure which is input into the algorithm. Next a file of randomly generated edges produced by the graph generator is input into the algorithm one edge at a time and the features listed in Table 2.1 of Section 2.3 are computed.

4.1. Toy Examples

The results presented here consist of two simple hand-crafted examples. The first illustrates the dynamic nature of streaming community detection and some of the output that *SCREAM* can produced. The second example shows that the *SCREAM* algorithm is actually deterministic given the initial starting community assignments and the order of the edge stream.

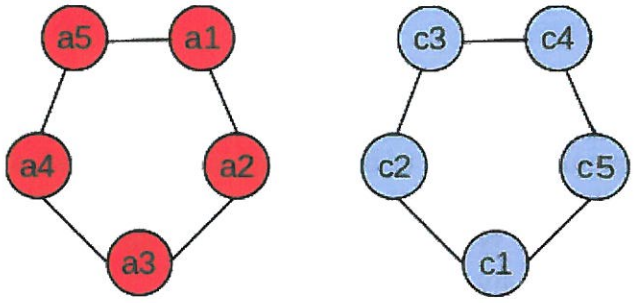
4.1.1. How *SCREAM* Determines Community Membership As New Edges Arrive

In this example, we discuss significant points during the evolution of communities over time as more edges are observed on the edge stream. As described in Section 3, the *STREAM* operator requires an initial graph to create community assignments. Figure 3(a) shows the initial graph with the communities indicated by color. After *SCREAM* computes the initial communities (using the McCloskey standardized modularity), the communities receive updates to their membership after each new edge is observed on the stream. After two inter-community edges from the edge stream have been processed, edge $(a1, c3)$ at time $t = 3$ (colored green) causes node a1 to move from the red community to the blue community as indicated in Figure 3(b). This is due to the overall modularity increase, as shown at $t = 3$ in Figure 4.[5] While modularity decreases for the red community, it increases at a greater rate for the blue community. Figure 4 shows the change in the community node counts after each new edge has been processed.

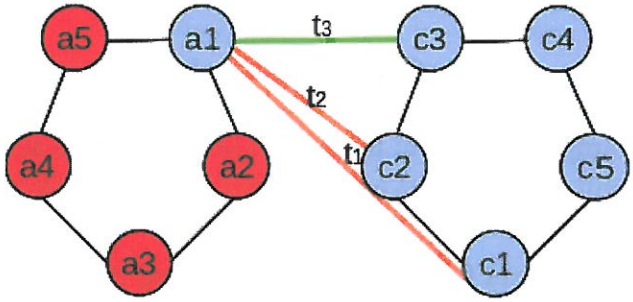
At $t = 8$, eight intra-community edges have been observed in the red community, making it more dense, and increasing it's ability to attract new members. Node a1 is now more connected to the red community than blue, adjacent to four red and three blue nodes respectively. As a result of these two factors, a1 is assigned back to the red community upon observation of the edge at $t = 8$, highlighted in green in Figure 3(c). The overall increase in modularity (communityCumulativeQ) can be observed in Figure 4, with both the red and blue community's modularity increasing after a1 moves back to the red community.

Following a1's re-assignment to the red community, inter-community edges are introduced at $t \geq 9$. The resulting graph at $t = 15$ is shown in Figure 3(d). Community membership stays stable (communityNodeCount) since both communities modularity scores decrease from $t = 8$ onward as shown in Figure 4.

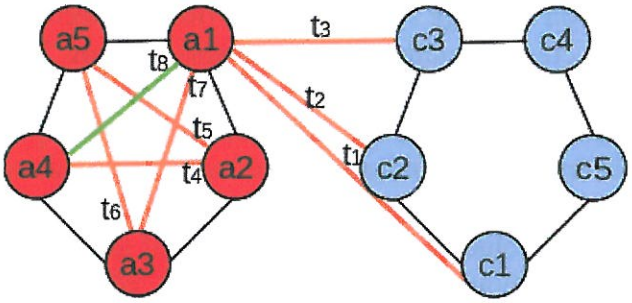
[5]We note that in Figure 4, the notation "_3" and "_9" refer to the red and blue initial communities, respectively. The reason is that community are labeled by one of its members and in the original graph, node "a3" was just "3" and node "c4" was "9." Finally, time t is just the index of a new edge on the stream.



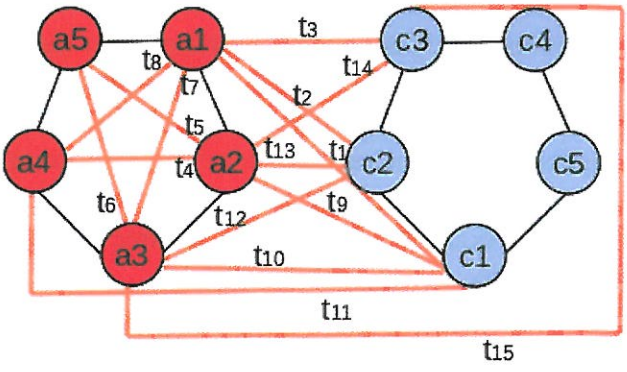
(a) Initial Graph at $t = 0$.



(b) Streaming Graph at $t = 3$.



(c) Streaming Graph at $t = 8$.



(d) Streaming Graph at $t = 15$.

Figure 3: Evolution of Community Structure.

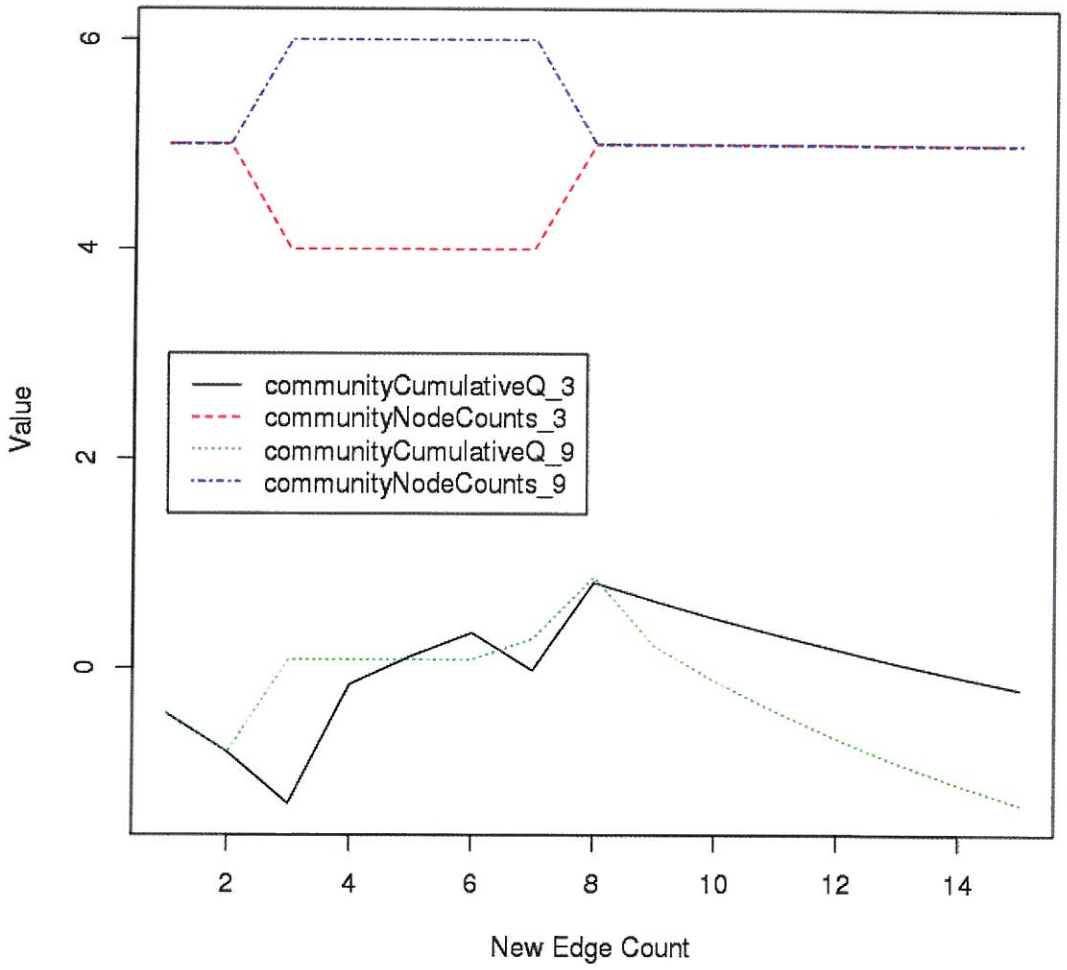


Figure 4: Streaming Standardized Louvain

4.1.2. Maximum Number of Community Membership Changes

In this example we show how a single edge can affect the modularity and community membership of four communities. Figure 5(a) shows communities, **a**, **b**, **c** and **d**, identified by the Louvain algorithm as the initial partition with edges labeled by the order in which they arrive.

Figure 5(b) captures the arrival of edges 17-20 to the initial graph, linking community **a** (red) to **b** (blue) and community **c** (green) to **d** (orange). Since nodes a4 and d4 are now both adjacent to two communities, the deltaQ function (Section 3.1.1, equation (3.1)) is evaluated to determine community assignment. Since both nodes are symmetrically connected to their two adjacent communities, the deltaQ functions indicate an equal change in modularity for the two possible communities the nodes are assigned. However, due to the iteration over a hashtable data structure in the deltaQ function, a4 is assigned to the first community with largest deltaQ, which is the **a** community. Thus the SCREAM algorithm is deterministic in which community it will assign when faced with equal deltaQ-scoring communities, i.e. no random community assignment is done in these cases. Another example of this determinism will be given in Section 4.2.2.

Edge 21, (a2,c2), shown in Figure 5(c) introduces an inter-community edge between communities **a** and **b** through nodes a2 and c2. This inter-community edge decreases the modularity score of both communities **a** and **b**, since the density of the both communities decrease. However this doesn't change the community assignments of a2 or c2 as they are still more strongly connected to the **a** and **c** communities, respectively.

Finally, in Figure 5(d) edge 22 is introduced between nodes a4 and c4, making them connected to three communities each. The deltaQ functions are evaluated for each node using equation (3.1) reproduced below.

$$\Delta Q^t_{i\to C} = \frac{N^t X^t_{iC} - X^t_{C+} X^t_{i+}}{\sqrt{N^t X^t_{C+} X^t_{i+}}} \tag{4.1}$$

We compute $\Delta Q^t_{i\to C}$ at time $t = 22$ corresponding to the arrival of edge $(a4, c4)$ under the following assumptions. We evaluate community membership for the left node, $a4$, first. In doing so, we suspend membership of $a4$ in its current community so that in the computations, $a4$ is treated as an isolated community. We next evaluate community membership for the right node, $c4$, after any change in membership for $a4$ has been recorded. Again, we suspend the membership of $c4$, treat it as an isolated community, and evaluate its community membership. We have

$$X_{a4,\mathbf{a}} = 2, X_{\mathbf{a}+} = 9, X_{a4+} = 5 \Longrightarrow \Delta Q^{22}_{a4\to\mathbf{a}} = \frac{44\cdot 2 - 9\cdot 5}{\sqrt{44\cdot 9\cdot 5}} = 0.9664; \tag{4.2}$$

$$X_{a4,\mathbf{b}} = 2, X_{\mathbf{b}+} = 8, X_{a4+} = 5 \Longrightarrow \Delta Q^{22}_{a4\to\mathbf{a}} = \frac{44\cdot 2 - 8\cdot 5}{\sqrt{44\cdot 8\cdot 5}} = 1.1442; \tag{4.3}$$

$$X_{a4,\mathbf{c}} = 1, X_{\mathbf{c}+} = 14, X_{a4+} = 5 \Longrightarrow \Delta Q^{22}_{a4\to\mathbf{c}} = \frac{44\cdot 1 - 14\cdot 5}{\sqrt{44\cdot 14\cdot 5}} = -0.468; \tag{4.4}$$

$$X_{a4,\mathbf{d}} = 0, X_{\mathbf{d}+} = 8, X_{a4+} = 5 \Longrightarrow \Delta Q^{22}_{a4\to\mathbf{d}} = \frac{44\cdot 0 - 8\cdot 5}{\sqrt{44\cdot 8\cdot 5}} = -0.9535. \tag{4.5}$$

The max deltaQ value is 1.1442 so we assign $a4$ to community **b**. Next, we consider $c4$.

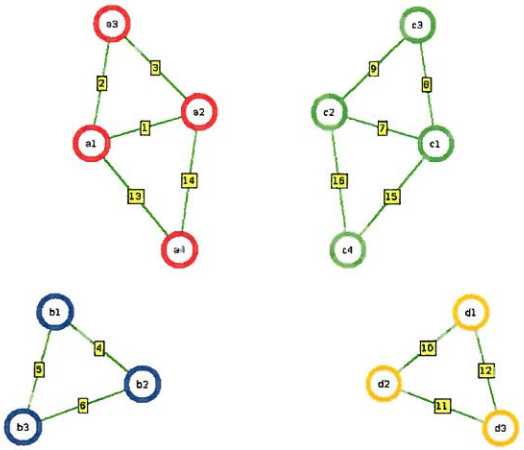
$$X_{c4,\mathbf{a}} = 2, X_{\mathbf{a}+} = 9, X_{c4+} = 5 \Longrightarrow \Delta Q^{22}_{c4\to\mathbf{a}} = \frac{44\cdot 1 - 9\cdot 5}{\sqrt{44\cdot 9\cdot 5}} = -0.02247; \tag{4.6}$$

$$X_{c4,\mathbf{b}} = 0, X_{\mathbf{a}+} = 13, X_{c4+} = 5 \Longrightarrow \Delta Q^{22}_{c4\to\mathbf{a}} = \frac{44\cdot 0 - 13\cdot 5}{\sqrt{44\cdot 13\cdot 5}} = -1.215; \tag{4.7}$$

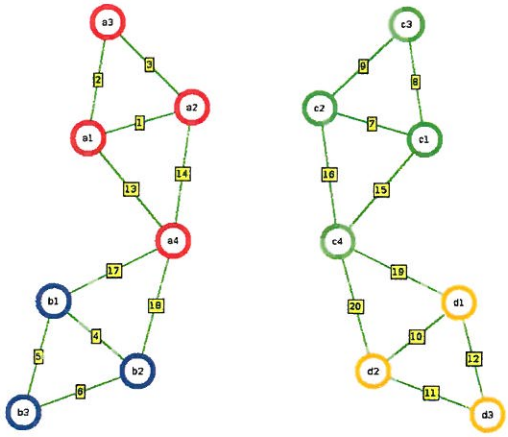
$$X_{c4,\mathbf{c}} = 2, X_{\mathbf{c}+} = 9, X_{c4+} = 5 \Longrightarrow \Delta Q^{22}_{c4\to\mathbf{c}} = \frac{44\cdot 2 - 9\cdot 5}{\sqrt{44\cdot 9\cdot 5}} = 0.9664; \tag{4.8}$$

$$X_{c4,\mathbf{d}} = 2, X_{\mathbf{d}+} = 8, X_{c4+} = 5 \Longrightarrow \Delta Q^{22}_{c4\to\mathbf{d}} = \frac{44\cdot 2 - 8\cdot 5}{\sqrt{44\cdot 8\cdot 5}} = 1.1442. \tag{4.9}$$

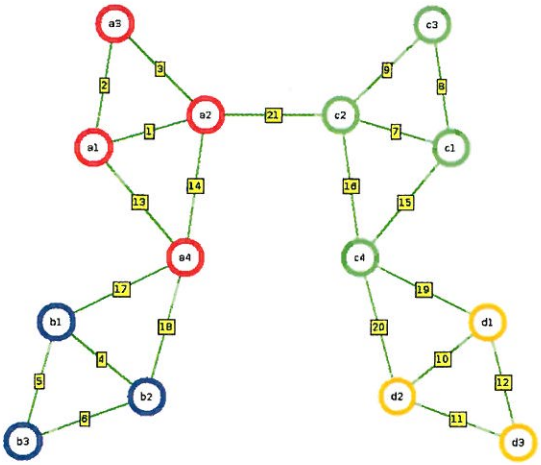
Thus node $c4$ should be assigned to community **d**. Notice that after all computations have been completed for edge $(a4, c4)$, each node is assigned to a different community: a4 assigned to **b** and c4 assigned to **d**. Interestingly, neither node is moved to one of the other's adjacent communities. This directly contradicts Lemma 2 of Nguyen et al. [NDXT11], p. 5 which claims that the best new candidate for community membership for $a4$ (resp. $c4$) is the community of $c4$ (resp. $a4$) if memberships change. They are in fact moved to the communities which had, at edges 18 and 20, scored equally to the nodes' original communities. The assignment decreases both nodes' former communities' intra-community weights (X_{C+}), and increases their newly assigned communities' intra-community weights, resulting in a modularity change for all four communities. This is significant, as this example demonstrates the case where a single edge can affect the modularity scores of four communities, which is the maximum number of communities whose membership can be changed with one edge.



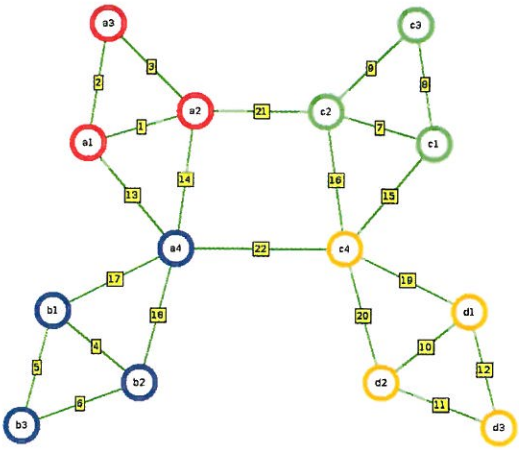
(a) Initial graph.



(b) Graph at edge 20.



(c) Graph at edge 21



(d) Graph at edge 22

Figure 5: Evolution of Graph

4.2. Simulation Results

Here we present results from two simulation examples produced by our graph generation program (see Appendix C for more detail and definitions of terminology such as *skeleton graph*).

4.2.1. A Simple Example

Figure 6 shows the initial graph and Figure 7 the final graphs. The initial graph 6 has 12 nodes and 13 edges arranged into three connected components whose skeletons are binary trees and constitute three distinct communities. The communities are labeled by randomly choosing one member node label. Thus community **a** is labeled by $a5$, community **b** by $b7$, and **c** by $c11$. The graph evolves from this initial graph by randomly adding twenty inter-community edges. Two partitions are shown in Figures 7(a) and 7(b) corresponding to the global Louvain and *SCREAM* algorithms. Clearly the algorithms produce quite different results.

If we run the (global) Louvain algorithm multiple times on the final graph, the top level (level 2) can output three distinct partitions due to the random order of sweeping across the nodes during phase 1. The three distinct possible partitions are:

$$\{a1, a3, a4, a5, b7\} \quad \{a2, b6, b8, c9, c10, c11, c12\} \tag{4.10}$$

$$\{a1, a2, a3\} \quad \{a4, c9, c11\} \quad \{a5, b6, b7, b8, c10, c12\} \tag{4.11}$$

$$\{a1, a2, a3, a5, b7\} \quad \{a4, c9, c11\} \quad \{b6, b8, c10, c12\}\,. \tag{4.12}$$

In Figure 7(a) the nodes together according to partition (4.12).

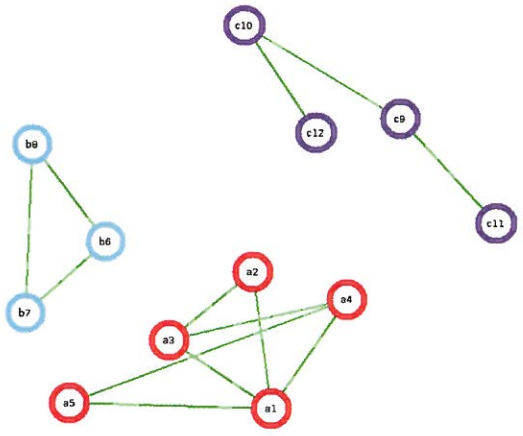
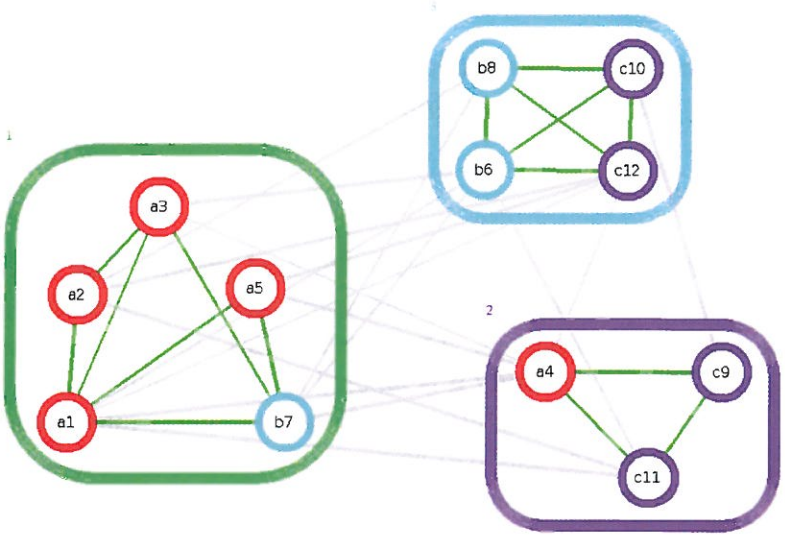
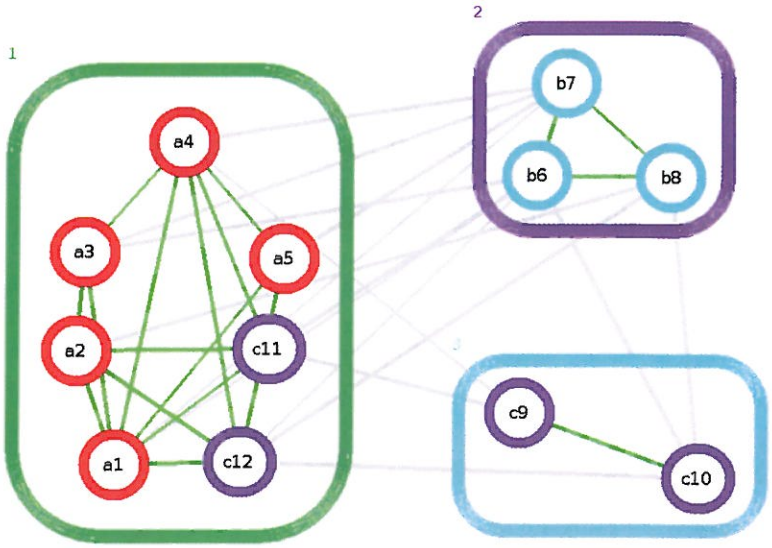


Figure 6: Initial Community Configuration.



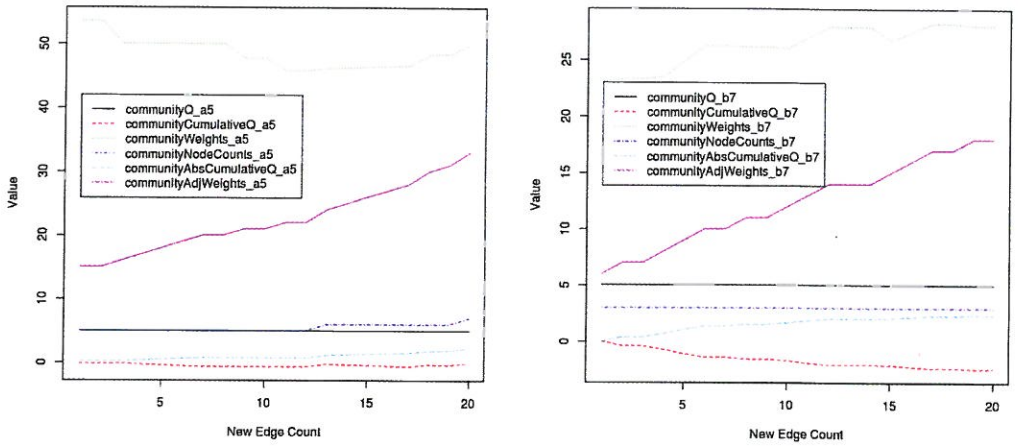
(a) Final global Louvain partition.



(b) Final *SCREAM* Partition.

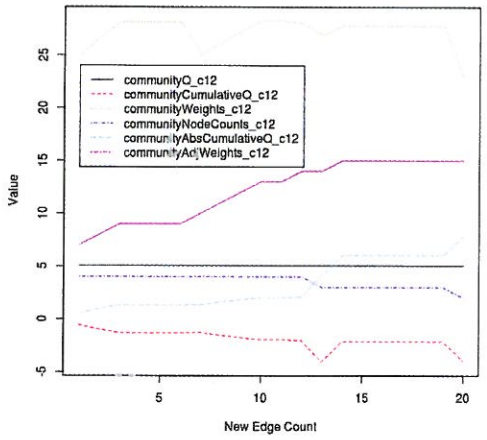
Figure 7: Final Graphs and their partitions.

Figure 8 plots the statistics for each community. The plots for *communityQ*, the community modularity, are constant in Figures 8(a) - 8(c) since they are computed once for each initial community before processing the new edge stream. The *communityNodeCounts* plots show that **a5** gains nodes at new edges 13 and 20, **b7** stays constant at 3, and **c12** looses nodes at edges 13 and 20. Now edge 13 is $(a1, c12)$, and at that time, the algorithm removes node c12 from **c** and adds it to community **a**. The effect of these operations on the incremental modularity ($incQ$) can be seen in Figure 9 at indices 25 and 26, respectively. We have marked the high and low points on this graph with the corresponding operations performed by the *SCREAM* algorithm.



(a) community a5

(b) community b7



(c) community c12

Figure 8: Plots of community statistics.

Edge 20 is $(a1, c11)$ and at that time, the algorithm removes node c11 from community **c** and adds it to **a**. Thus the *SCREAM* algorithm at edge count 20 gives the following communities:

$$\{a1, a2, a3, a4, a5, c11, c12\}\ \{b6, b7, b8\}\ \{c9, c10\}\,. \tag{4.13}$$

The arrangement in Figure 7(b) exhibits this partition.

This example illustrates the difference between the global Louvain and *SCREAM* algorithms. In the global case, there are three possible partitions which result from a random nature of the Louvain algorithm. The order in which new edges arrived is not considered. On the other hand, the time sequencing of the new edges plays a central role in the *SCREAM* algorithm. The partition that results reflects community membership resulting from significant changes in modularity of the communities involved in a particular edge. This gives some evidence that a sequential streaming approach used in *SCREAM* is not the same as a snapshot approach to node partitioning. That is, even if we could do a global Louvain partitioning after each new edge arrives, it probably would not give the same result as *SCREAM* partitioning. In the next section we explore in more detail the differences between a global community detection algorithm and *SCREAM*.

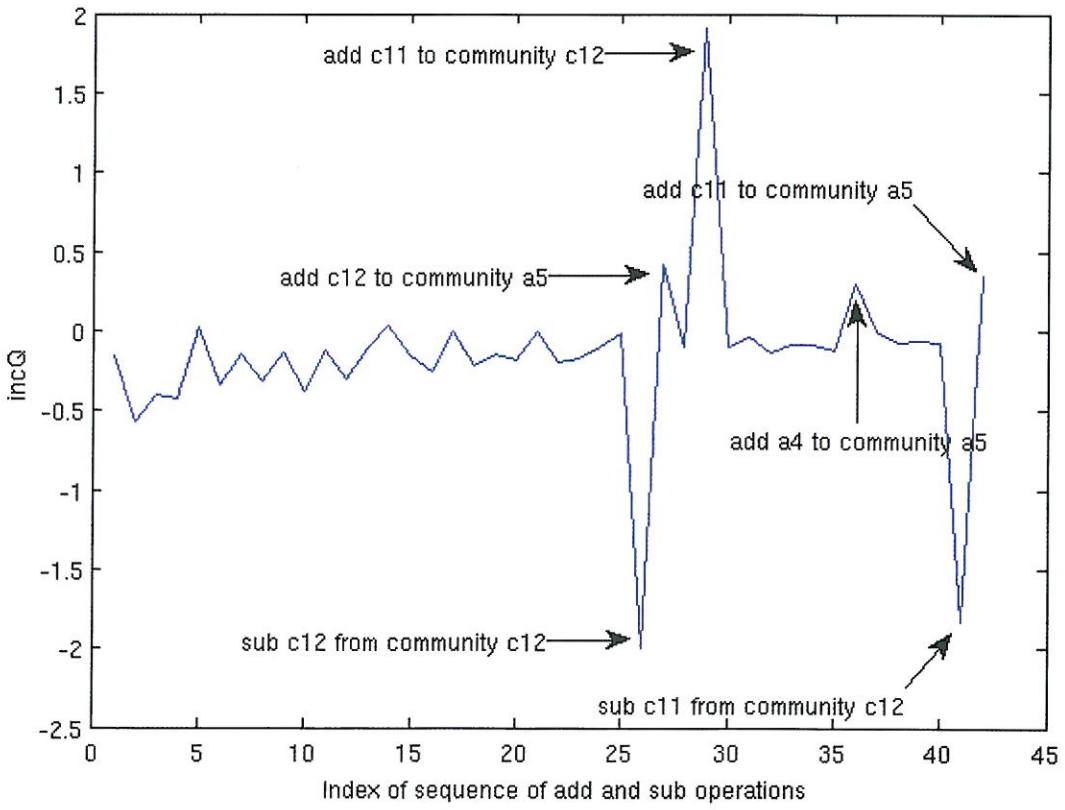
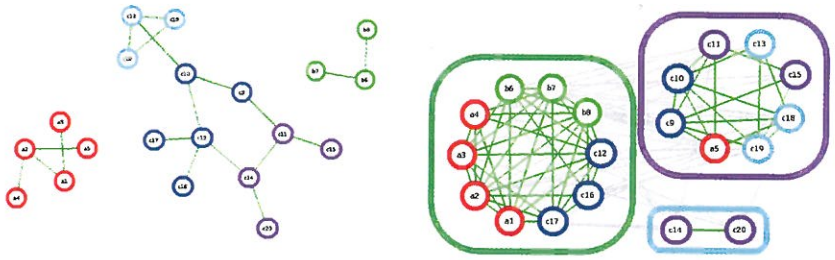


Figure 9: $incQ$ values for *add* and *sub* calc operations.



(a) Initial graph (19 edges). (b) Final graph (99 edges).

Figure 10: Initial and final graph with communities.

4.2.2. Sensitivity to Order

In this example, we give another demonstration of the deterministic nature of *SCREAM* by showing that the order in which edges arrive affects the partitioning of the graph. The initial graph shown in Figure 10(a) contains 20 nodes and 19 edges arranged into three connected components of orders 3, 5, and 12 respectively. The skeleton graphs for these components are binary trees. The initial graph consists of five communities indicated by colored nodes in Figure 10(a) with the largest component partitioned into three communities. The final graph consists of 80 new edges added at random. This gives a total of 99 edges with 62 inter-community edges (14 between a and b, 26 between a and c, and 22 between b and c), and 37 intra-community edges. The final graph contains three communities of sizes 2 (c14,c20), 8 (a5,c9,c10,c11,c13,c15,c18,c19) and 10 (a4, etc.) as determined by *SCREAM*. Figure 10(b) exhibits these three communities with the intra-community edges colored green and inter-community edges colored gray.

The initial communities obtained from the standardized Louvain algorithm are:

$$\mathbf{a4} = \{a1, a2, a3, a4, a5\} \tag{4.14}$$
$$\mathbf{b7} = \{b6, b7, b8\} \tag{4.15}$$
$$\mathbf{c18} = \{c9, c10, c12, c16, c17\} \tag{4.16}$$
$$\mathbf{c19} = \{c13, c18, c19\} \tag{4.17}$$
$$\mathbf{c20} = \{c11, c14, c15, c20\}\,. \tag{4.18}$$

Note that node c18 belongs to community **c19**! This is an unfortunate consequence of the initial labeling of communities and the order of arrival of the initial 19 edges.[6]

An important point to make before we begin our demonstration that edge order affects the community structure is the following. The *SCREAM* algorithm is **deterministic** in the sense that if we start with the same initial community structure, then multiple runs of the algorithm on the same stream of new edges will produce identical output and community assignments per edge. In this example, one hundred runs of the Louvain algorithm on the initial graph always produce

[6]The problem of labeling communities in a dynamic environment where communities can merge together, split apart or die out is difficult. Our approach is clearly not optimal!

the same initial community structure shown in Figure 10(a) in contrast to example 4.2.1 where running the standardized Louvain algorithm multiple times produced three different partitions. In addition, multiple runs of the *SCREAM* algorithm on the graph in this example produce identical community assignments, i.e. the community assignments at each new edge are exactly the same for each run. Finally, we remark that since *SCREAM* determines the initial community structure by running the standardized Louvain algorithm on the initial graph, the random starts of the Louvain algorithm can produce different initial partitions. Therefore, given different initial start community assignments, *SCREAM* may output different communities with the same new edge stream.

Now to demonstrate the effect of new edge order, we construct two sequential graphs, $\mathcal{G}_{\text{ord}}$ and $\mathcal{G}_{\text{scr}}$ from the initial graph and the new edge sequence as follows. Both graphs have exactly the same nodes and edges as the final graph. The difference between the two is the order in which the new edges are presented to the *SCREAM* algorithm. $\mathcal{G}_{\text{ord}}$ is the final graph with the original edge order. $\mathcal{G}_{\text{scr}}$ is constructed by segmenting the ordered list of 80 new edges from $\mathcal{G}_{\text{ord}}$ into successive groups of twenty edges. Each group of twenty (ordered by time) edges is then scrambled using a random permutation and presented in the new scrambled order[7].

Since the level 1 partition produced by the Louvain algorithm depends on the order of the sequential first pass over the vertices, different final partitions can be output at the top level for each run of the algorithm. How different these partitions can be is indicated in Figure 11. This figure shows a histogram of the Adjusted Rand Index (ARI)[8] of pairs of partitions of the vertex set of $\mathcal{G}_{\text{ord}}$ produced by 200 runs of the standardized Louvain algorithm. This resulted in 200 top level partitions (usually level 2). For each pair of partitions $\mathcal{P}_1$ and $\mathcal{P}_2$, produced by distinct runs, the score, $\text{ARI}(\mathcal{P}_1, \mathcal{P}_2)$, was computed. The ARI is a measure of agreement between two partitions which is bounded above by 1. A score of 1 indicates that the two partitions have the same number of communities, and the communities are identical after possibly permuting their indices. ARI scores were computed for each of the $\binom{200}{2} = 19,900$ pairs of partitions produced by the 200 runs of the Louvain algorithm. The histogram in Figure 11 was produced by binning these scores into 20 bins. Although the mode of this distribution is 1, it is clear that many of the partitions produced by Louvain on $\mathcal{G}_{\text{ord}}$ are very different with a substantial number of ARI scores below 0.3.

What about the scrambled version of the graph, $\mathcal{G}_{\text{scr}}$? Again, 200 runs of Louvain on $\mathcal{G}_{\text{scr}}$ produced top-level partitions of the vertex set, and these in turn yield 19,900 ARI scores for pairs of partitions produced by distinct runs of Louvain. Denote by $V(\mathcal{G}_{\text{scr}})$ the vector of these sorted scores, and let $V(\mathcal{G}_{\text{ord}})$ denote the vector of sorted scores produced in the previous paragraph. To compare the distributions of these two sets of ARI scores, a two-sample Kolmogorov-Smirnov test was run on the sorted vectors using the programming language R. To avoid problems caused by tied scores, we jittered the entries of the vectors. Using the two-sided option for the alternative hypothesis, we obtained the test statistic $D = 0.0327$ with p-value $1.204e-09$. This gives strong evidence that the two sets of ARI scores come from the same distribution, and hence the partitions produced by Louvain on $\mathcal{G}_{\text{scr}}$ and $\mathcal{G}_{\text{ord}}$ are very similar.

To provide further evidence that the order of the edges does not affect the partitions produced by Louvain, we did 19,900 runs of Louvain on each of the graphs, $\mathcal{G}_{\text{scr}}$ and $\mathcal{G}_{\text{ord}}$. For each run,

[7]This experiment was suggested by James Fairbanks of the Georgia Institute of Technology.

[8]See Appendix B for more information on the ARI.

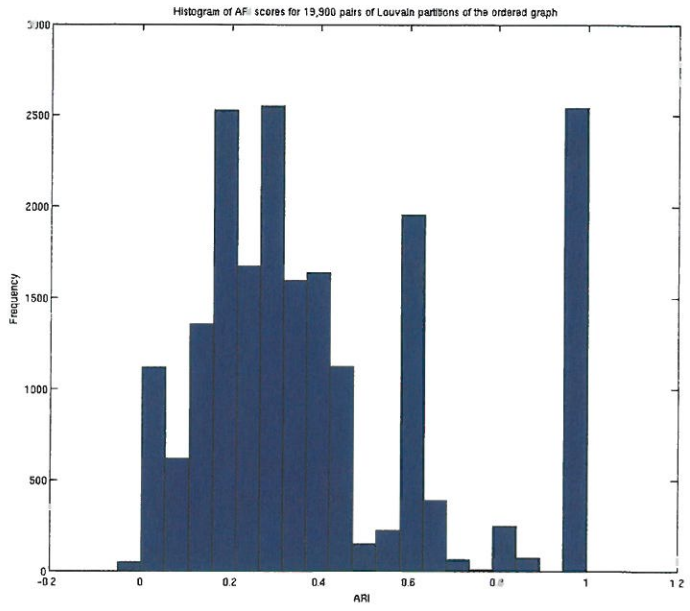


Figure 11: Histogram of ARI scores of paired standardized Louvain partitions of the ordered graph.

we computed the ARI score of the pair of partitions. This produced 19,900 scores which are sorted and stored in a vector, $V\left(\mathcal{G}_{\text{ord}}, \mathcal{G}_{\text{scr}}\right)$. Two-sample Kolmogorov-Smirnov tests were then run on the distinct pairs of sorted vectors of ARI scores. The results are shown in Table 4.1. The p-values indicate that the three sets of ARI scores all come from the same distribution with extremely high confidence. Therefore, it is highly likely that the distribution of partitions produced by the standardized Louvain algorithm is the same regardless of the order of arrival of the edges.

On the otherhand, the *SCREAM* algorithm was designed to be dependent on edge order. If we examine the community assignments given by the SCREAM algorithm at the breakpoints for the ordered and scrambled graphs, then the communities can be very different as shown in Figures 12 and 13. Although both partitions at breakpoint 39 (new edge 20) have five communities (Figures 12(a) and 12(b)), by the second breakpoint at edge 59 (new edge 40) the scrambled edge sequence graph (Figure 12(d)) has essentially reduced its partition to three communities with one singleton community containing node a5 that is about to be merged into the lower left-hand community of Figure 12(d). Figures 12(c) and 13(a) show that the ordered edge sequence graphs still have five communities at breakpoints 59 and 79 (new edges 40 and 60), while the scrambled edge

Vector Pairs	KS statistic D	p-value
$V\left(\mathcal{G}_{\text{ord}}\right), V\left(\mathcal{G}_{\text{scr}}\right)$	0.0327	1.204e-09
$V\left(\mathcal{G}_{\text{ord}}\right), V\left(\mathcal{G}_{\text{ord}}, \mathcal{G}_{\text{scr}}\right)$	0.0357	1.856e-11
$V\left(\mathcal{G}_{\text{scr}}\right), V\left(\mathcal{G}_{\text{ord}}, \mathcal{G}_{\text{scr}}\right)$	0.0295	5.696e-08

Table 4.1: Kolmogoroff-Smirnoff test results on pairs of ARI vectors.

sequence graph at breakpoint 79 has only three communities. Finally, at the end, both graphs have three communities but with drastically different orders as can be seen by comparing Figures 13(c) and 13(d). Furthermore, if we compute the Adjusted Rand Index (ARI) for each pair of partitions produced by *SCREAM* at the breakpoints, we obtain the following scores:

1. ARI(ordered_39,scrambled_39)=0.344695
2. ARI(ordered_59,scrambled_59)=0.490295
3. ARI(ordered_79,scrambled_79)=0.478097
4. ARI(ordered_99,scrambled_99)=0.006864.

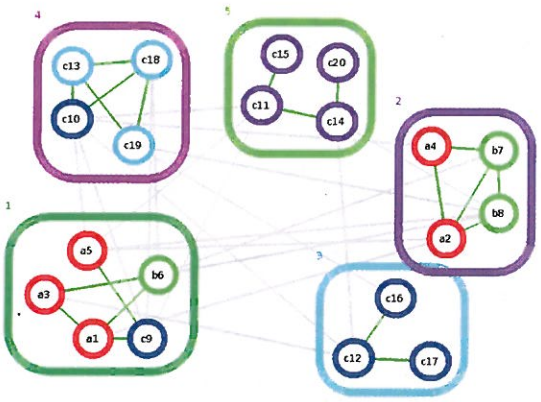
Hence by the time the final graph (edge 99) is realized, the community structures given by the partitions for the ordered and scrambled graph are very different as indicated by an ARI close to 0.

It is clear that the ordered and scrambled graphs are moving towards a complete graph with a single clique as the final community. Since the graph simulator only emits unique edges, this implies that eventually only inter-community edges can be generated connecting the remaining communities. However, the rate at which the single community appears obviously depends on the relative proportions of inter- versus intra-community edges that arrive over time.

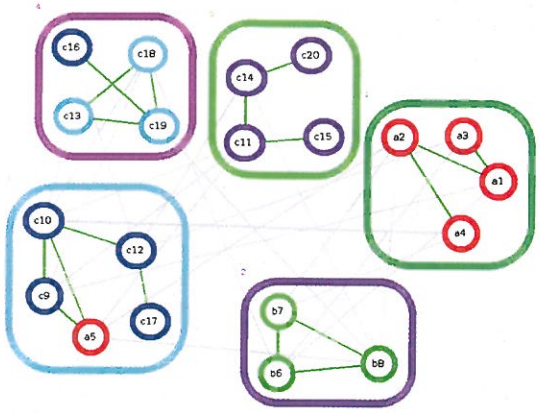
Figure 16 plots the inter-community edge proportion as each new edge is generated and added to the ordered and scrambled edge sequence graphs. The graph shows that the scrambled edge sequence graph's proportion is consistently larger than that of the ordered edge sequence graph for the first half of the new edges (i.e. up to new edge 40) . This suggests that the scrambled edge sequence graph is becoming more "random" due to the larger number of inter-community edges arriving. By the time the 40th new edge arrives, one large community has formed for the scrambled graph and dominates thereafter. This contrasts with the two ordered edge sequence communities in Figure 13(c) that evolve with about the same number of nodes.

The differences between the emerging community structures of the ordered and scrambled graphs can also be seen in the plots of Figures 14 and 15. Here the horizontal axes give the index of the new edges (1-80) as they arrive to join the initial 19 edges in Figure 10(a). Figure 14 shows the nodes counts for the five initial communities over time for both the ordered and scrambled edge sequence graphs. The figures in 15 show the changes in the communityCumulativeQ and communityAdjacencyWeight features as each new edge is generated. Note the rapid increase in communityCumulativeQ starting about new edge time 65 for community c19 for the scrambled graph Figure 15(b). By comparing this with the communityNodeCounts plot in Figure 14(b), we see that community c19 is rapidly loosing members during this time. With each loss of member from community c19, we suspect that the intra-community edges for the remaining nodes increase in importance for modularity.

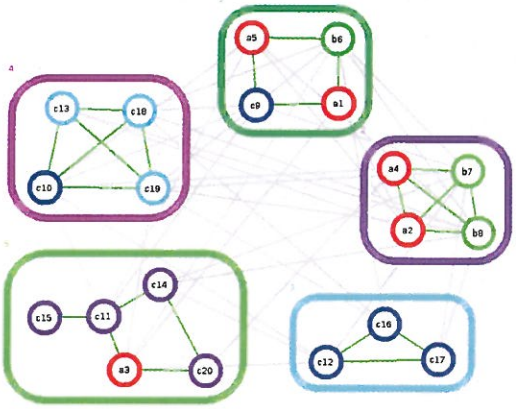
Finally, Figure 17 shows the incremental modularity changes with the addition of each new edge for the ordered and scrambled edge sequence graphs. The larger number of peaks in the incQ scrambled sequence may indicate that the larger community c20 is forming and absorbing the smaller communities. The contrast between these two graphs is striking and needs to be further explored using tools from time series analysis.



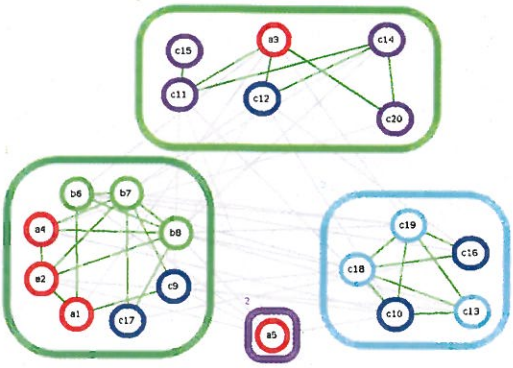
(a) Partition at edge 39 (ordered).



(b) Partition at edge 39 (scrambled).

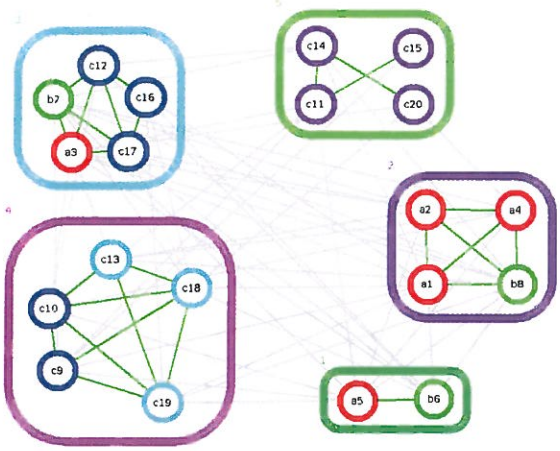


(c) Partition at edge 59 (ordered).

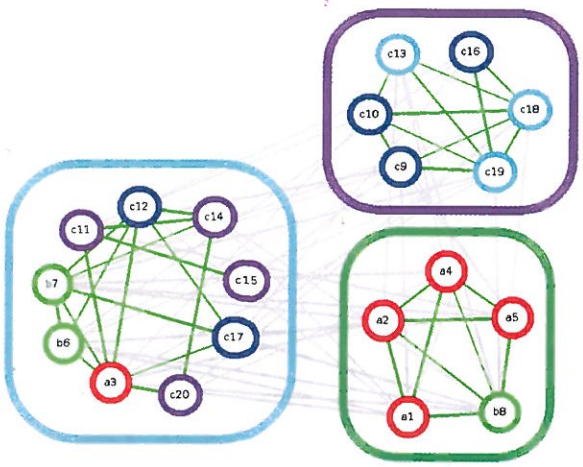


(d) Partition at edge 59 (scrambled).

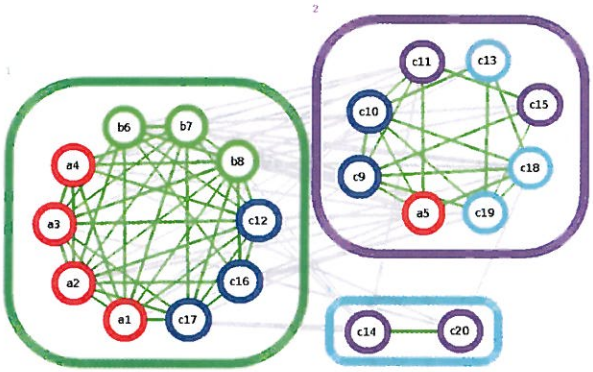
Figure 12: Partitions at breakpoints 39 and 59 for $\mathcal{G}_{\text{ord}}$ and $\mathcal{G}_{\text{scr}}$.



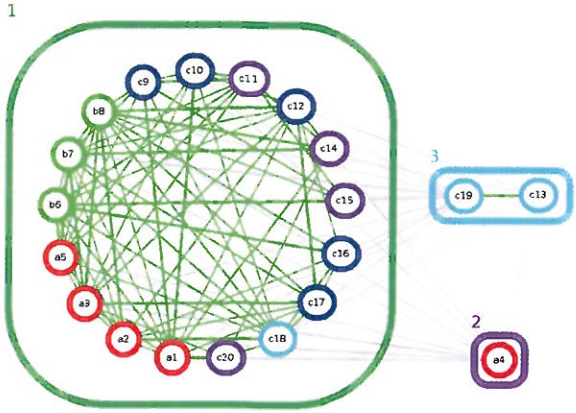
(a) Partition at edge 79 (ordered).



(b) Partition at edge 79 (scrambled).

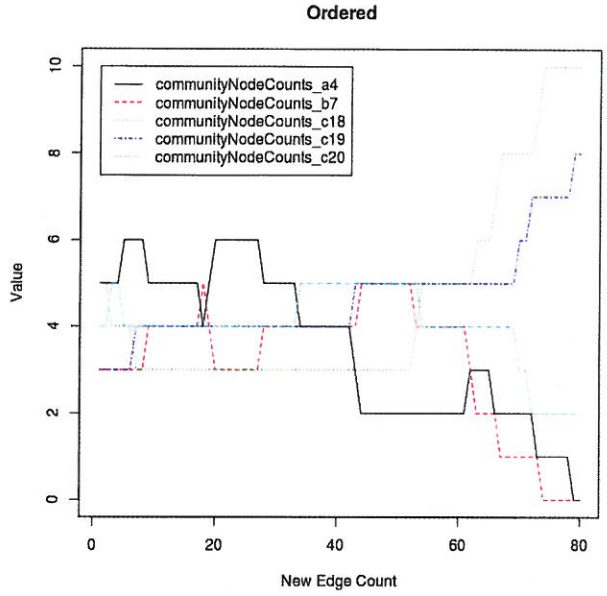


(c) Partition at edge 99 (ordered).

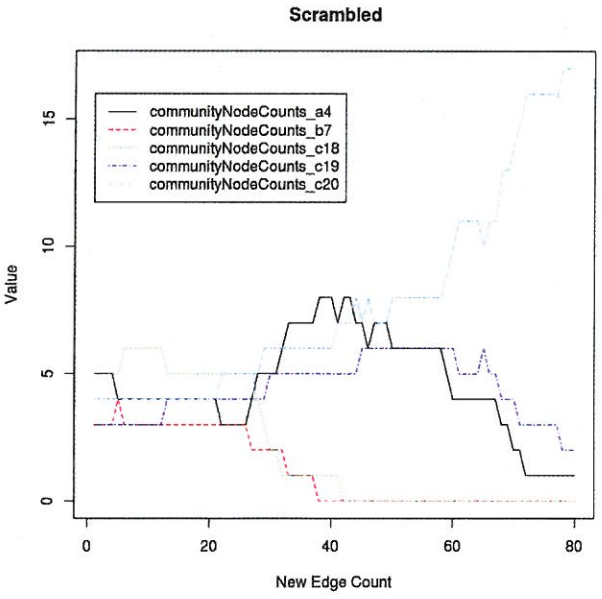


(d) Partition at edge 99 (scrambled).

Figure 13: Partitions at breakpoints 79 and 99 for $\mathcal{G}_{\text{ord}}$ and $\mathcal{G}_{\text{scr}}$.

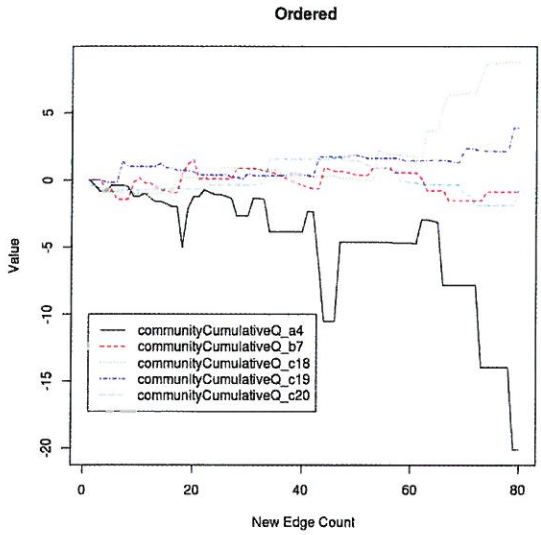


(a) CommunityNodeCounts.

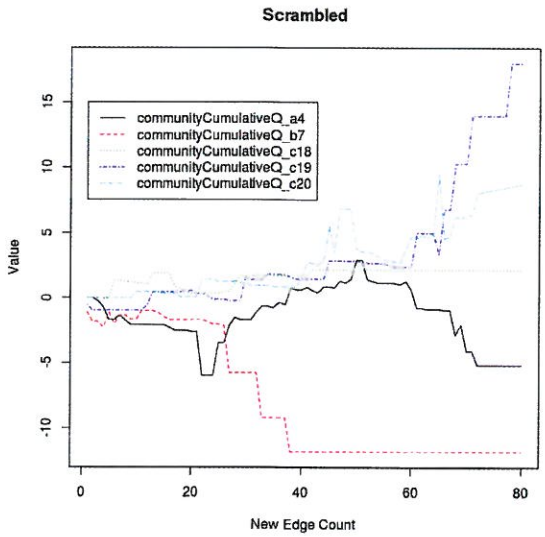


(b) CommunityNodeCounts.

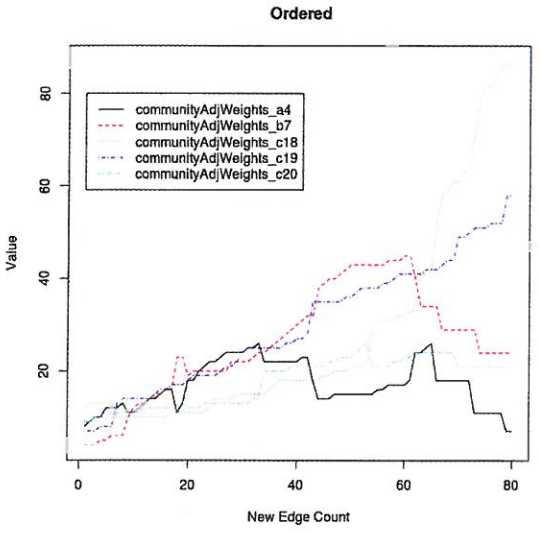
Figure 14: Community Statistics for Ordered and Scrambled Graphs.



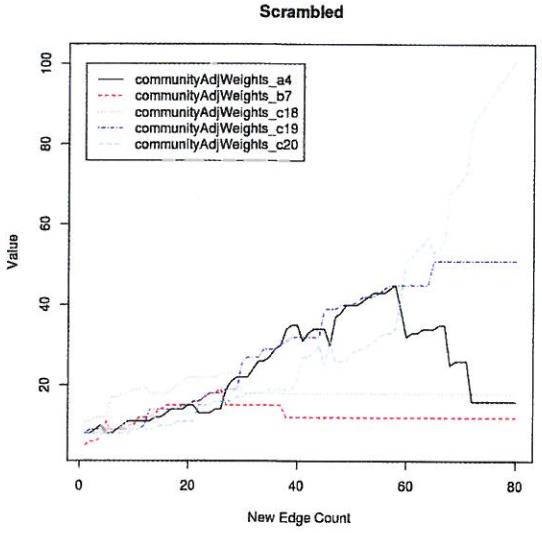
(a) CommunityCumulativeQ.



(b) CommunityCumulativeQ.



(c) CommunityAdjWeights.



(d) CommunityAdjWeights.

Figure 15: Community Statistics for Ordered and Scrambled Graphs.

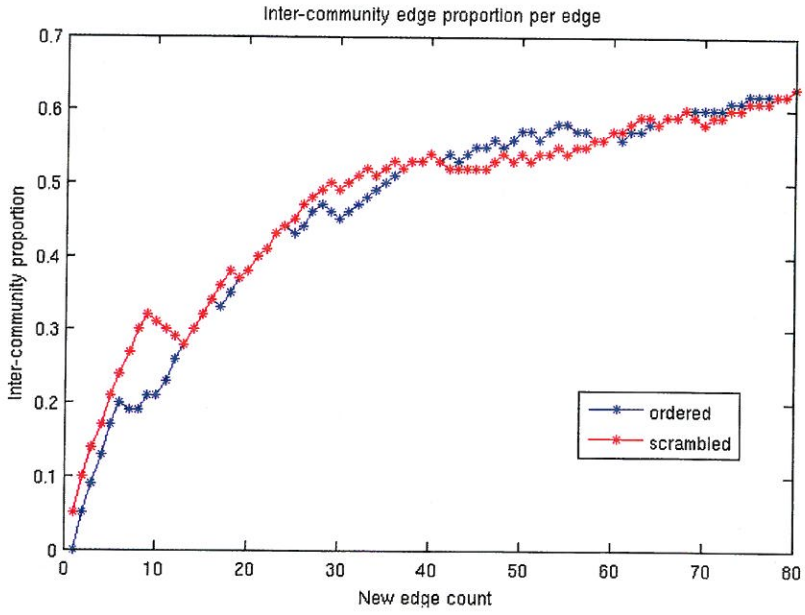
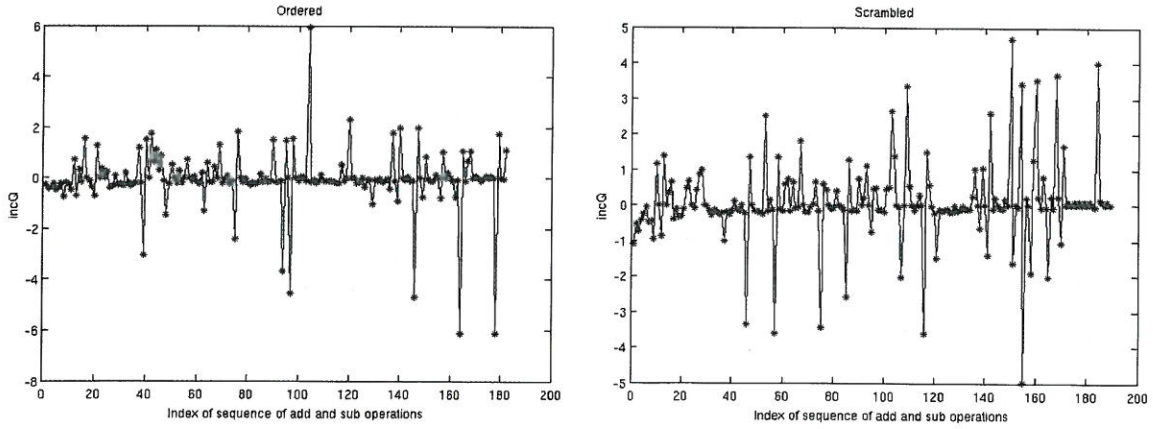


Figure 16: Inter-community edge proportion.



(a) incQ ordered sequence.

(b) incQ scrambled sequence.

Figure 17: Incremental Q values for sequence of add and sub operations.

4.2.3. Large Example

The final example that we will consider is a much larger graph than the previous ones. The initial configuration has 1,000 nodes organized into 16 connected components. The size of the component was generated using a power-law distribution with minimum size 20 and exponent $\alpha = 2.0$. The resulting sizes (number of components of that size) from smallest to largest are:

$$20(3), 21(2), 26(1), 27(1), 32(2), 38(2), 41(1), 47(1), 72(1), 79(1), \text{ and } 466(1).$$

The minimum density of the components was set at 0.05. Hence for a component of size n, approximately 5% of the $\binom{n}{2}$ edges of the complete graph on n nodes are present. However, each component is by definition connected. For a component with 20 nodes, 5% density only gives 1 edge, so additional edges are added to form a binary tree on the nodes. Thus for 20 nodes, the component will have 19 edges. Figure 18 shows a plot of the initial configuration. This graph has 6077 edges for an average degree $< k >= 6$. We can think of the components as communities. However, the McCloskey standardized Louvain algorithm breaks up the largest component into 6 smaller communities resulting in a total of 21 communities. Table 4.2 lists the communities, their containing component, and the initial size of each community as well as other statistics on community size as the graph evolves in time.

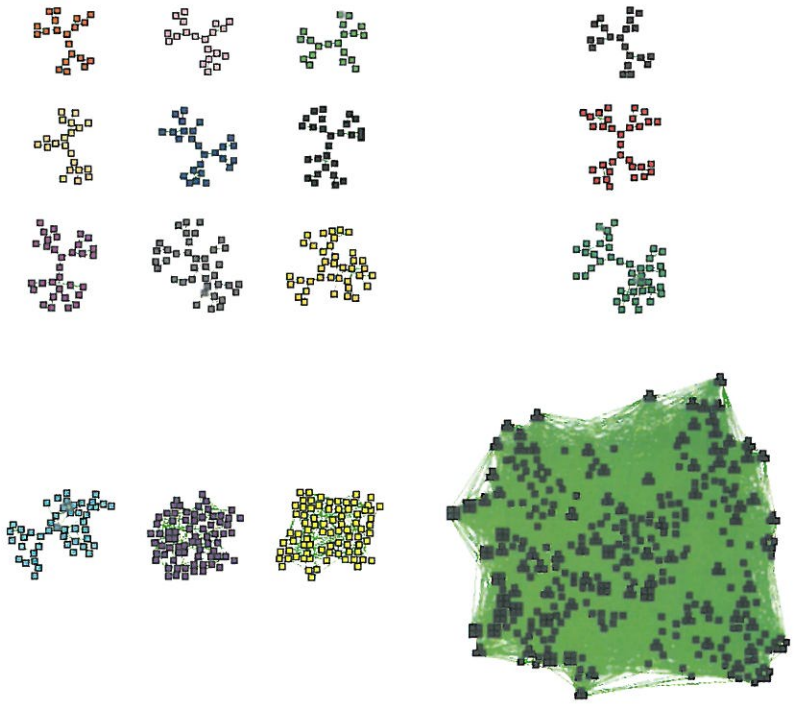


Figure 18: Example 2 initial configuration at 6,077 edges.

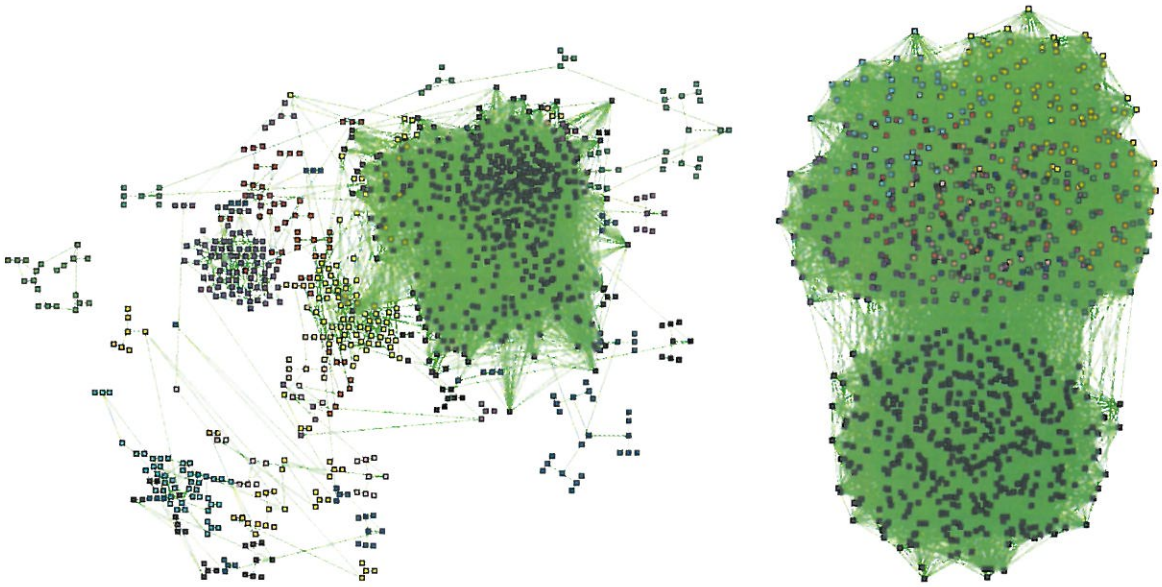
Community	Component	Initial Size	Min Size	Max Size	Final Size
a6	a	26	25	30	26
b58	b	47	42	48	47
c112	c	72	66	72	72
d167	d	41	37	43	41
e198	e	20	20	27	20
f236	f	32	30	35	32
g256	g	20	20	31	20
h278	h	21	21	33	21
i296	i	21	21	30	21
j345	j	79	75	79	79
k398	k	38	33	39	38
l437	l	20	20	26	20
m460	m	32	30	34	32
n484	n	27	27	33	27
o534	o	54	54	441	441
o584	o	58	9	78	9
o598	o	74	5	74	5
o636	o	121	3	121	3
o747	o	72	4	74	4
o908	o	87	4	87	4
p992	p	38	29	41	38

Table 4.2: Community size statistics

We next randomly generate 13,923 edges so the final graph has 20,000 edges for average degree $\langle k \rangle = 20$. The probability of generating a random inter-component edge was set at 0.5. Figure 19 shows two arrangements of the final graph. In Figure 19(b), most of the nodes in the lower ball shape are from community "o." Processing this data took less than 10 minutes with a substantial amount of time required to print out the results.[9]

Table 4.2 indicates some considerable node movement between communities, especially for those belonging to the initial component "o." Notice that the initial and final sizes differ only for those communities belonging to the "o" component. Figure 20 shows the time series of community sizes for the six "o" communities. It appears from the plots that community o534 is absorbing the nodes from the other five "o" communities. Table 4.3 gives confirmation of this since all 466 of the original "o" component nodes lie in the Louvain level 3 partition of the final graph (column 7). However, there are differences between the local and global community assignments.

[9]For example, the output file test_20130726_newrandomedges.txt_standardized_1374849429.out contained 295 megabytes of output.



(a) Snapshot at 12,200 edges.

(b) Snapshot at 20,000 edges.

Figure 19: Two Snapshots of the graph.

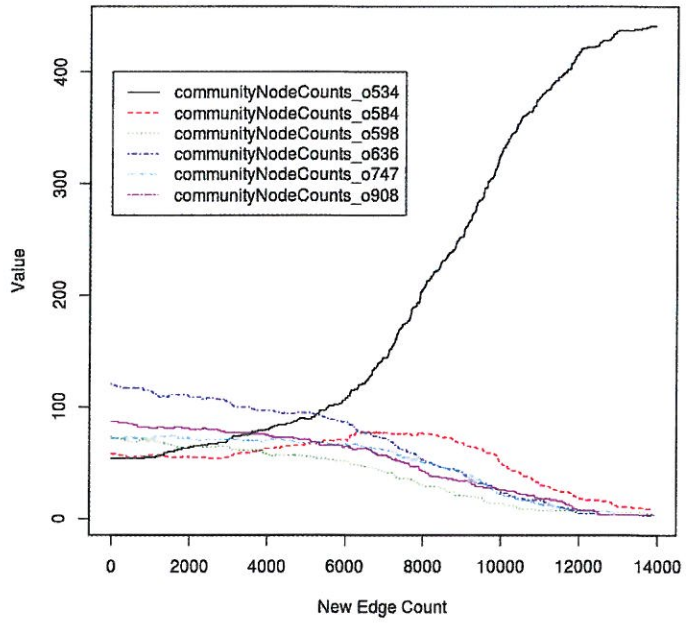
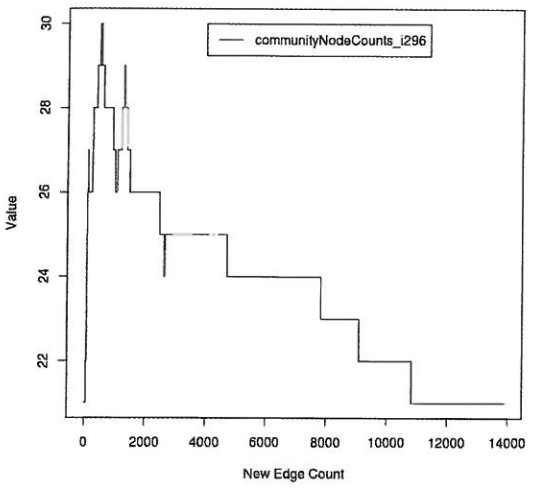
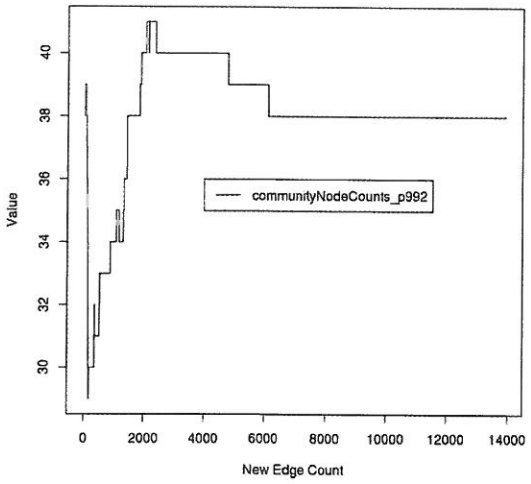


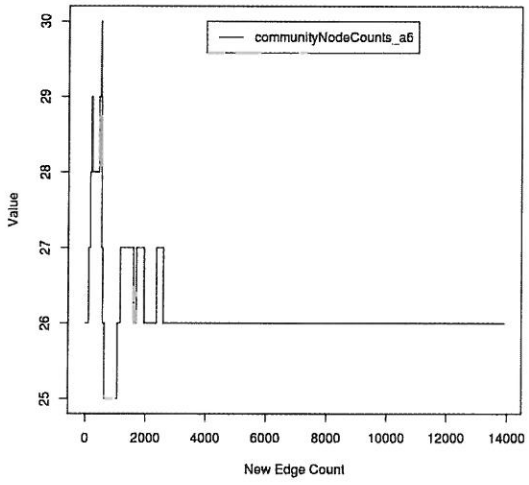
Figure 20: Time series of community "o" sizes.



(a) Community i296.



(b) Community p992



(c) Community a6

Figure 21: More examples of community size time series.

The communities other than "o" exhibit increase and decrease in size before returning to their initial sizes. Community i296 exhibits the largest increase of this type while community p992 exhibits the largest drop as shown in Figure 21. Finally, community a6 exhibits some interesting flux in community size as shown in the last subfigure of Figure 21.

We emphasize that the algorithm outputs local information about communities which doesn't necessarily coincide with a global computation of community membership. If we run a plain Louvain algorithm on the final graph, level 1 has 146 communities, level 2 has 47, and at the top level of the hierarchy (level 3), we obtain 9 communities of sizes 32, 38(2), 41,47,72,79, 187, and 466. Table 4.3 shows that the global Louvain level 3 partition 4 merges the eight of the original components (a,e,g,h,i,l,m,n) into a single community of size 187. The remaining eight components (b,c,d,f,j,k,o,p) stay the same in the final graph. We note that the adjusted Rand index (ARI) value

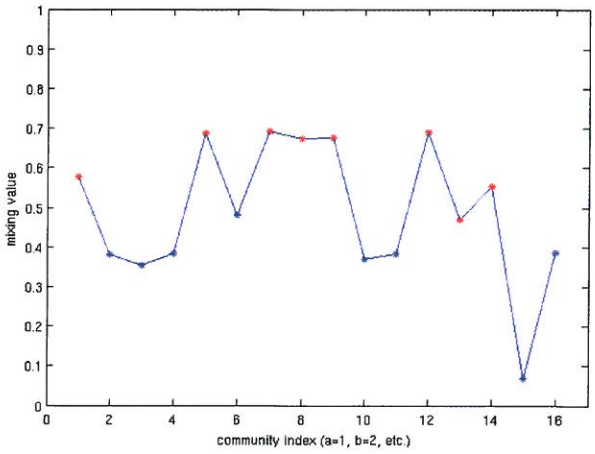


Figure 22: Plot of mixing values

of 0.919933 for the comparison in Table 4.3 is very close to 1.0. This indicates that the initial and final global Louvain partitions are not that far apart. Indeed, 813 of the 1,000 nodes remain in the same communities initially assigned while the remaining eight of the smaller communities were merged into a single global community.

Figure 22 shows a plot of the median mixing value for each community in the starting graph. The mixing value of a node v is the proportion of edges (v, w) incident to a node w from a different community than v out of the total edges involving v.[10] The median mixing value of a community is just the median of mixing value of the nodes belonging to that community. Those communities whose median is above 0.5 (the red points in the figure) were merged into the single community 4 of level 3 with 187 nodes. We note that the original component o was split into eight communities by the dynamic community algorithm, and the final Lovain partitioning does not exactly coincide with the local partitioning (although here it is close).

[10]This is referred to as the *mixing parameter* in [LF09], p. 2 defined as "the ratio between the external degree of a node with respect to its community and the total degree of the node.

	Final Louvain Level 3 Partitions									
Component	1	2	3	4	5	6	7	8	9	Initial size
a	0	0	0	26	0	0	0	0	0	26
b	47	0	0	0	0	0	0	0	0	47
c	0	0	72	0	0	0	0	0	0	72
d	0	41	0	0	0	0	0	0	0	41
e	0	0	0	20	0	0	0	0	0	20
f	0	0	0	0	0	0	0	0	32	32
g	0	0	0	20	0	0	0	0	0	20
h	0	0	0	21	0	0	0	0	0	21
i	0	0	0	21	0	0	0	0	0	21
j	0	0	0	0	79	0	0	0	0	79
k	0	0	0	0	0	38	0	0	0	38
l	0	0	0	20	0	0	0	0	0	20
m	0	0	0	32	0	0	0	0	0	32
n	0	0	0	27	0	0	0	0	0	27
o	0	0	0	0	0	0	466	0	0	466
p	0	0	0	0	0	0	0	38	0	38
Final size	47	41	72	187	79	38	466	38	32	1000

Table 4.3: Table comparing initial and final partitions (ARI=0.919933).

5. Conclusions

The *SCREAM* algorithm presented in this paper is a modularity based streaming algorithm for detecting community structure in a graph which is evolving over time. After careful review, it appears to be the first algorithm of its class, and fills a gap in current community detection research. Its novelty stems from its ability to dynamically detect community change upon observing a new edge.

The *SCREAM* algorithm is a local topological community detection algorithm. As shown in the example of Section 4.2.2, *SCREAM* is not just an application of the (global) Louvain algorithm (or, in fact, any of the other global community detection algorithms) to snapshots of the graph taken after each new edge has arrived. Due to the randomized start agglomeration process of the Louvain algorithm, the potential community structure could radically change with the arrival of each new edge. In effect, once the snapshot is taken, the order of edge arrival becomes irrelevant with this approach. In *SCREAM* the updating explicitly depends on the order of arrival of the edges in the stream, and only those communities within 1-hop of the endpoints of the arriving edge are potentially affected.

The *SCREAM* algorithm is signficantly different from existing community detection algorithms in that it uses the temporal ordering of new edges in addition to the topological structural of the graph to assign communities. This has as a consequence that given the same initial state, *SCREAM* is deterministic in assigning communities to nodes on incoming edges as the graph evolves. Conversely, when the order (arrival time) of incoming edges is permuted, the algorithm can have greatly varying assignments of communities. This shows that the ordering of incoming edges has a significant effect on the clustering assignments produced as shown in Section 4.2.2.

While the effects of new edges are confined to the incident node's local neighborhood (1-hop), the dynamics between communities are non-trivial. A simple example is described in Section 4.1.2, where a new edge results in both of its nodes being assigned to communities distinct from either of the node's previous communities. The significance of this counter-intuitive assignment is that a single edge can affect the dynamics of four communities, not just the two that the edge's nodes are currently assigned. We conjecture that the maximum number of communities that can be changed with the arrival of a new edge is exactly four.

6. Final Comments and Future Work

While the *SCREAM* algorithm accounts for the temporal ordering of new edges in its community assignments, it is limited to making assignments to existing communities. This is not the nature of most time-evolved graphs, communities emerge and disappear, expand and contract, split and merge. Fortunately, our graph generator described in Appendix C is capable of interleaving emerging communities that appear in a burst of edges within its edge stream. A more robust algorithm is being designed to account for the six social group evolutions (growth, merge, contraction, split, emergence, and extinction) outlined in Palla et al. [PBV07].

We have just begun to explore how to make use of the statistical output of the *SCREAM* algorithm. Although the graphical output can be useful in documenting change in community membership and cohesiveness, what is lacking is a time series analysis of this output to determine if

significant changepoints in the community structure can be detected or even predicted.

A concept not explored in *SCREAM* is the notion of community roles. The classification of nodes into one of the four node types as defined in Scripps, Tan and Esfahanian [STE07] is of interest when attempting to understand the dynamics between large number of communities. With the addition of the temporal dimension in *SCREAM*, the community role a node plays in a graph could be an intriguing source of information when analyzing community dynamics over time. The dynamics of community roles could be an indication of a larger community event, such as a merging of communities. This will be addressed in the description of *SCREAM 2* in a second paper.

Thus far, the discussion of an evolving graph has only been in the context of growth of communities. However, significant work has been done on the decay of graphs by Rosvall and Bergstrom [RB08] and Palla, Barabasi and Vicsek [PBV07]. Incorporating the ability to age off edges in our graph generator will have implications for community membership, and is an interesting problem not yet explored by the graph community. Similarly, re-occurring motifs and patterns in the graph (i.e. frequently observed sequences of edges) present problems not explored by *SCREAM* thus far. These problems will be addressed in the second iteration of the algorithm, *SCREAM 2*.

7. Acknowledgements

The authors would like to thank Joseph P. McCloskey for helpful comments during the course of this research. Any remaining errors are due solely to the authors. Finally, we thank James Fairbanks of the Georgia Institute of Technology for suggesting the experiment in Section 4.2.2.

Appendix

Here we collect auxiliary information about modularity, partition comparison and graph generation used in this paper.

A. Change in Modularity after Moving a Node from One Community into Another

In this section we compute using the setup in Section 2.1 the change in modularity, ΔQ, when moving a node i from community V_r into community V_s. We do this in two steps. First, remove i from V_r and create the singleton community $\{i\}$. Second, move isolated node i into community V_s. Let us imagine the "state" of the community structure at time t to be $V_1, \ldots, V_r, \ldots, V_s, \ldots, V_K$. At time $t+1$ the state is $V_1, \ldots, V_r - \{i\}, \{i\}, \ldots, V_s, \ldots, V_K$, and at $t+2$ the state is $V_1, \ldots, V_r - \{i\}, \ldots, V_s \cup \{i\}, \ldots, V_K$. Denote by Q^t the modularity at time t. Then $\Delta Q \equiv \Delta Q^{t,t+2} = (Q^{t+1} - Q^t) + (Q^{t+2} - Q^{t+1}) = \Delta Q^{t,t+1} + \Delta Q^{t+1,t+2}$. From Section 2.1 equation (2.7), the second difference is

$$\Delta Q^{t+1,t+2} = \frac{l_{i,V_s}}{2m} - \frac{2d_{V_s}\, d_i}{4m^2}. \tag{A.1}$$

To compute the first difference, recall that $l_r = |E_{V_r}|$ and set $\hat{l}_r = |E_{V_r - \{i\}}|$, the number of edges incident to nodes in $V_r - \{i\}$, so $l_r = \hat{l}_r + l_{i,V_r} + A_{ii}$ and $\sum_{j,k \in V_r} A_{jk} = 2l_r = 2(\hat{l}_r + l_{i,V_r})$.

Similarly, let $\hat{d}_{V_r} = d_{V_r} - d_i$. We have from (2.3)

$$Q_{V_r} = \frac{2l_r}{2m} - \left(\frac{d_{V_r}}{2m}\right)^2 \tag{A.2}$$

$$= \frac{2\hat{l}_r}{2m} + \frac{l_{i,V_r} + A_{ii}}{2m} - \left(\frac{\hat{d}_{V_r}}{2m} + \frac{d_i}{2m}\right)^2 \tag{A.3}$$

$$= \left[\frac{2\hat{l}_r}{2m} - \left(\frac{\hat{d}_{V_r}}{2m}\right)^2\right] + \left[\frac{A_{ii}}{2m} - \left(\frac{d_i}{2m}\right)^2\right] + \frac{l_{i,V_r}}{2m} - \frac{2\hat{d}_{V_r}\, d_i}{4m^2} \tag{A.4}$$

$$= Q_{V_r-\{i\}} + Q_{\{i\}} + \frac{l_{i,V_r}}{2m} - \frac{2\hat{d}_{V_r}\, d_i}{4m^2}, \tag{A.5}$$

where $Q_{V_r-\{i\}}$ and $Q_{\{i\}}$ denote the modularity of the communities $V_r - \{i\}$ and $\{i\}$, respectively. Therefore, the first difference

$$\Delta Q^{t,t+1} = Q_{V_r-\{i\}} + Q_{\{i\}} - Q_{V_r} = -\frac{l_{i,V_r}}{2m} + \frac{2\hat{d}_{V_r}\, d_i}{4m^2}, \tag{A.6}$$

and hence the change in modularity, $\Delta Q = \Delta Q^{t,t+2} = \Delta Q^{t,t+1} + \Delta Q^{t+1,t+2}$, resulting from removing node i from V_r and moving it to V_s using (A.1) and (A.6) is

$$\Delta Q = \frac{l_{i,V_s} - l_{i,V_r}}{2m} - \frac{\left(d_{V_s} - \hat{d}_{V_r}\right) d_i}{2m^2} \tag{A.7}$$

$$= \frac{1}{4m^2}\left[2m\left(l_{i,V_s} - l_{i,V_r}\right) - \left(d_{V_s} - \hat{d}_{V_r}\right) 2d_i\right]. \tag{A.8}$$

B. Comparing Partitions with the Adjusted Rand Index

To compare two partitionings of the nodes, we make use of the adjusted Rand index first introduced by Hubert & Arabie [HA85]. Let $X = \{X_1, \ldots, X_r\}$ and $Y = \{Y_1, \ldots, Y_s\}$ be two partitions of a set S with n elements, and set $n_{ij} = |X_i \cap Y_j|$, $n_{i\cdot} = \sum_j n_{ij}$ and $n_{\cdot j} = \sum_i n_{ij}$. The adjusted Rand index (ARI) is defined by

$$\mathrm{ARI}(X, Y) = \frac{\sum_{i,j}\binom{n_{ij}}{2} - \left[\sum_i \binom{n_{i\cdot}}{2}\sum_j \binom{n_{\cdot j}}{2}\right]}{\frac{1}{2}\left[\sum_i \binom{n_{i\cdot}}{2} + \sum_j \binom{n_{\cdot j}}{2}\right] - \left[\sum_i \binom{n_{i\cdot}}{2} + \sum_j \binom{n_{\cdot j}}{2}\right]/\binom{n}{2}}. \tag{B.1}$$

The ARI has the form

$$\text{Adjusted Index} = \frac{\text{Index} - \text{ExpectedIndex}}{\text{MaxIndex} - \text{ExpectedIndex}}, \tag{B.2}$$

where Index refers to the Rand Index. The ARI attains a maximum value of 1 when there is perfect agreement between the two partitions, i.e. $r = s$ and $Y_i = X_{\pi(i)}$ where π is a permutation of the indices $1, 2, \ldots, r$. An ARI score of 0 occurs when the Rand Index equals its expected

value under the null hypothesis that the $r \times s$ contigency table is constructed from the generalized hypergeometric distribution, i.e. the X and Y partitions are picked at random, subject to having the original number of classes and objects in each. It is possible to have negative values of the ARI. However, Hubert and Arabie state "It might be appropriate to have a well-defined lower bound as well for the index in (4) (Equation B.2 here), but since negative values of the index have no substantive use, the required normalization would offer no practical benefits."[11] For more information, consult the original paper by Hubert & Arabie [HA85].

C. Graph Generation

Our dynamic graph generator is written in Objected-Oriented Perl (see [Sri97]. Here we describe only those aspects of the generator that were used for the examples in this paper. In a later paper we may give full details of the functionality and options available with this graph generator. The purpose of our generator is to produce a dynamically changing graph which creates communities and significantly changes membership in these communities over time.

In [LF09] the Lancichenetti, Fortunato and Radicchi (LFR) benchmark for graph generation is summarized as a special case of the planted l-partition model where communities have varying sizes and nodes have different degrees. Details on this popular benchmark are given in [LFR08]. Unlike LFR, our generator produces a graph that has an initial well-defined community structure with prescribed density of edges, and the edges of the initial community structures appear in sequence so that separate communities can "emerge" over time. As the graph evolves, inter-community edges are introduced with probability p, a parameter that can be set by the user. As a consequence, the degree distribution evolves over time and at various stages can be power-law, normally distributed, or even uniform. This is in constrast to LFR which allows the user to prescribe a degree sequence for the nodes apriori by drawing from a power-law distribution. Our code generates an edge stream which in addition to containing new edges between existing edges, can also contain new nodes, and this can affect the degree distribution as well. However, we have borrowed the idea from the LFR benchmark of choosing the community sizes from a power-law distribution which in turn LFR attributes to work of Clauset, Newman and Moore [CNM04].

C.1. Stage 1

The graph is generated in two stages. The first builds a graph with a user-defined number of nodes, N, and either a user-defined or randomly generated number of connected components. The sizes of these components as measured by the number of nodes is generated using a power law model subject to the constraint that the sum of the nodes in the components equals N. The user supplies the power-law exponent. Each component is constructed by starting with a "skeleton" subgraph on the nodes in the component. The skeleton can be either a binary tree or a 2-regular subgraph (i.e. a "circle") on the nodes. For example, a component with 5 nodes would have 4 edges in the binary tree skeleton, and 5 edges in the circular skeleton. Additional random edges may then be added to the skeleton to obtain the desired number of edges in the component.

[11] Hubert & Arabie,[HA85], p. 198.

The number of edges in a component is controlled by its density. If a component contains n nodes then the density, d, of this component is defined to be the ratio of the number of edges in the component divided by the number of edges in K_n, the complete graph on n vertices. Thus the number of edges is $d\binom{n}{d}$.

The user specifies the density of the components. Currently, the density is uniform across all components. If the density is too small for the number of nodes in the set, then either a binary tree or circular skeleton is constructed on those nodes. If one wanted to generate a graph with three connected components consisting of binary trees, then choosing the binary tree option for the skeleton together with a density of zero after setting the number of components to 3 would create the desired graph. On the otherhand, if the graph is to consist of three connected components each of which is a complete graph, one inputs the number of nodes, the number of components, and sets the density equal to 1.0. The initial graph in Section 4.2.1 was generated using three components, of sizes 3,4, and 5 with density 0.66 using a binary tree skeleton.

Finally, each edge in the graph has a timestamp. The timestamp for the first edge is generated from the current date and time the program was invoked using the Perl *timelocal* function which outputs Unix epoch seconds format. Each succeeding edge timestamp is obtained from the previous one by adding one to the first edge's timestamp. The graph generator writes the initial graph edge stream to a file. In addition, a listing of the nodes and their initial community assignment is written to a file.

C.2. Stage 2

The second stage of the graph generator produces additional random edges for the graph created in stage 1. The user inputs the probability, p of an inter-community edge and the number of additional edges to generate. For the next edge to be generated, a random number, u, between 0 and 1 is drawn. If $p \geq u$ then an edge is generated connected two randomly chosen nodes from two randomly chosen distinct communities. Otherwise, a random community is chosen and within that community, two random nodes are chosen to connect with an (intra-community) edge. The graph generator keeps track of the edges it has generated and in the current version of the program, only unique edges are generated. That is, once an edge has been generated, it will not be generated in the future. With this in mind, the graph generator also keeps track of how many edges each component currently has, and if this number equals the number of edges in the complete graph on the number of the component's nodes, then this component is removed from consideration when generating an intra-community edge. Future versions of the graph generator will have this constraint relaxed.

The graph generator writes the new edge stream to a file. Thus the output of the current graph generator consists of three files: a node and community assignment file, an initial graph edge stream file, and a new edge stream file containing additional edges ordered by timestamp.

C.3. Generating Community Sizes

The distribution of community sizes can be approximated by a power-law distribution with exponents between 1 and 3 (Fortunato, [For10], p. 83). The LFR graph generator [LFR08] employs

power-law distributions for both community size and degree distribution. Our graph generator determines community size according to a discrete power-law distribution described in Corral et al. [CDiC12]. The simulator of Corral et al. makes use of a generalization of the accept/reject method. The discrete power-law distribution with exponent β and minimum value $a \in \mathbb{N}$ is defined by

$$f(n) = \Pr[X = n] = \frac{1}{\zeta(\beta+1, a)\, n^{\beta+1}}, \text{where } n \in \mathbb{N},\ n \geq a, \beta > 0, \text{ and} \tag{C.1}$$

$$\zeta(\gamma, a) = \sum_{k=0}^{\infty} \frac{1}{(a+k)^{\gamma}} \tag{C.2}$$

is the Hurwitz zeta function.

The generation of random values according to this distribution is given by the following method from [CDiC12].

- Generate a uniform random number $0 \leq u \leq u_{max}$ with $a = 1/u_{max}^{1/\beta}$.
- Form a new random number
$$y = \lfloor (1/u^{1/\beta}) \rfloor.$$
The probability function of y according to [CDiC12] is given by
$$q(y) = (a/y)^{\beta} - (a/(y+1))^{\beta}.$$
- Accept y if a new uniform random number $0 \leq v \leq 1$ satisfies
$$v \leq \frac{f(y)q(a)}{f(a)q(y)}$$
and reject otherwise. If accepted, take $n = y$. As remarked in [CDiC12], we note that computation of the zeta function ζ is avoided here. If we define $\tau = (1 + y^{-1})^{\beta}$ and $b = (a+1)^{\beta}$ then after some algebraic manipulation, the acceptance condition becomes the much easier to verify
$$vy\frac{\tau - 1}{b - a^{\beta}} \leq \frac{a\tau}{b}.$$

References

[BGLL08] V. Blondel, J. Guillaume, R. Lambiotte, and E. Lefebvre, *Fast unfolding of communities in large networks*, Journal of Statistical Mechanics: Theory and Experiment 2008 (10) P10008 (2008).

[BM10] D. Bader and J.P. McCloskey, *Modularity and Graph Algorithms*, SIAM Annual Meeting (2010).

[CDiC12] A. Corral, A. Delucca, and R.Ferre i Cancho, *A practical recipe to fit discrete power-law distributions*, arXiv:1209.128-v1[stat.AP] (2012).

[CNM04] A. Clauset, M.E.J. Newman, and C. Moore, *Finding local community structure in networks*, Phys. Rev. E. **70** 066111 (2004).

[For10] S. Fortunato, *Community detection in graphs*, Physics Reports 486 75-174 (2010).

[HA85] L. Hubert and P. Arabie, *Comparing Partitions*, Journal of Classification **2**: 193-218 (1985).

[IBM12] IBM, *IBM Streams Processing Language Specification*, 2012.

[JCLZ10] N. Jiang, J. Cao, Y. Jin L.E. Li, and Z-L. Zhang, *Identifying Suspicious Activities through DNS Failure Graph Analysis*, IEEE 1,20 (2010).

[LF09] A. Lancichinetti and S. Fortunato, *Community detection algorithms: a comparative analysis*, Physical Review E**80** 056117 (2009).

[LFR08] A. Lancichinetti, S. Fortunato, and F. Radicchi, *Benchmark graphs for testing community detection algorithms*, Physical Review E **78** 046110 (2008).

[MFFP11] P. De Meo, E. Ferrara, G. Fiumara, and A. Provetti, *Generalized Louvain method for community detection in large networks*, ISDA'11: Proceedings of the 11th International Conference on Intelligent Systems Design and Applications (2011).

[NDXT11] N.P. Nguyen, T.N. Dinh, Y. Xuan, and M.T. Thai, *Adaptive Algorithms for Detecting Community Structure in Dynamic Social Networks*, *www.cise.ufl.edu/yxuan/papers* (2011).

[NG04] M.E.J. Newman and M. Givan, *Finding and evaluating community structure in networks*, Phys. Rev. E. **69**(2),026113 (2004).

[PBV07] G. Palla, A.-L. Barabási, and T. Vicsek, *Quantifying social group evolution*, Nature **446**, 664 (2007).

[PKVS12] S. Papadopoulos, Y. Kompartsiaris, A. Vakali, and P. Spyridonos, *Community Detection in Social Media*, Data Min Knowl Disc 2012, 24: 515-554 (2012).

[RB08] M. Rosvall and C.T. Bergstrom, *Maps of random walks on complex networks reveal community structure*, PNAS, vol. 105, no. 4 (2008).

[Sri97] S. Srinivasan, *Advanced Perl Programming*, O'Reilly & Associates, Inc., 1997.

[STE07] J. Scripps, P-N. Tan, and A-H. Esfahanian, *Node Roles and Community Structure in Networks*, Joint 9th WEBKDD and 1st SNA-KDD Workshop '07 (2007).