GraphScope: Parameter-free Mining of Large Time-evolving Graphs

Jimeng Sun[‡]
Philip S. Yu[§]

Spiros Papadimitriou[§]
Christos Faloutsos[‡]

[‡] Carnegie Mellon University
{jimeng,christos}@cs.cmu.edu

[§] IBM TJ Watson lab
{spapadim,psyu}@us.ibm.com

ABSTRACT

How can we find communities in dynamic networks of social interactions, such as who calls whom, who emails whom, or who sells to whom? How can we spot discontinuity time-points in such streams of graphs, in an on-line, any-time fashion? We propose *GraphScope*, that addresses both problems, using information theoretic principles. Contrary to the majority of earlier methods, it needs *no* user-defined parameters. Moreover, it is designed to operate on large graphs, in a streaming fashion. We demonstrate the efficiency and effectiveness of our *GraphScope* on real datasets from several diverse domains. In all cases it produces meaningful time-evolving patterns that agree with human intuition.

Categories and Subject Descriptors

H.2.8 [**Database applications**]: Data mining

General Terms

Algorithms

1. INTRODUCTION

Graphs arise naturally in a wide range of disciplines and application domains, since they capture the general notion of an association between two entities. However, the aspect of time has only recently begun to receive some attention [15, 20]. Some examples of the time-evolving graphs include: (a) Network traffic events indicate ongoing communication between source and destination hosts, similar to the `NETWORK` dataset in our experiments; (b) Email networks associate a sender and a recipient at a given date, like the `ENRON` data set [2] in the experiments; (c) Call detail records in telecommunications networks associate a caller with a callee. The set of all conversation pairs over each week forms a graph that evolves over time, like the publicly available '`CELLPHONE`' dataset of MIT users calling each other [1]; (d) Transaction data: in a financial institution, who accessed what account, and when; (e) In a database compliance setting [4], again we need to record which user accessed what data item and when.

To complicate matters further, large amounts of data such as those in the above examples are continuously collected and patterns are also changing over time. Therefore, batch methods for pattern discovery are not sufficient. We need tools that can incrementally find the communities and monitor the changes. In summary, there are two key problems that need to be addressed:

P1) *Community discovery:* Which groups or communities of nodes are associated with each other?

P2) *Change detection:* When does the community structure change and how to quantify the change?

Moreover, we want to answer these questions (a) *without* requiring any user-defined parameters, and (b) in a *streaming* fashion.

For example, we want to answer questions such as: How do the network hosts interact with each other? What kind of host groups are there, e.g., inactive/active hosts; servers; scanners? Who emails whom? Do the email communities in a organization such as `ENRON` remain stable, or do they change between workdays (e.g., business-related) and weekends (e.g., friend and relatives), or during major events (e.g.,the FBI investigation and CEO resignation)?

We propose GraphScope, which addresses both of the above problems simultaneously. More specifically, GraphScope is an efficient, adaptive mining scheme on time-evolving graphs. Unlike many existing techniques, it requires no user-defined parameters, and it operates completely automatically, based on the Minimum Description Length (MDL) principle. Furthermore, it adapts to the dynamic environment by automatically finding the communities and determining good *change-points* in time.

In this paper we consider bipartite graphs, which treat source and destination nodes separately (see example in Figure 2). As will become clear later, we discover separate source and destination partitions, which are desirable in several application domains. Nonetheless, our methods can be easily modified to deal with unipartite graphs, by constraining the source-partitions to be the same as the destination-partitions [6].

The main insight of dealing with such graphs is to group "similar" sources together into *source-groups* (or *row-groups*), and also "similar" destinations together, into *destination-groups* (or *column-groups*). Examples in Section 6.2 show how much more orderly (and easier to compress) the adjacency matrix of a graph is, after we strategically re-order its rows and columns. The exact definition of "similar" is actually simple, and rigorous: the most similar source-partitions for a given source node is the one that leads to small encoding cost (see Section 4 for more details).

Furthermore, if these communities (source and destination partitions) do not change much over time, consecutive snapshots of the evolving graphs have similar descriptions and can also be grouped together into a time segment, to achieve better compression. Whenever a new graph snapshot cannot fit well into the old segment (in terms of compression), GraphScope introduces a change-point, and starts a new segment at that time-stamp. Those change points often detect drastic discontinuities in time. For example on the ENRON dataset, the change points all coincide with important events related to the ENRON company, as shown in Figure 1 (more details in Section 6.2).

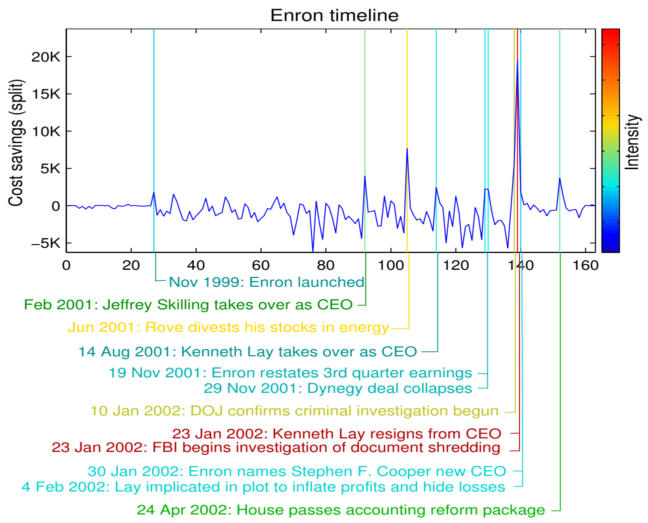


Figure 1: ENRON dataset (Best viewed in color). Relative compression cost versus time. Large cost indicates change points, which coincide with the key events. E.g., at time-tick 140 (Feb 2002), CEO Ken Lay was implicated in fraud.

Contributions. Our proposed approach, GraphScope, monitors communities and their changes in a stream of graphs efficiently. It has the following key properties:

- *Parameter-free:* GraphScope is completely automatic, requiring no parameters from the user (like number of communities, thresholds to assess community drifts, and so on). Instead, it is based on sound information-theoretic principles, specifically, MDL.
- *Adaptive:* It can effectively track communities over time, discovering both communities as well as change-points in time, that agree with human intuition.
- *Streaming:* It is fast, incremental and scalable for the streaming environment.

We demonstrate the efficiency and effectiveness of our approach in discovering and tracking communities in real graphs from several domains.

The rest of the paper is organized as follows: Section 2 reviews the related work. Section 3 introduces some necessary definitions and formalizes the problem. Section 4 presents the objective function. Section 5 presents our proposed method to search for an optimal solution, Section 6 shows the experimental evaluation and Section 7 concludes.

2. RELATED WORK

Here we discuss related work from three areas: mining static graphs, mining dynamic graphs, and stream mining.

2.1 Static Graphs

Graph mining has been a very active area in the data mining community. From the exploratory aspect, Faloutsos et al. [10] have shown the power-law distribution on the Internet graph. Kumar et al. [14] discovered the bow-tie model for web graphs.

From the algorithmic aspect, graph partitioning has attracted much interest, with prevailing methods being METIS [12] and spectral partitioning [16]. Even in these top-performing methods, users must specify the number of partitions k. Moreover, they typically also require a measure of imbalance between the two pieces of each cut.

Information-theoretic Co-clustering [9] simultaneously reorders the rows and columns of a normalized contingency table or a two-dimensional probability distribution, where the number of clusters has to be specified. The Cross-association method [7] formulates the co-clustering problem as a binary matrix compression problem. Noble and Cook [17] propose an entropy-based anomaly detection scheme for graphs.

All these methods deal with static matrices or graphs, while GraphScope is designed to work with dynamic streams. Moreover, most of methods except for cross-association require some user-defined parameters, which may be difficult to set and which may dramatically affect the final result, as observed in [13]. Keogh et al. [13] proposed the notion of parameter free data mining. GraphScope shares the same spirit but focuses on different problems.

In addition to graph mining, several storage schemes [11, 18] have been proposed to compress large binary matrices (graphs) by column reordering. However, none of those scheme perform both column and row reordering and their focus is on compression rather than mining.

2.2 Dynamic Graphs

From the exploratory viewpoint, Leskovec et al. [15] discovered the shrinking diameter phenomena on time-evolving graphs. Backstrom et al. [5] study community evolution in social networks.

From the algorithmic aspect, Sun et al. [20] present dynamic tensor analysis, which incrementally summarizes tensor streams (high-order graph streams) as smaller core tensor streams and projection matrices. This method still requires user-defined parameters (like the size of the core tensor). Moreover, it gives lossy compression. Aggarwal and Yu [3] propose a method to selectively store a subset of graphs to approximate the entire graph stream and to find community changes in time-evolving graphs based on the user specified time interval and the number of communities. Again, our GraphScope framework avoids all these user-defined parameters.

3. PROBLEM DEFINITION

In this section, we formally introduce neccessary notation and formulate the problems.

3.1 Notation and definition

Calligraphic letters always denote *graph streams* or *graph stream segments* (consisting of one or more graph snapshots),

Sym.	Definition
$\mathcal{G}$, $\mathcal{G}^{(s)}$	Graph stream, Graph segment
t	Timestamp, $t \geq 1$.
m, n	Number of source(destination) nodes.
$G^{(t)}$	Graph at time t ($m \times n$ adjacency matrix).
i, j	Node indices, $1 \leq i \leq m$, $1 \leq j \leq n$.
$G^{(t)}_{i,j}$	Indicator for edge (i, j) at time t.
s	Graph segment index, $s \geq 1$.
t_s	Starting time of s-th segment.
k_s,ℓ_s	Number of source (dest.) partitions for segment s.
p, q	Partition indices, $1 \leq p \leq k_s$, $1 \leq q \leq \ell_s$.
$I^{(s)}_p$	Set of sources belonging to the p-th partition, during the s-th segment.
$J^{(s)}_q$	Similar to $I^{(s)}_p$, but for destination nodes.
$m^{(s)}_p$	Source partition size, $m^{(s)}_p \equiv \|I^{(s)}_p\|$, $1 \leq p \leq k_s$.
$n^{(s)}_p$	Dest. partition size, $n^{(s)}_p \equiv \|J^{(s)}_p\|$, $1 \leq p \leq \ell_s$.
$\mathcal{G}^{(s)}_{p,q}$	Subgraphs induced by p-th and q-th partitions of segment s, i.e., subgraph segment
$\|\mathcal{G}^{(s)}_{p,q}\|$	Size of subgraphs segment, $\|\mathcal{G}^{(s)}_{p,q}\| := m^{(s)}_p n^{(s)}_q (t_{s+1} - t_s)$.
$\|E\|^{(s)}_{p,q}$	Number of edges in $\mathcal{G}^{(s)}_{p,q}$
$\rho^{(s)}_{p,q}$	Density of $\mathcal{G}^{(s)}_{p,q}$, $\frac{\|E\|^{(s)}_{p,q}}{\|\mathcal{G}^{(s)}_{p,q}\|}$
$H(.)$	Shannon entropy function

Table 1: Definitions of symbols

while individual graph snapshots are denoted by non-calligraphic, upper-case letters. Superscripts in parentheses denote either timestamps t or graph segment indices s, accordingly. Similarly, subscripts denote either individual nodes i, j or node partitions p, q.

Definition 3.1 (Graph stream). *A* graph stream $\mathcal{G}$ *is a sequence of graphs $G^{(t)}$, i.e.,*

$$\mathcal{G} := \{G^{(1)}, G^{(2)}, \ldots, G^{(t)}, \ldots\},$$

which grows indefinitely over time. Each of these graphs links m source nodes to n destination nodes.

For example in Figure 2, the first row shows the first three graphs in a graph stream, where $m = 4$ and $n = 3$. Furthermore, the graphs are represented as sparse matrices in the bottom of Figure 2 (a black entry is 1, which indicates an edge between the corresponding nodes; likewise a white entry is 0).

In general, each graph may be viewed as an $m \times n$ binary adjacency matrix, where rows $1 \leq i \leq m$ correspond to source nodes and columns $1 \leq j \leq n$ correspond to destination nodes. We use sparse representation of the matrix (i.e., only non-zero entries are stored) whose space consumption is similar to adjacency list representation. Without loss of generality, we assume m and n are the same for all graphs in the stream; if not, we can introduce all-zero rows or columns in the adjacency matrices.

One of our goals is to track how the structure of the graphs $G^{(t)}$, $t \geq 1$, evolves over time. To that end, we will group consecutive timestamps into segments.

Definition 3.2 (Graph stream segment). *The set of graphs between timestamps t_s and $t_{s+1}-1$ (inclusive) consist the s-th segment $\mathcal{G}^{(s)}$, $s \geq 1$, which has length $t_{s+1} - t_s$,*

$$\mathcal{G}^{(s)} := \{G^{(t_s)}, G^{(t_s+1)}, \ldots, G^{(t_{s+1}-1)}\}.$$

Intuitively, a "*graph stream segment*" (or just "*graph segment*") is a set of consecutive graphs in a graph stream. For example in Figure 2, $\mathcal{G}^{(1)}$ is a graph segment consisting of two graph $G^{(1)}$ and $G^{(2)}$.

Next, within each segment, we will partition the source and destination nodes into source partitions and destination partitions, respectively.

Definition 3.3 (Graph segment partitions). *For each segment $s \geq 1$, we partition source nodes into k_s source partitions and destination nodes into ℓ_s destination partitions. The set of source nodes that are assigned into the p-th source partition $1 \leq p \leq k_s$ is denoted by $I^{(s)}_p$. Similarly, the set of destination nodes assigned to the q-th destination partition is denoted by $J^{(s)}_q$, for $1 \leq q \leq \ell_s$.*

The sets $I^{(s)}_p$ $(1 \leq p \leq k_s)$ partition the source nodes, in the sense that $I^{(s)}_p \cap I^{(s)}_{p'} = \emptyset$ for $p \neq p'$, while $\bigcup_p I^{(s)}_p = \{1, \ldots, m\}$. Similarly, the sets $J^{(s)}_q$ $(1 \leq q \leq l_s)$ partition the destination nodes. For example in Figure 2, the first graph segment $\mathcal{G}^{(1)}$ has source partitions $I^{(1)}_1 = \{1, 2\}$, $I^{(1)}_2 = \{3, 4\}$, and destination partitions $J^{(1)}_1 = \{1\}$, $J^{(1)}_2 = \{2, 3\}$ where $k_1 = 2$, $\ell_1 = 2$. We can similarly define source and destination partition for the second graph segment $\mathcal{G}^{(2)}$, where $k_2 = 3$, $\ell_2 = 3$.

3.2 Problem formulation

In this paper, the ultimate goals are to find communities on time-evolving graphs along with the *change-points*, if any. Thus, the following two problems need to be addressed.

Problem 1 (Partition Identification). *Given a graph stream segment $\mathcal{G}^{(s)}$, how to find good partitions of source and destination nodes, which summarize the fundamental community structure.*

The meaning of "good" will be made precise in the next section, which formulates our cost objective function. However, to obtain an answer for the above problem, two important sub-questions need to be answered (see Section 5.1):

- How to assign the m source and n destination nodes into k_s source and ℓ_s destination partitions?
- How to determine k_s and ℓ_s?

Problem 2 (Time Segmentation). *Given a graph stream $\mathcal{G}$, how can we incrementally construct graph segments, by selecting good change points t_s.*

Section 5.2 presents the algorithms and formalizes the notion of "good" for both problems above. We name the whole analytic process *GraphScope.*

4. GRAPHSCOPE ENCODING OBJECTIVE

Our mining framework is based on one form of the Minimum Description Length (MDL) principle and employs a lossless encoding scheme for a graph stream. Our objective function estimates the number of bits needed to encode the full graph stream so far. Our proposed encoding scheme takes into account both the community structures, as well as their change points in time, in order to achieve a concise description of the data. The fundamental trade-off that decides the "best" answers to problems 1 and 2 in Section 3.2

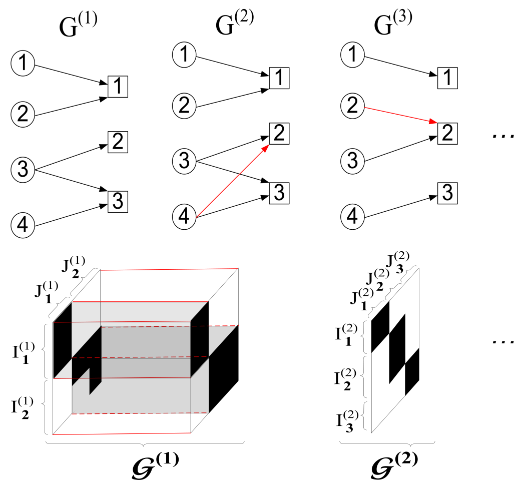


Figure 2: Notation illustration: A graph stream with 3 graphs in 2 segments. First graph segment consisting of $G^{(1)}$ and $G^{(2)}$ has two source partitions $I_1^{(1)} = \{1,2\}$, $I_2^{(1)} = \{3,4\}$; two destination partitions $J_1^{(1)} = \{1\}$, $J_2^{(1)} = \{2,3\}$. Second graph segment consisting of $G^{(3)}$ has three source partitions $I_1^{(2)} = \{1\}$, $I_2^{(2)} = \{2,3\}$, $I_3^{(2)} = \{4\}$; three destination partitions $J_1^{(2)} = \{1\}$, $J_2^{(2)} = \{2\}$, $J_2^{(2)} = \{3\}$.

is between (i) the number of bits needed to describe the communities (or, partitions) and their change points (or, segments) and (ii) the number of bits needed to describe the individual edges in the stream, given this information.

We begin by first assuming that the change-points as well the source and destination partitions for each graph segment are given, and we show how to estimate the bit cost to describe the individual edges (part (ii) above). Next, we show how to incorporate the partitions and segments into an encoding of the entire stream (part (i) above).

4.1 Graph encoding

In this paper, a graph is presented as a m-by-n binary matrix. For example in Figure 2, $G^{(1)}$ is represented as

$$G^{(1)} = \begin{pmatrix} 1 & 0 & 0 \\ 1 & 0 & 0 \\ 0 & 1 & 1 \\ 0 & 0 & 1 \end{pmatrix} \tag{1}$$

Conceptually, we can store a given binary matrix as a binary string with length mn, along with the two integers m and n. For example, equation 1 can be stored as 1100 0010 0011 (in column major order), along with two integers 4 and 3.

To further save space, we can adopt some standard lossless compression scheme (such as Huffman coding, or arithmetic coding [8]) to encode the binary string, which formally can be viewed as a sequence of realizations of a binomial random variable X. The code length for that is accurately estimated as $mnH(X)$ where $H(X)$ is the entropy of variable X. For notational convenience, we also write that as $mnH(G^{(t)})$. Additionally, three integers need to be stored: the matrix sizes m and n, and the number of ones in the matrix (i.e., the number of edges in the graph) denoted as $|E|$ [1]. The cost for storing three integers is $\log^\star|E| + \log^\star m + \log^\star n$ bits, where $\log^\star$is the universal code length for an integer[2]. Notice that this scheme can be extended to a sequence of graphs in a segment.

More generally, if the random variable X can take values from the set M, with size $|M|$ (a multinomial distribution), the entropy of X is

$$H(X) = -\sum_{x\in M} p(x)\log p(x).$$

where $p(x)$ is the probability that $X = x$. Moreover, the maximum of $H(X)$ is $\log|M|$ when $p(x) = \frac{1}{|M|}$ for all $x \in M$ (pure random, most difficult to compress); the minimum is 0 when $p(x) = 1$ for a particular $x \in M$ (deterministic and constant, easiest to compress). For the binomial case, if all symbols are all 0 or all 1 in the string, we do not have to store anything because by knowing the number of ones in the string and the sizes of matrix, the receiver is already able to decode the data completely.

With this observation in mind, the goal is to organize the matrix (graph) into some homogeneous sub-matrices with low entropy and compress them separately, as we will describe next.

4.2 Graph Segment encoding

Given a graph stream segment $\mathcal{G}^{(s)}$ and its partition assignments, we can precisely compute the cost for transmitting the segment as two parts: 1) *Partition encoding cost*: the model complexity for partition assignments, 2) *Graph encoding cost*: the actual code for the graph segment.

Partition encoding cost

The description complexity for transmitting the partition assignments for graph segment $\mathcal{G}^{(s)}$ consists of the following terms:

First, we need to send the number of source and destination nodes m and n using $\log^\star m + \log^\star n$ bits. Note that, this term is constant, which has no effect on the choice of final partitions.

Second, we shall send the number of source and destination partitions which is $\log^\star k_s + \log^\star \ell_s$.

Third, we shall send the source and destination partition assignments. To exploit the non-uniformity across partitions, the encoding cost is $mH(P) + nH(Q)$ where P is a multinomial random variable with the probability $p_i = \frac{m_i^{(s)}}{m}$ and $m_i^{(s)}$ is the size of i-th source partition $1 \le i \le k_s$). Similarly, Q is another multinomial random variable with $q_i = \frac{n_i^{(s)}}{n}$ and $n_i^{(s)}$ is the size of i-th destination partition, $1 \le i \le \ell_s$.

For example in Figure 2, the partition sizes for first segment $\mathcal{G}^{(1)}$ are $m_1^{(1)} = m_2^{(1)} = 2$, $n_1^{(1)} = 1$, and $n_2^{(1)} = 2$; the partition assignments for $\mathcal{G}^{(1)}$ costs $-4(\frac{2}{4}\log(\frac{2}{4}) + \frac{2}{4}\log(\frac{2}{4})) - 3(\frac{1}{3}\log(\frac{1}{3}) + \frac{2}{3}\log(\frac{2}{3}))$ bits.

In summary, the partition encoding cost for graph segment $\mathcal{G}^{(s)}$ is

$$\begin{aligned} C_p^{(s)} \quad := \quad & \log^\star m + \log^\star n + \log^\star k_s + \log^\star \ell_s + \\ & mH(P) + nH(Q) \end{aligned} \tag{2}$$

[1]$|E|$ is needed for computing the probability of ones or zeros, which is required for several encoding scheme such as Huffman coding

[2]To encode a positive integer x, we need $\log^\star x \approx \log_2 x + \log_2 \log_2 x + \ldots$, where only the positive terms are retained and this is the optimal length, if the range of x is unknown [19]

where P and Q are multinomial random variables for source and destination partitions, respectively.

Graph encoding cost

After transmitting the partition encoding, the actual graph segment $\mathcal{G}^{(s)}$ is transmitted as $k_s\ell_s$ subgraph segments. To facilitate the discussion, we define the entropy term for a subgraph segment $\mathcal{G}^{(s)}_{p,q}$ as

$$H(\mathcal{G}^{(s)}_{p,q}) = -\big(\rho^{(s)}_{p,q}\log\rho^{(s)}_{p,q} + (1-\rho^{(s)}_{p,q})\log(1-\rho^{(s)}_{p,q})\big) \quad (3)$$

where $\rho^{(s)}_{p,q} = \frac{|E|^{(s)}_{p,q}}{|\mathcal{G}^{(s)}_{p,q}|}$ is the density of subgraph segment $\mathcal{G}^{(s)}_{p,q}$. Intuitively, $H(\mathcal{G}^{(s)}_{p,q})$ quantifies how difficult it is to compress the subgraph segment $\mathcal{G}^{(s)}_{p,q}$. In particular, if the entire subgraph segment is all 0 or all 1 (the density is exactly 0 or 1), the entropy term becomes 0.

With this, the graph encoding cost is

$$C^{(s)}_g := \sum_{p=1}^{k_s}\sum_{q=1}^{\ell_s}\big(\log^{\star}|E|^{(s)}_{p,q} + |\mathcal{G}^{(s)}_{p,q}|\cdot H(\mathcal{G}^{(s)}_{p,q})\big). \quad (4)$$

where $|E|^{(s)}_{p,q}$ is the number of edges in the (p,q) sub-graphs of segment s; $|\mathcal{G}^{(s)}_{p,q}|$ is the size of sub-graph segment, i.e, $m^{(s)}_p n^{(s)}_q (t_{s+1}-t_s)$, and $H(\mathcal{G}^{(s)}_{p,q})$ is the entropy of the subgraph segment defined in equation 3.

In the sub-graph segment $\mathcal{G}^{(1)}_{2,2}$ of Figure 2, the number of edges $|E|^{(1)}_{2,2} = 3+4$, $\mathcal{G}^{(1)}_{2,2}$ has the size $|\mathcal{G}^{(1)}_{2,2}| = 2\times 2\times 2$, the density $\rho^{(1)}_{2,2} = \frac{7}{8}$, and the entropy $H(\mathcal{G}^{(1)}_{2,2}) = -(\frac{7}{8}\log\frac{7}{8} + \frac{1}{8}\log\frac{1}{8})$.

Putting everything together, we obtain the segment encoding cost as the follows:

Definition 4.1 (Segment encoding cost).

$$C^{(s)} := \log^{\star}(t_{s+1}-t_s) + C^{(s)}_p + C^{(s)}_g. \quad (5)$$

where $t_{s+1}-t_s$ is the segment length, $C^{(s)}_p$ is the partition encoding cost, $C^{(s)}_g$ is the graph encoding cost.

4.3 Graph stream encoding

Given a graph stream $\mathcal{G}$, we partition it into a number of graph segments $\mathcal{G}^{(s)} (s \geq 1)$ and compress each segment separately such that the total encoding cost is small.

Definition 4.2 (Total cost). *The total encoding cost is*

$$C := \textstyle\sum_s C^{(s)}. \quad (6)$$

where $C^{(s)}$ is the encoding cost for s-th graph stream segment.

For example in Figure 2, the encoding cost C up to timestamp 3 is the sum of the costs of two graph stream segments $\mathcal{G}^{(1)}$ and $\mathcal{G}^{(2)}$.

Intuition. Intuitively, our encoding cost objective tries to decompose the graph into subgraphs that are homogeneous, i.e., close to either fully-connected (cliques) or fully-disconnected. Additionally, if such cliques are stable over time, then it places subsequent graphs into the same segment. The encoding cost penalizes a large number of cliques or lack of homogeneity. Hence, our model selection criterion favors simple enough decompositions that adequately capture the essential structure of the graph over time.

Having defined the objective precisely in equation 3 and equation 4, the next step is to search for optimal partition and time segmentation. However, finding the optimal solution is NP-hard[3]. Next, in Section 5, we present an alternating minimization method coupled with an incremental segmentation process to perform the overall search.

5. GRAPHSCOPE

In this section we describe our method, GraphScope by solving the two problems proposed in Section 3.2. The goal is to find the appropriate number and position of change-points, and the number and membership of source and destination partitions so that the cost of (6) is minimized. Exhaustive enumeration is prohibitive, and thus we resort to alternating minimization. Note that we drop the subscript s on k_s and ℓ_s whenever it is clear from the context.

Specifically, we have two steps: (a) how to find good communities (source and destination partitions), for a given set of graph snapshots that belong to the same segment. (b) when to declare a time-tick as a *change point* and start a new graph segment. We describe each next.

5.1 Partition Identification

Here we explain how to find source and destination partitions for a given graph segment $\mathcal{G}^{(s)}$. In order to do that, we need to answer the following two questions:

- How to find the best partitioning given the number of source and destination partitions?
- How to search for the appropriate number of source and destination partitions?

Next, we present the solution for each step.

Algorithm 1: ReGroup(Graph Segment $\mathcal{G}^{(s)}$; partition size k,ℓ; initial partitions $I^{(s)}$, $J^{(s)}$)

```
1  Compute density $\rho^{(s)}_{p,q}$ for all $p, q$ based on $I^{(s)}, J^{(s)}$.
   repeat
2      forall source $s$ in $\mathcal{G}^{(s)}$ do
           // assign $s$ to the most similar partition
3          $s$ is split in $\ell$ parts
4          compute source density $p_i$ for each part
5          assign $s$ to source partition with the minimal
           encoding cost (Equation 8).
6      Update destination partitions similarly
7  until no change;
```

Finding the best partitioning

Given the number of the best source and destination partitions k and ℓ, we want to re-group sources and destinations into the better partitions. Typically this regrouping procedure alternates between source and destination nodes. Namely, we update the source partitions with respect to the current destination partitions, and vice versa. More specifically, we alternate the following two steps until it converges:

- **Update source partitions:** for each source (a row of the graph matrix), consider assigning it to the source partition that incurs the smallest encoding cost

[3]It is NP-hard since, even allowing only column re-ordering, a reduction to the TSP problem can be found [11].

- **Update destination partitions:** Similarly, for each destination (column), consider assigning it to the destination partition that yields smaller encoding cost.

The cost of assigning a row to a row-group is discussed later (see (8)). The pseudo-code is listed in Algorithm 1. The initialization of Algorithm 1 is discussed separately in Section 5.3.

Determining the number of partitions

Given different values for k and ℓ, we can easily run Algorithm 1 and choose those leading to a smaller encoding cost. However, the search space for k and ℓ is still too large to perform exhaustive tests. The central idea is to do local search around some initial partition assignments, and adjust the number of partitions k and ℓ as well as the partition assignments based on the encoding cost.

```
Algorithm 2: SEARCHKL(Graph Segment $\mathcal{G}^{(s)}$; initial partition size $k$,$\ell$; initial partitions $I^{(s)}$, $J^{(s)}$)
1  repeat
       // try to merge source partitions
2      repeat
3          Find the source partition pair $(x, y)$ s.t. merging $x$ and $y$ gives smallest encoding cost for $\mathcal{G}^{(s)}$.
4          if total encoding decrease then merge $x$,$y$
5      until no more merge;
       // try to split source partition
6      repeat
7          Find source partition $x$ with largest average entropy per node.
8          foreach source $s$ in $x$ do
9              if average entropy reduces without $s$ then
10                 assign $s$ to the new partition
11         ReGroup($\mathcal{G}^{(s)}$, updated partitions)
12     until no decrease in encoding cost;
13     Search destination partitions similarly
14 until no changes;
```

Cost computation for partition assignments

Here we present the details of how to compute the encoding cost of assigning a node to a particular partition. Our discussion focuses on assigning a source node to a source partition. The assignment for a destination node is symmetric.

Recall a graph segment $\mathcal{G}^{(s)}$ consists of $(t_{s+1} - t_s)$ graphs, $G^{(t_s)}, \ldots, G^{(t_{s+1}-1)}$. For example in Figure 2, $\mathcal{G}^{(1)}$ consists of 2 graphs, $G^{(1)}$ and $G^{(2)}$. Likewise, every source node in a graph segment $\mathcal{G}^{(s)}$ is associated with $(t_{s+1} - t_s)$ sets of edges in these $(t_{s+1} - t_s)$ graphs. Therefore, the total number of possible edges out of one source node in $\mathcal{G}^{(s)}$ is $(t_{s+1} - t_s)n$.

Furthermore, the destination partitions $J_i^{(s)}$ divide the destination nodes into ℓ disjoint sets with size $n_i^{(s)}$ ($1 \leq i \leq \ell$, $\sum_i n_i^{(s)} = n$). For example, $\mathcal{G}^{(1)}$ of Figure 2 has two destination partitions ($\ell = 2$), where the first destination partition $J_1^{(1)} = \{1\}$, and the second destination partition $J_2^{(1)} = \{2, 3\}$.

Similarly, all the edges from a single source node in graph segment $\mathcal{G}^{(s)}$ are also split into these ℓ sets. In $\mathcal{G}^{(1)}$ of Figure 2, the edges from the 4-th source node are split into two sets, where the first set $J_1^{(1)}$ has 0 edges and the second set $J_2^{(1)}$ 3 edges[4].

More formally, the edge pattern out of a source node is generated from ℓ binomial distributions $\mathbf{p}_i (1 \leq i \leq \ell)$ with respect to ℓ destination partitions. Note that $\mathbf{p}_i(1)$ is the density of the edges from that source node to the destination partition $J_i^{(s)}$, and $\mathbf{p}_i(0) = 1 - \mathbf{p}_i(1)$. In $\mathcal{G}^{(1)}$ of Figure 2, the 4-th source node has $\mathbf{p}_1(1) = 0$ since there are 0 edges from 4 to $J_1^{(1)} = \{1\}$, and $\mathbf{p}_1(1) = \frac{3}{4}$ since 3 out of 4 possible edges from 4 to $J_1^{(2)} = \{2, 3\}$.

One possible way of encoding the edges of one source node is based on precisely these distributions $\mathbf{p}_i$, but as we shall see later, this is not very practical. More specifically, using the "true" distributions $\mathbf{p}_i$, the encoding cost of the source node's edges in the graph segment $\mathcal{G}^{(s)}$ would be

$$C(\mathbf{p}) = (t_{s+1} - t_s) \textstyle\sum_{i=1}^{\ell} n_i H(\mathbf{p}_i) \qquad (7)$$

where $(t_{s+1} - t_s)$ is the number of graphs in the graph segment, n is the number of possible edges out of a source node for each graph[5], $H(\mathbf{p}_i) = \sum_{x=\{0,1\}} \mathbf{p}_i(x) \log \mathbf{p}_i(x)$ is the entropy for the each source node's partition.

In $\mathcal{G}^{(1)}$ of Figure 2, the number of graphs is $t_{s+1} - t_s = 3-1=2$; the number of possible edges out of the 4-th source node $n = 3$; therefore, the 4-th source node costs $2\times3\times(0 + \frac{3}{4}\log\frac{3}{4} + \frac{1}{4}\log\frac{1}{4}) = 2.25$. Unfortunately, this is not practical to do so for every source node, because the model complexity is too high. More specifically, we have to store additional $m\ell$ integers in order to decode all source nodes.

The practical option is to group them into a handful of source partitions and to encode/decode one partition at a time instead of one node at a time. Similar to a source node, the edge pattern out of a source partition is also generated from ℓ binomial distributions $\mathbf{q}_i$ ($1 \leq i \leq \ell$). Now we encode the i-th source node based on the distribution $\mathbf{q}_i$ for a partition instead of the "true" distribution $\mathbf{p}_i$ for the node. The encoding cost is

$$C(\mathbf{p}, \mathbf{q}) = (t_{s+1} - t_s) \textstyle\sum_{i=1}^{\ell} n_i H(\mathbf{p}_i, \mathbf{q}_i) \qquad (8)$$

where $H(\mathbf{p}_i, \mathbf{q}_i) = \sum_{x=\{0,1\}} \mathbf{p}_i(x) \log \mathbf{q}_i(x)$ is the cross-entropy. Intuitively, the cross-entropy is the encoding cost when using the distribution $\mathbf{q}_i$ instead of the "true" distribution $\mathbf{p}_i$. In $\mathcal{G}^{(1)}$ of Figure 2, the cost of assigning the 4-th node to second source partition $I_2^{(1)}$ is $2\times3\times(0+\frac{3}{4}\log\frac{7}{8}+\frac{1}{4}\log\frac{1}{8}) = 2.48$ which is slightly higher than using the true distribution that we just computed (2.25). However, the model complexity is much lower, i.e., $k\ell$ integers are needed instead of $m\ell$.

5.2 Time Segmentation

So far, we have discussed how to partition the source and destination nodes given a graph segment $\mathcal{G}^{(s)}$. Now we present the algorithm to construct the graph segments incrementally when new graph snapshots at arrive every time-tick. Intuitively, we want to group "similar" graphs from consecutive timestamps into one graph segment and encode them all together. For example, in Figure 2, graphs $G^{(1)}$ and $G^{(2)}$ are similar (only one different edge), and therefore

[4]One edge from 4 to 3 in $G^{(1)}$, two edges from 4 to 2 and 3 in $G^{(2)}$ in Figure 2.

[5] $(t_{s+1} - t_s)n$ is the total number of possible edges of a source node in the graph segment

we group them into one graph segment, $\mathcal{G}^{(1)}$. On the other hand, $G^{(3)}$ is quite different from the previous graphs, and hence we start a new segment $\mathcal{G}^{(2)}$ whose first member is $G^{(3)}$.

The guiding principle here is still the encoding cost. More specifically, the algorithm will combine the incoming graph with the current graph segment if there is a storage benefit, otherwise we start a new segment with that graph. The meta-algorithm is listed in Algorithm 3.

Algorithm 3: GraphScope(Graph Segment $\mathcal{G}^{(s)}$; Encoding cost c_o; New Graph $G^{(t)}$

```
  output: updated graph segment, new partition
          assignment $I^{(s)}$, $J^{(s)}$
1 Compute new encoding $c_n$ of $\mathcal{G}^{(s)} \bigcup \{G^{(t)}\}$
2 Compute encoding cost $c$ for just $G^{(t)}$
  // check if there is any encoding benefit
3 if $c_n - c_o < c$ then
      // add $G^{(t)}$ in $\mathcal{G}^{(s)}$
4     $\mathcal{G}^{(s)} \leftarrow \mathcal{G}^{(s)} \bigcup \{G^{(t)}\}$
5     SearchKL for updated $\mathcal{G}^{(s)}$
6 else
      // start a new segment from $G^{(t)}$
7     $\mathcal{G}^{(s+1)} := \{G^{(t)}\}$
8     SearchKL for new $\mathcal{G}^{(s+1)}$
```

5.3 Initialization

Once we decide to start a new segment, how should we initialize the number and membership of its partitions? There are several ways to do the initialization. Trading-off convergence speed versus compression quality, we propose and study two alternatives:

Fresh-start. One option is to start from a small k and ℓ, typically $k = 1$ and $\ell = 1$, and progressively increase them (see Algorithm 2) as well as re-group sources and destinations (see Algorithm 1). From our experiments, this scheme is very effective in leading to a good result. In terms of computational cost, it is relatively fast, since we start with small k and ℓ.

Resume. For time evolving graphs, consecutive graphs often have a strong similarity. We can leverage this similarity in the search process by starting from old partitions. More specifically, we initialize k_{s+1} and ℓ_{s+1} to k_s and ℓ_s, respectively. Additionally, we initialize $I^{(s+1)}$ and $J^{(s+1)}$ to $I^{(s)}$ and $J^{(s)}$. We study the relative CPU performance of *fresh-start* and *resume* in Section 6.3.

6. EXPERIMENTAL EVALUATION

In this section, we will evaluate the result on both *community discovery* and *change detection* of GraphScope using several real, large graph datasets. We first describe the datasets in Section 6.1. Then we present our experiments, which are designed to answer the following two questions:

- *Mining Quality:* How good is our method in terms of finding meaningful communities and change points (Section 6.2).
- *Speed:* How fast is it, and how does it scale up (Section 6.3).

Finally, we present some additional mining observations that our method automatically identifies. To the best of our knowledge, no other parameter-free and incremental method for time-evolving graphs has been proposed to date. Our goal is to automatically determine the best change-points in time, as well as the best node partitionings, which concisely reveal the basic structure of both communities as well as their change over time. It is not clear how parameters of other methods (e.g., number of partitions, graph similarity thresholds, etc) should be set for these methods to attain this goal. GraphScopeis fully automatic and, as we will show, still able to find meaningful communities and change points.

6.1 Datasets

In this section, we describe the datasets in our experiments.

name	m**-by-**n	**avg.**$\lvert E \rvert$	**time T**
NETWORK	29K-by-29K	12K	1, 222
ENRON	34k-by-34k	15K	165
CELLPHONE	97-by-3764	430	46
DEVICE	97-by-97	689	46
TRANSACTION	28-by-28	132	51

Table 2: Dataset summary

The NETWORK *Dataset*

The traffic trace consists of TCP flow records collected at the backbone router of a class-B university network. Each record in the trace corresponds to a directional TCP flow between two hosts, with timestamps indicating when the flow started and finished. With this traffic trace, we use a window size of one hour to construct the source-destination graph stream. Each graph is represented by a sparse adjacency matrix with the rows and the columns corresponding to source and destination IP addresses, respectively. An edge in a graph $G^{(t)}$ means that there exist TCP flows (packets) sent from the i-th source to the j-th destination during the t-th hour. The graphs involve m=n=21,837 unique campus hosts (the number of source and destination nodes) with a per-timestamp average of over 12K distinct connections (the number of edges). The total number of timestamps T is 1,222. Figure 3(a) shows an example of superimposing[6] all source-destination graphs in one time segment of 18 hours. Every row/column corresponds to a source/destination; the dot there indicates there is at least a packet from the source to the destination during that time segment. The graphs are correlated, with most of the traffic to or from a small set of server-like hosts.

GraphScope automatically exploits the sparsity and correlation by organizing the sources and destinations into homogeneous groups as shown in Figure 3(b).

The ENRON *Dataset*

This consists of the email communications in Enron Inc. from Jan 1999 to July 2002 [2]. We construct sender-to-recipient graphs on a weekly basis. The graphs have $m = n = 34,275$ senders/recipients (the number of nodes) with

[6] Two graphs are superimposed together by taking the union of their edges.

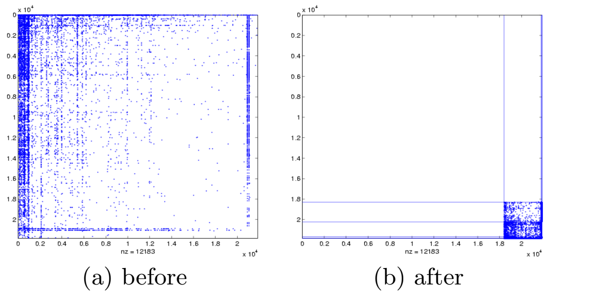


Figure 3: NETWORK before and after GraphScope for the graph segment between Jan 7 1:00, 2005 and Jan 7 19:00, 2005. GraphScope successfully rearrange the sources and destinations such that the sub-matrices are much more homogeneous.

an average of 1,479 distinct sender-receiver pairs (the number of edges) every week.

Like the NETWORK dataset, the graphs in ENRON are also correlated. GraphScope can reorganize the graph into homogeneous partitions (see the visual comparison in Figure 4).

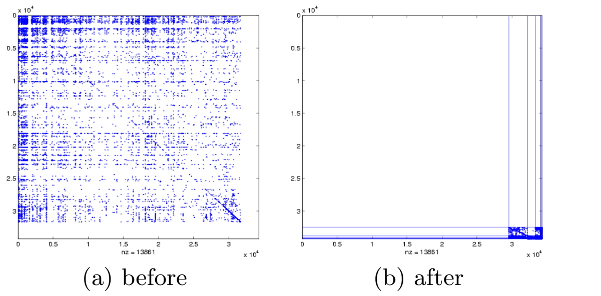


Figure 4: ENRON before and after GraphScope for the graph segment of week 35, 2001 to week 38, 2001. GraphScope can achieve significant compression by partitioning senders and recipients into homogeneous groups

The CELLPHONE *Dataset*

The CELLPHONE dataset records the cellphone activity for m=n=97 users from two different labs in MIT [1]. Each graph snapshot corresponds to a week, from Jan 2004 to May 2005. We thus have T=46 graphs, one for each week, excluding weeks with no activity.

We plot the superimposed graphs of weeks 38 to 42 in 2004 at Figure 5(a), which looks much more random than NETWORK and ENRON. However, GraphScope is still able to extract the hidden structure from the graph as shown in Figure 5(b), which looks much more homogeneous (more details in Section 6.2).

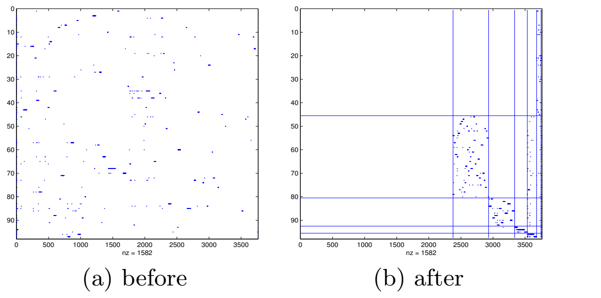


Figure 5: CELLPHONE before and after GraphScope, for the period of week 38 to 42 in 2004

The DEVICE *dataset*

DEVICE dataset is constructed on the same 97 users whose cellphones periodically scan for nearby phones and computers over Bluetooth. The goal is to understand people's behavior from their proximity to others. Figure 6(a) plots the superimposed user-to-user graphs for one time segment where every dot indicates that the two corresponding users are physically near each other. Note that the first row represents all the devices that do not belong to any of the 97 individual users (mainly laptop computers, PDAs and other peoples' cellphones). Figure 6(b) shows the resulting user partitions for that time segment, where cluster structure is revealed (see Section 6.2 for details).

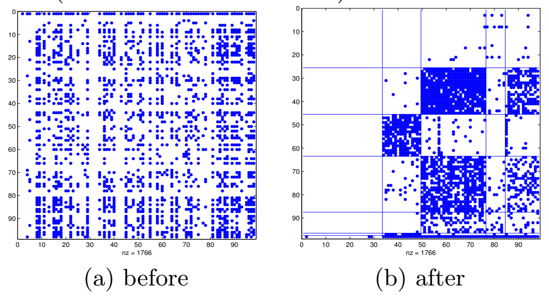


Figure 6: DEVICE before and after GraphScope for the time segment between week 38, 2004 and week 42, 2004. Interesting communities are identified

The Transaction Dataset

The TRANSACTION dataset has m=n=28 accounts of a company, over 2,200 days. An edge indicates that the source account had funds transfered to the destination account. Each graph snapshot covers transaction activities over a window of 40 days, resulting in T=51 time-ticks for our dataset.

Figure 7(a) shows the transaction graph for one timestamp. Every black square at the (i, j) entry in Figure 7(a) indicates there is at least one transaction debiting the ith account and crediting the jth account. After applying GraphScope on that timestamp (see Figure 7(b)), the accounts are organized into very homogeneous groups with a few exceptions (more details in Section 6.2).

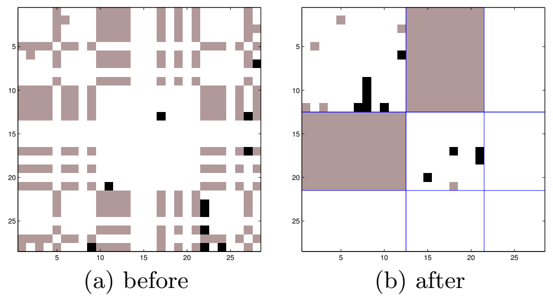


Figure 7: TRANSACTION before and after GraphScope for a time segment of 5 months. GraphScope is able to group accounts into partitions based on their types. Darker color indicates multiple edges over time.

6.2 Mining Case-studies

Now we qualitatively present the mining observation on all the datasets. More specifically, we illustrate that (1) source and destination groups correspond to semantically meaningful clusters; (2) the groups evolve over time; (3) time segments indicate interesting change-points.

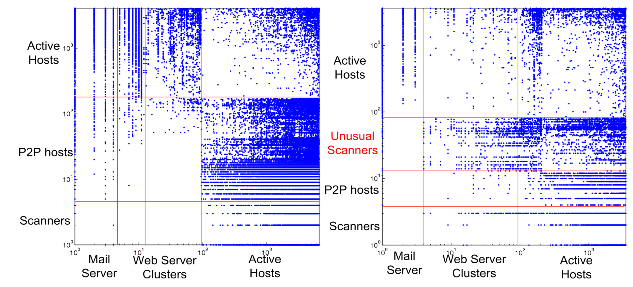


Figure 8: `NETWORK` zoom-in (log-log plot): (a) Source nodes are grouped into active hosts and security scanning program; Destination nodes are grouped into active hosts, clusters, web servers and mail servers. (b) on a different time segment, a group of unusual scanners appears, in addition to the earlier groups.

`NETWORK`: *Interpretable groups*

Despite the bursty nature of network traffic, GraphScope can successfully cluster the source and destination hosts into meaningful groups. Figure 8(a) and (b) show the active source and destination nodes organized by groups for two different time segments. Note that Figure 8 is in log-log scale to visualize those small partitions. For example, source nodes are grouped into (1) active hosts which talk to a small number of hosts, (2) P2P hosts that scan a number of hosts, and (3) administrative scanning hosts[7] which scan many hosts. Similarly, destination hosts are grouped into (1) active hosts, (2) cluster servers at which many students login remotely to work on different tasks, (3) web servers which hosts the websites of different schools, and (4) mail servers that have the most incoming connections. The main difference between Figure 8(a) and (b) is that a source group of unusual scanners emerges in the latter. GraphScope can automatically identify the change and decide to split into two time segments.

`CELLPHONE`: *Evolving groups*

As in `NETWORK`, we also observe meaningful groups in `CELLPHONE`. Figure 9 (a) illustrate the calling patterns in the fall semester of 2004, where two strong user partitions (G1 and G2) exist. The dense small partition G3 is the service call in campus, which has a lot of incoming calls from everyone. Figure 9 (b) illustrates that the calling patterns changed during the winter break which follows the fall semester.

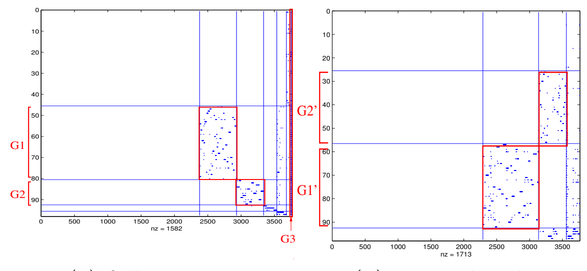
`DEVICE`: *Evolving groups*

The evolving group behavior is also observed in the `DEVICE` dataset. In particular, two dense partitions appear in Figure 10(a): after inspecting the user ids and their attributes, we found that the users in group $U1$ are all from the same school with similar schedule, probably taking the same class; the users in $U2$ all work in the same lab. In a later time segment (see Figure 10(b)), the partition $U1$ disappeared, while the partition $U2$ is unchanged.

`TRANSACTION`

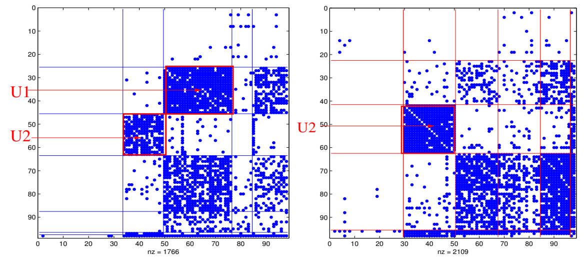
As shown in Figure 7(b), GraphScope successfully organizes the 28 accounts into three partitions. Upon closer inspection, these groups correspond to the different functional groups of the accounts (e.g., 'marketing', 'sales')[8]. In Figure 7(b), the interaction between first source partition (from the top) and second destination partition (from the left) correspond to mainly the transactions from assets accounts to liability and revenue accounts, which obeys common business practice.

[7]The campus network is constantly running some port-scanning program to identify potential vulnerabilities of the in-network hosts.



(a) fall semester (b) winter break

Figure 9: `CELLPHONE`: a) Two calling groups appear during the fall semester; b) Call groups changed in the winter break. The change point corresponds to the winter break.



(a) two groups (b) one group disappeared

Figure 10: `DEVICE`: (a) two groups are prominent. Users in $U1$ are all from the same school with similar schedule possibly taking the same class; Users in $U2$ are all working in the same lab. (b) $U1$ disappears in the next time segment, while $U2$ remains unchanged.

`ENRON`: *Change-point detection*

The source and destination partitions usually correspond to meaningful clusters for the given time segment. Moreover, the time segments themselves usually encode important information about changes. Figure 1 plots the encoding cost difference between incorporating the new graph into the current time segment vs. starting a new segment. The vertical lines on Figure 1 are the top 10 splits with largest cost savings when starting a new segment, which actually correspond to the key events related to Enron Inc. Moreover, the intensity in terms of magnitude and frequency dramatically increases around Jan 2002 which coincides with several key incidents such as the investigation on document shredding, and the CEO resignation.

6.3 Quality and Scalability

We compare *fresh-start* and *resume* (see Section 5.3) in terms of compression benefit, against the global compression estimate and the space requirement for the original graphs, stored as sparse matrices (adjacency list representation). Figure 11 shows that both *fresh-start* and *resume*

[8] Due to anonymity requirements, the account types are obfuscated.

GraphScope achieve high compression gain (less than 4% of the original space), which is even better than the global compression on the graphs (the 3rd bar for each dataset). Our two variations require about the same space.

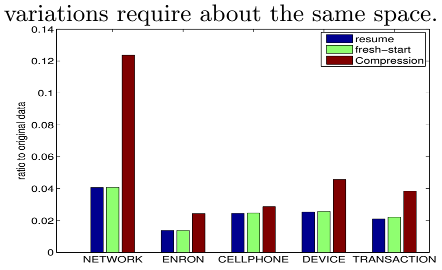
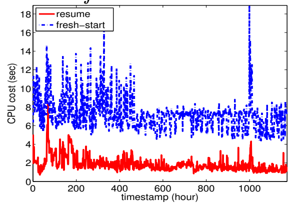


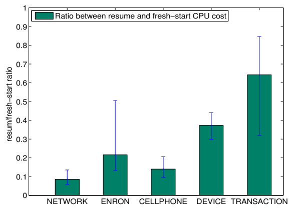
Figure 11: Relative Encoding Cost: Both *resume* and *fresh-start* methods give over an order of magnitude space saving compared to the raw data and are much better than global compression on the raw data.

Now we illustrate the CPU cost (scalability) of *fresh-start* and *resume*. As shown in Figure 12(a) for NETWORK (similar result are achieved for the other datasets, hence omitted), the CPU cost per timestamp/graph is stable over time for both *fresh-start* and *resume*, which suggests that both proposed methods are scalable to streaming environments.

Furthermore, *resume* is much faster than *fresh-start* as shown in Figure 12(b), especially for large graphs such as in NETWORK. There, *resume* only uses 10% of CPU time compared to *fresh-start*.



(a) CPU cost (NETWORK)



(b) Relative CPU

Figure 12: CPU cost: (a) the CPU costs for both *resume* and *fresh-start* GraphScope are stable over time; (b) *resume* GraphScope is much faster than *fresh-start* GraphScope on the same datasets (the error bars give 25% and 75% quantiles).

7. CONCLUSIONS

We propose GraphScope, a parameter-free scheme to mine streams of graphs. Our method has all of the following desired properties: 1) It is rigorous and automatic, with no need for user-defined parameters. Instead, it uses the Minimum Description Language (MDL) principle, to decide how to form communities, and when to modify them. 2) It is fast and scalable, carefully designed to work in a streaming setting. 3) It is effective, discovering meaningful communities and meaningful transition points.

We also present experiments on several real datasets, spanning 500 Gigabytes. The datasets were from widely diverse applications (university network traffic, email from the Enron company, cellphone call logs and Bluetooth connections). Because of its generality and its information theoretic underpinnings, GraphScope is able to find meaningful groups and patterns in all the above settings, without any specific fine-tuning on our side.

Future research directions include extensions to create hierarchical groupings, both of the communities as well as of the time segments.

Acknowledgements: This material is based upon work supported by the National Science Foundation under Grants No. IIS-0326322 IIS-0534205 and under the auspices of the U.S. Department of Energy by University of California Lawrence Livermore National Laboratory under contract No.W-7405-ENG-48. This work is also partially supported by the Pennsylvania Infrastructure Technology Alliance (PITA), an IBM Faculty Award, a Yahoo Research Alliance Gift, with additional funding from Intel, NTT and Hewlett-Packard. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation, or other funding parties.

8. REFERENCES

[1] http://reality.media.mit.edu/download.php.
[2] http://www.cs.cmu.edu/ enron/.
[3] C. C. Aggarwal and P. S. Yu. Online analysis of community evolution in data streams. In *SDM*, 2005.
[4] R. Agrawal, R. Bayardo, C. Faloutsos, J. Kiernan, R. Rantzau, and R. Srikant. Auditing compliance with a hippocratic database. *VLDB*, 2004.
[5] L. Backstrom, D. P. Huttenlocher, J. M. Kleinberg, and X. Lan. Group formation in large social networks: membership, growth, and evolution. In *KDD*, pages 44–54, 2006.
[6] D. Chakrabarti. Autopart: Parameter-free graph partitioning and outlier detection. In *PKDD*, pages 112–124, 2004.
[7] D. Chakrabarti, S. Papadimitriou, D. S. Modha, and C. Faloutsos. Fully automatic cross-associations. In *KDD*, pages 79–88. ACM Press, 2004.
[8] T. M. Cover and J. A. Thomas. *Elements of information theory*. Wiley-Interscience, New York, NY, USA, 1991.
[9] I. S. Dhillon, S. Mallela, and D. S. Modha. Information-theoretic co-clustering. In *KDD*, pages 89–98, 2003.
[10] M. Faloutsos, P. Faloutsos, and C. Faloutsos. On power-law relationships of the internet topology. In *SIGCOMM*, 1999.
[11] D. S. Johnson, S. Krishnan, J. Chhugani, S. Kumar, and S. Venkatasubramanian. Compressing large boolean matrices using reordering techniques. In *VLDB*, pages 13–23, 2004.
[12] G. Karypis and V. Kumar. Multilevel k-way partitioning scheme for irregular graphs. *Journal of Parallel and Distributed Computing*, 48(1):96–129, 1998.
[13] E. Keogh, S. Lonardi, and C. A. Ratanamahatana. Towards parameter-free data mining. In *KDD*, pages 206–215, New York, NY, USA, 2004. ACM Press.
[14] R. Kumar, P. Raghavan, S. Rajagopalan, and A. Tomkins. Extracting large-scale knowledge bases from the web. In *VLDB*, 1999.
[15] J. Leskovec, J. Kleinberg, and C. Faloutsos. Graphs over time: Densification laws, shrinking diameters and possible explanations. In *SIGKDD*, 2005.
[16] A. Y. Ng, M. I. Jordan, and Y. Weiss. On spectral clustering: Analysis and an algorithm. In *NIPS*, pages 849–856, 2001.
[17] C. C. Noble and D. J. Cook. Graph-based anomaly detection. In *KDD*, pages 631–636, 2003.
[18] A. Pinar, T. Tao, and H. Ferhatosmanoglu. Compressing bitmap indices by data reorganization. In *ICDE*, pages 310–321, 2005.
[19] J. Rissanen. A universal prior for integers and estimation by minimum description length. *Annals of Statistics*, 11(2):416–431, 1983.
[20] J. Sun, D. Tao, and C. Faloutsos. Beyond streams and graphs: dynamic tensor analysis. In *KDD*, pages 374–383, 2006.