Colibri: Fast Mining of Large Static and Dynamic Graphs

Hanghang Tong
Carnegie Mellon University
Pittsburgh, PA, USA
htong@cs.cmu.edu

Spiros Papadimitriou
IBM T.J. Watson
Hawthorne, NY, USA
spapadim@us.ibm.com

Jimeng Sun
IBM T.J. Watson
Hawthorne, NY, USA
jimeng@us.ibm.com

Philip S. Yu
University of Illinois at Chicago
Chicago, IL, USA
psyu@cs.uic.edu

Christos Faloutsos
Carnegie Mellon University
Pittsburgh, PA, USA
christos@cs.cmu.edu

ABSTRACT

Low-rank approximations of the adjacency matrix of a graph are essential in finding patterns (such as communities) and detecting anomalies. Additionally, it is desirable to track the low-rank structure as the graph evolves over time, efficiently and within limited storage. Real graphs typically have thousands or millions of nodes, but are usually very sparse. However, standard decompositions such as SVD do not preserve sparsity. This has led to the development of methods such as CUR and CMD, which seek a non-orthogonal basis by sampling the columns and/or rows of the sparse matrix.

However, these approaches will typically produce overcomplete bases, which wastes both space and time. In this paper we propose the family of *Colibri* methods to deal with these challenges. Our version for static graphs, *Colibri-S*, iteratively finds a non-redundant basis and we prove that it has *no* loss of accuracy compared to the best competitors (CUR and CMD), while achieving significant savings in space and time: on real data, *Colibri-S* requires much less space and is *orders of magnitude* faster (in proportion to the square of the number of non-redundant columns). Additionally, we propose an efficient update algorithm for dynamic, time-evolving graphs, *Colibri-D*. Our evaluation on a large, real network traffic dataset shows that *Colibri-D* is over *100 times* faster than the best published competitor (CMD).

Categories and Subject Descriptors

H.2.8 [**Database Management**]: Database Applications – Data Mining

General Terms

Algorithm, experimentation

Keywords

Low-rank approximation, scalability, graph mining

KDD'08, August 24–27, 2008, Las Vegas, Nevada, USA.

1. INTRODUCTION

Graphs appear in a wide range of settings, like computer networks, the world wide web, biological networks, social networks and many more. How can we find patterns, e.g. communities and anomalies, in a large sparse graph? How can we track such patterns of interest if the graph is evolving over time?

A common representation of a graph is a matrix, such as an adjacency matrix for a unipartite graph where every row/column corresponds to a node in the graph, and every non-zero entry is an edge; an interaction matrix for a bipartite graph where rows and columns correspond to two different types of nodes and non-zero entries denote edges between them.

Naturally, low-rank approximations on matrices provide powerful tools to answer the above questions. Formally, a rank-c approximation of matrix $\mathbf{A}$ is a matrix $\tilde{\mathbf{A}}$ where $\tilde{\mathbf{A}}$ is of rank c and $\|\tilde{\mathbf{A}} - \mathbf{A}\|$ is small. The low-rank approximation is usually presented in a factorized form e.g., $\tilde{\mathbf{A}} = \mathbf{LMR}$ where $\mathbf{L}$, $\mathbf{M}$, and $\mathbf{R}$ are of rank-c.

Depending on the properties of those matrices, many different approximations have been proposed in the literature. For example, in SVD [14], $\mathbf{L}$ and $\mathbf{R}$ are orthogonal matrices whose columns/rows are singular vectors and $\mathbf{M}$ is a diagonal matrix whose diagonal entries are singular values. Among all the possible rank-c approximations, SVD gives the best approximation in terms of squared error. However, the SVD is usually dense (i.e., most of the entries are non-zero), even if the original matrix is sparse. Furthermore, the singular vectors are abstract notions of best orthonormal basis, which is not intuitive for the interpretation of data analysis results.

Recently, alternatives have started to appear, such as CUR [8] and CMD [28], which use the actual columns and rows of the matrix to form $\mathbf{L}$ and $\mathbf{R}$. We call these *example-based low-rank approximations*. The benefit is that they provide an intuitive as well as sparse representation, since $\mathbf{L}$ and $\mathbf{R}$ are directly sampled from the original matrix. However, the approximation is often sub-optimal compared to SVD and the matrix $\mathbf{M}$ is no longer diagonal, which means a more complicated interaction.

Despite of the vast amount of literature on these topics, one of the major research challenges lies in the efficiency: (1) for a static graph, given the desired approximation accuracy, we want to compute the example-based low-rank approximation with the least computational and space cost; and (2) for a dynamic graph[1], we

[1]In this paper, we use 'dynamic graphs' and 'time-evolving graphs' interchangeably.

want to monitor/track this approximation efficiently over time.

To deal with the above challenges, we propose the family of *Colibri* methods. Adjacency matrices for large graphs may contain near-duplicate columns. For example, all nodes that belong to the same closed and tightly-connected community would have the same sets of neighbors (namely, the community's members). CMD addresses the problem of duplicate elimination. However, even without duplicates, it is still possible that the columns of $\mathbf{L}$ are linearly dependent, leading to a redundant representation of the approximating subspace, which wastes both time and space. The main idea of our method for static graphs (*Colibri-S*) is to eliminate linearly dependent columns while iterating over sampled columns to construct the subspace used for low rank approximation. Formally, the approximation $\tilde{\mathbf{A}} = \mathbf{LMR}$ where $\mathbf{L}$ consists of judiciously selected columns, $\mathbf{M}$ is an incrementally maintained core matrix, and $\mathbf{R}$ is another small matrix. *Colibri-S* is provably better or equal compared to the best competitors in the literature, in terms of both speed and space cost, while it achieves the same approximation accuracy. In addition, we provide an analysis of the gains in terms of the redundancy present in the data. Furthermore, our experiments on real data show significant gains in practice. With the same approximation accuracy, *Colibri-S* is up to $52\times$ faster than the best known competitor, while it only requires about 1/3 of the space.

For dynamic graphs, we propose *Colibri-D*. Again, for the same accuracy, *Colibri-D* is provably better or equal compared to the best known methods (including our own *Colibri-S*) in terms of speed. The main idea of *Colibri-D* is to leverage the "smoothness", or similarity between two consecutive time steps, to quickly update the approximating subspace. Our experiments show that, with the same accuracy, *Colibri-D* achieves up to $112\times$ speedup over the best published competitor, and is 5 times faster than *Colibri-S* applied from scratch for each time step.

The main contributions of the paper are summarized as follows:

- A family of novel, low rank approximation methods (*Colibri-S*, *Colibri-D*) for static and dynamic graphs, respectively;
- Proofs, and complexity analysis, showing our methods are provably equal or better compared to the best known methods in the literature, for the same accuracy;
- Extensive experimental evaluation, showing that our methods are significantly faster, and nimbler than the top competitors. See Figure 1 for an example of the time and space savings of our *Colibri-S* over CUR and CMD [28].

The rest of the paper is organized as follows: after reviewing the related work in Section 2, we introduce notation and formally define the problems in Section 3. We present and analyze the proposed *Colibri-S* and *Colibri-D* in Section 4 and Section 5, respectively. We provide experimental evaluation in Section 6. Finally, we conclude in Section 7.

2. RELATED WORK

In this section, we briefly review the related work, which can be categorized into two parts: graph mining and matrix low rank approximation.

Graph Mining. There is a lot of research work on static graph mining, including pattern and law mining [2, 6, 10, 4, 20], frequent substructure discovery [30], influence propagation [18], community mining [11, 12, 13], detect anomaly nodes and edges[26], proximity[22, 29] and so on.

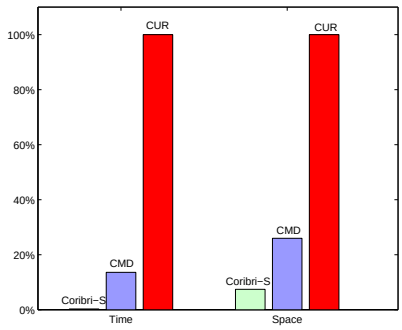


Figure 1: *Colibri-S* is significantly more efficient than both CUR and CMD in terms of both speed and space. Note that all these methods lead to the same approximation accuracy. Both speed and space cost are normalized by the most expensive one (i.e., CUR in both cases).

More recently, there is an increasing interest in mining time-evolving graphs, such as densification laws and shrinking diameters [19], community evolution [3], dynamic tensor analysis [27], and dynamic communities [5, 25], etc.

Low Rank Approximation. Low rank approximation [14, 8, 1] plays a very important role in graph mining. For example, the low rank approximation structure is often a good indicator to identify the community in the graph. A significant deviation from such structure often implies anomalies in the graph.

For static graphs, the most popular choices include SVD/PCA [14, 17] and random projection [16]. However, these methods often ignore the sparseness of many real graphs and therefore often need huge amount of space and processing time (See [28] for a detailed evaluation). More recently, Drineas et al [8] proposed the CUR decomposition, which partially deals with the sparsity of the graphs. CUR is proved to achieve an optimal approximation while maintain the sparsity of the matrix. Sun et al [28] further improve CUR by removing the duplicate columns/row in the sampling stage. Their method, named as CMD, is shown to produce the same approximation accuracy, but it often requires much less time and space. Our method (*Colibri-S*) further improves the efficiency in speed and space by leveraging the linear correlation among different sampled columns. As a result, our method saves the computational time and space cost, while it outputs exactly the same low rank approximation as CUR/CMD.

The worst-case computational complexity of CUR, CMD and *Colibri* is linear to the size of the matrix. A more accurate CUR approximation has been proposed in [9], but it requires SVD operation on the whole matrix as a preprocessing step which is often too expensive for many large scale applications.

For dynamic graphs, a lot of SVD based techniques have been proposed, such as multiple time series mining [15, 23], dynamic tensor analysis [27], incremental spectral clustering [21] etc. As for the static graphs, these methods might suffer from the loss-of-sparsity issue for large sparse graphs despite their success in the general cases. Sun et al [28] deal with this issue by applying their CMD method independently for each time step. However, how to make use of the smoothness between two consecutive time steps to do even more efficient computation is not exploited in [28]. This is exactly the unique feature of our *Colibri-D*, - it leverages such smoothness to do fast update while maintaining the sparseness of the resulting low rank approximation.

Table 1: Symbols

Symbol	Definition and Description
$\mathbf{A}, \mathbf{B}, \ldots$	matrices (bold upper case)
$\mathbf{A}(i,j)$	the element at the i^{th} row and j^{th} column of matrix $\mathbf{A}$
$\mathbf{A}(i,:)$	the i^{th} row of matrix $\mathbf{A}$
$\mathbf{A}(:,j)$	the j^{th} column of matrix $\mathbf{A}$
$\mathbf{A}'$	transpose of matrix $\mathbf{A}$
$\vec{a}, \vec{b}, \ldots$	column vectors
$\mathcal{I}, \mathcal{J}, \ldots$	sets (calligraphic)
$\mathbf{A}^{(t)}$	$n \times l$ time-aggregate interaction matrix at time t
$\vec{a}_j^{(t)}$	the j^{th} column of $\mathbf{A}^{(t)}$, i.e., $\vec{a}_j^{(t)} = \mathbf{A}^{(t)}(:,j)$
$\mathcal{I}$	indices for columns sampled: $\mathcal{I} = \{i_1, ..., i_c\}$
n, l	number of for type 1 and type 2 objects, respectively
c	sample size. i.e. the number of columns sampled
$\mathbf{C}_0^{(t)}$	$n \times c$ initial sampling matrix, consisting of c columns from $\mathbf{A}^{(t)}$. i.e., $\mathbf{C}_0^{(t)} = \mathbf{A}^{(t)}(:,\mathcal{I})$
$m_0^{(t)}$	number of edges in $\mathbf{C}_0^{(t)}$ at time t

3. PROBLEM DEFINITIONS

Table 3 lists the main symbols we use throughout the paper. In this paper, we consider the most general case of bipartite graphs. Uni-partite graph can be viewed as a special case. We represent a general bipartite graph by its adjacency matrix[2]. Following the standard notation, we use capital letters for matrices (e.g. $\mathbf{A}$), arrows for vectors (e.g. $\vec{a}_j$), and calligraphic fonts for sets (e.g. $\mathcal{I}$). We denote the transpose with a prime (i.e., $\mathbf{A}'$ is the transpose of $\mathbf{A}$), and we use parenthesized superscripts to denote time (e.g., $\mathbf{A}^{(t)}$ is the time-aggregate adjacency matrix at time t). When we refer to a static graph or, when time is clear from the context, we omit the superscript (t). We use subscripts to denote the size of matrices/vectors (e.g. $\mathbf{A}_{n\times l}$ means a matrix of size $n \times l$). Also, we represent the elements in a matrix using a convention similar to Matlab, e.g., $\mathbf{A}(\mathbf{i},\mathbf{j})$ is the element at the i^{th} row and j^{th} column of the matrix $\mathbf{A}$, and $\mathbf{A}(:,\mathbf{j})$ is the j^{th} column of $\mathbf{A}$, etc. With this notation, we can define matrix $\mathbf{C}_0$ as $\mathbf{C}_0 = \mathbf{A}(:,\mathcal{I}) = [\mathbf{A}(:,i_1), ..., \mathbf{A}(:,i_c)]$. In other words, $\mathbf{C}_0$ is the sub-matrix of $\mathbf{A}$ by stacking all its columns indexed by the set $\mathcal{I}$.Without loss of generality, we assume that the numbers of type 1 and type 2 objects (corresponding to rows and columns in the adjacency matrix) are fixed, i.e., n and l are constant for all time steps; if not, we can reserve rows/columns with zero elements as necessary.

At each time step, we observe a set of new edges, with associated edge weights. While there are multiple choices to update the adjacency matrix (e.g. sliding window, exponential forgetting etc), we use global aggregation for simplicity: once an edge appears at some time step t, the corresponding entry of the adjacency matrix is updated and the edge is never deleted or modified. This assumption facilitates presentation, but our methods can naturally apply to other update schemes.

With the above notations and assumptions, our problems can be formally defined as follows:

PROBLEM 1. **(Static Case.)** *Low rank approximation for static sparse graphs*

Given: *A large, static sparse graph $\mathbf{A}_{n\times l}$, and sample size c;*

Find: *Its low-rank approximation structure efficiently. That is, find three matrices $\mathbf{L}_{n\times\tilde{c}}, \mathbf{M}_{\tilde{c}\times\tilde{c}}$, and $\mathbf{R}_{\tilde{c}\times l}$ such that $\mathbf{A}_{n\times l} \approx \mathbf{L}_{n\times\tilde{c}}\mathbf{M}_{\tilde{c}\times\tilde{c}}\mathbf{R}_{\tilde{c}\times l}$, where $\tilde{c} \leq c$.*

[2]In practice, we store these matrices using an adjacency list representation, since real graphs are often very sparse.

PROBLEM 2. **(Dynamic Case.)** *Low rank approximation for dynamic sparse graphs*

Given: *A large, dynamic sparse graph $\mathbf{A}_{n\times l}^{(t)}$, for $t = 1, 2, \ldots$, and the sample size c;*

Track: *Its low-rank approximation structure over time efficiently. That is, to find three matrices $\mathbf{L}^{(t)}, \mathbf{M}^{(t)}$, and $\mathbf{R}^{(t)}$ for each time step t such that $\mathbf{A}_{n\times l}^{(t)} \approx \mathbf{L}_{n\times\tilde{c}^{(t)}}^{(t)}\mathbf{M}_{\tilde{c}^{(t)}\times\tilde{c}^{(t)}}^{(t)}\mathbf{R}_{\tilde{c}^{(t)}\times l}^{(t)}$, where $\tilde{c}^{(t)} \leq c$.*

4. *Colibri-S* FOR STATIC GRAPHS

In this section, we address problem 1 and introduce our *Colibri-S* for static graphs. After some necessary background in subsection 4.1, we present the algorithm in subsection 4.2, followed by the proofs and complexity analysis in subsection 4.3.

4.1 Preliminaries

Here, we want to decompose the adjacency matrix $\mathbf{A}_{n\times l}$ of a static graph into three matrices: $\mathbf{L}_{n\times\tilde{c}}$, $\mathbf{M}_{\tilde{c}\times\tilde{c}}$, and $\mathbf{R}_{\tilde{c}\times l}$. The goal is to achieve a good balance between efficiency and approximation quality. For the quality, we want $\tilde{\mathbf{A}} = \mathbf{LMR}$ to approximate the original adjacency matrix $\mathbf{A}$ as well as possible. Throughout the paper, we use the Frobenius norm of $\tilde{\mathbf{A}} - \mathbf{A}$ to measure the approximation error. As for efficiency, we want to (1) keep the matrices $\mathbf{L}$ and $\mathbf{R}$ small ($\tilde{c} \ll l$) and sparse, to save space; and (2) compute the decomposition using minimal running time.

The best known methods to achieve such balance are CUR [8] and its improved version, CMD [28]. The key idea behind CUR and CMD is to sample some columns of $\mathbf{A}$ with replacement, biased towards those with larger norms[3]; and then to use the projection of the original adjacency matrix $\mathbf{A}$ into the subspace spanned by these sampled columns as the low rank approximation of the matrix $\mathbf{A}$. As shown in [8], such procedures provably achieve an optimal approximation. Additionally, the matrices $\mathbf{L}$ and $\mathbf{R}$ by CUR/CMD are usually very sparse, thus the CUR/CMD decomposition is shown to be much faster than standard SVD.

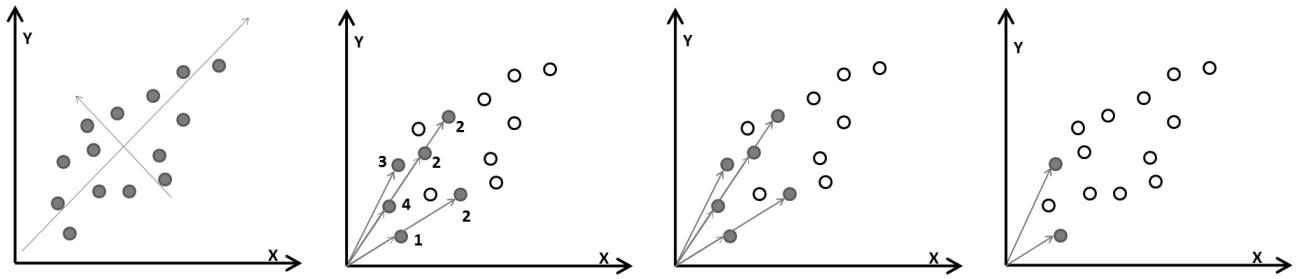
4.2 Algorithm

Our algorithm shares the same high-level principle as CUR and CMD. That is, we want to sample some columns of the matrix $\mathbf{A}$ and then project $\mathbf{A}$ into the subspace spanned by these columns. As we show later, our method achieves exactly the same approximation accuracy as CUR/CMD, but it is equal or better compared to CUR/CMD in terms of both space and time.

If we concatenate all the sampled columns into a matrix $\mathbf{C}_0$, we can use $\mathbf{C}_0(\mathbf{C}_0'\mathbf{C}_0)^\dagger\mathbf{C}_0'\mathbf{A}$ as the approximation of the original adjacency matrix $\mathbf{A}$, where $(\mathbf{C}_0'\mathbf{C}_0)^\dagger$ is the Moore-Penrose pseudo-inverse of the square matrix $\mathbf{C}_0'\mathbf{C}_0$.

However, the sampled columns in $\mathbf{C}_0$ may contain duplicates (or near duplicates)—for example, all nodes that belong to the same closed and tightly-connected community would have the same sets of neighbors (namely, the community's members). CMD essentially performs duplicate elimination. However, more generally, the columns of $\mathbf{C}_0$ may be unequal but linear dependences may still be present. In other words, the columns of $\mathbf{C}_0$ form a *redundant* or *overcomplete* basis. This is clearly not efficient in terms of space. Moreover, if we keep these redundant columns, we have

[3]In [8, 28], the authors also suggest simultaneously sampling columns and rows. Our method can be naturally generalized to handle this case. For simplicity, we focus on sampling columns only.



(a) SVD (b) CUR (c) CMD (d) *Colibri-S*

Figure 2: A pictorial comparison for different methods. To construct the same subspace, SVD will use all the data points (dark ones); CUR will use a subset of data point with possibly a lot duplications (the number besides the arrow is the number of duplicate copies); CMD will remove the duplicate the columns in CUR; and our *Colibri-S* will further remove all linearly dependent columns which is most efficient in both speed and space. For illustrative purpose, we set the approximation accuracy of each method to be always 100% in this example.

to estimate the pseudo-inverse of a larger matrix, which adversely affects running time as well.

The heart of *Colibri-S* is to iteratively construct the desired subspace, eliminate these redundant columns in the process. Algorithm 1 shows the full pseudocode.

Algorithm 1 *Colibri-S* for Static Graphs

Input: The adjacency matrix $\mathbf{A}_{n\times l}$, tolerance ϵ, and the sample size c

Output: Three matrices $\mathbf{L}_{n\times\tilde{c}}$, $\mathbf{M}_{\tilde{c}\times\tilde{c}}$, and $\mathbf{R}_{\tilde{c}\times l}$, where $\tilde{c}\le c$.

1: Compute column distribution for $x = 1, ..., l$: $P(x) = \sum_i \mathbf{A}(i,x)^2 / \sum_{i,j}\mathbf{A}(i,j)^2$;
2: Sample c columns from $\mathbf{A}$ based on $P(x)$. Let $\mathcal{I} = \{i_1, ..., i_c\}$ be the indices of these columns.
3: Initialize $\mathbf{L} = [\mathbf{A}(:,i_1)]$; $\mathbf{M} = 1/(\mathbf{A}(:,i_1)' \cdot \mathbf{A}(:,i_1))$
4: **for** $k = 2 : c$ **do**
5: Compute the residual: $r\vec{e}s = \mathbf{A}(:,i_k) - \mathbf{LML}'\mathbf{A}(:,i_k)$
6: **if** $\|r\vec{e}s\| \le \varepsilon\|\mathbf{A}(:,i_k)\|$ **then**
7: Continue;
8: **else**
9: Compute: $\delta = \|r\vec{e}s\|^2$; and $\vec{y} = \mathbf{ML}'\mathbf{A}(:,i_k)$
10: Update the core matrix $\mathbf{M}$: $\mathbf{M} \leftarrow \begin{pmatrix}\mathbf{M}+\vec{y}'\vec{y}/\delta & -\vec{y}/\delta \\ -\vec{y}'/\delta & 1/\delta\end{pmatrix}$
11: Expand $\mathbf{L}$: $\mathbf{L} \leftarrow [\mathbf{L},\ \mathbf{A}(:,i_k)]$
12: **end if**
13: **end for**
14: Compute $\mathbf{R} = \mathbf{L}'\mathbf{A}$.

There are three stages in algorithm 1. First (steps 1-2), we sample c columns of matrix $\mathbf{A}$ with replacement, biased towards those with higher norms, exactly as CUR does (first step in Figure. 3). Then, we try to select linearly independent columns from the initially sampled columns and build the $\mathbf{M}$ matrix (referred to as the "core matrix"): after an initialization step (step 3), we iteratively test if a new column $\mathbf{A}(:,i_k)$ is linearly dependent on the current columns of $\mathbf{L}$ (steps 5-7). If so, we skip the column $\mathbf{A}(:,i_k)$. Otherwise, we append $\mathbf{A}(:,i_k)$ into $\mathbf{L}$ and update the core matrix $\mathbf{M}$ (steps 9-11). Note that if the new column $\mathbf{A}(:,i_k)$ is linearly independent wrt the current columns in $\mathbf{L}$ (i.e., if $\|r\vec{e}s\| > \varepsilon\|\mathbf{A}(:,i_k)\|$), we can use the residual $r\vec{e}s$ computed in step 5 to update the core matrix $\mathbf{M}$ in step 9. Conversely, we use the core matrix $\mathbf{M}$ to estimate the residual and test linear dependence of the new column (step 5). In this way, we simultaneously prune the redundant columns and update the core matrix. The last step in Figure. 3 shows the final $\mathbf{L}$ obtained after eliminating the redundant columns from $\mathbf{C}_0$. Finally, we define the $\mathbf{R}$ matrix to be $\mathbf{L}'\mathbf{A}$.[4]

4.3 Proofs and Analysis

Here we provide the proofs and the performance analysis of *Colibri-S*. We also make a brief comparison with the state-of-art techniques, such as CUR/CMD.

4.3.1 *Proof of Correctness for* Colibri-S

We have the following theorem for the correctness of Alg. 1:

THEOREM 1. **Correctness of *Colibri-S*.** *Let the matrix $\mathbf{C}_0$ contain the initial sampled columns from $\mathbf{A}$(i.e. $\mathbf{C}_0 = \mathbf{A}(:,\mathcal{I})$). With tolerance $\epsilon = 0$, the following facts hold for the matrices $\mathbf{L}$ and $\mathbf{M}$ in Alg. 1:*

P1: the columns of $\mathbf{L}$ are linearly independent;
P2: $\mathbf{L}$ shares the same column space as $\mathbf{C}_0$;
P3: the core matrix $\mathbf{M}$ satisfies $\mathbf{M} = (\mathbf{L}'\mathbf{L})^{-1}$.

PROOF. First, we will prove 'P3' in Theorem 1 by induction. The base case (step 3 of Alg. 1) is obviously true.

For the induction step of 'P3', let us suppose that (1) $\mathbf{M} = (\mathbf{L}'\mathbf{L})^{-1}$ holds up to the $k_1^{th}(2 \le k_1 \le c)$ iteration; and (2) $\mathbf{L}$ will be expanded next in the k_2^{th} iteration ($k_1 < k_2 \le c$).

Let $\tilde{\mathbf{L}} = (\mathbf{L}\ \ \mathbf{A}(:,i_{k_2}))$. We have

$$\tilde{\mathbf{L}}'\tilde{\mathbf{L}} = \begin{pmatrix}\mathbf{L}' \\ \mathbf{A}(:,i_{k_2})'\end{pmatrix} \times \begin{pmatrix}\mathbf{L}\ \ \mathbf{A}(:,i_{k_2})\end{pmatrix} = \begin{pmatrix}\mathbf{L}'\mathbf{L} & \mathbf{L}'\mathbf{A}(:,i_{k_2}) \\ \mathbf{A}(:,i_{k_2})'\mathbf{L} & \mathbf{A}(:,i_{k_2})'\mathbf{A}(:,i_{k_2})\end{pmatrix} \quad (1)$$

Define $\tilde{\mathbf{M}} = \begin{pmatrix}\mathbf{M}+\vec{y}'\vec{y}/\delta & -\vec{y}/\delta \\ -\vec{y}'/\delta & 1/\delta\end{pmatrix}$, where $\vec{y}$ and δ are defined in Alg. 1.

[4]Note that while $\mathbf{L}$ is sparse since it consists of a subset of the original columns from $\mathbf{A}$, the matrix $\mathbf{R}$ is the multiplication of two sparse matrices and is not necessarily sparse. In order to further save space, we can use a randomized algorithm [7] to approximate $\mathbf{R}$. This can be naturally incorporated into Alg. 1. However, it is an orthogonal to what we are proposing in this paper. For simplicity, we will use $\mathbf{R} = \mathbf{L}'\mathbf{A}$ throughout this paper.

Since $\mathbf{M} = (\mathbf{L}'\mathbf{L})^{-1}$ by inductive hypothesis, it can be verified that $r\vec{e}s$ is the residual if we project the column $\mathbf{A}(:, i_{k_2})$ into the column space of $\mathbf{L}$. Based on the orthogonality property of the projection, we have

$$\begin{aligned} \delta &= \|r\vec{e}s\|^2 \\ &= r\vec{e}s'(r\vec{e}s + \mathbf{L}\mathbf{M}\mathbf{L}'\mathbf{A}(:, i_{k_2})) \\ &= r\vec{e}s'\mathbf{A}(:, i_k) \end{aligned} \quad (2)$$

Now, applying the Sherman-Morrison lemma [24] to the matrix $\tilde{\mathbf{L}}'\tilde{\mathbf{L}}$ in the form of eq. 1, based on eq. 2, we can verify that $\tilde{\mathbf{M}} = (\tilde{\mathbf{L}}'\tilde{\mathbf{L}})^{-1}$ holds, which completes the proof of 'P3'.

Next, let us prove 'P1' in Theorem 1 by induction. Again, the base case for 'P1' is obviously true (step 3 of Alg. 1).

For the induction step for 'P1', let us suppose that (1) all the columns in $\mathbf{L}_{n\times\hat{c}}$ are linearly independent up to the k_1^{th} iteration ($2 \le \hat{c} \le k_1 \le c$); and (2) $\mathbf{L}$ will be expanded next in the k_2^{th} iteration ($k_1 < k_2 \le c$). We only need to prove that $\mathbf{A}(:, i_{k_2}))$ is linear independent wrt the columns in the current $\mathbf{L}$ matrix.

By 'P3', the $r\vec{e}s$ computed in step 5 is the exactly the residual if we project the column $\mathbf{A}(:, i_{k_2}))$ into the column space spanned by the current $\mathbf{L}$ matrix. Since we decide to expand $\mathbf{L}$ by $\mathbf{A}(:, i_{k_2}))$, with tolerance $\epsilon = 0$, it must be true that the residual satisfies $\text{res} > 0$ (step 8). In other words, the column $\mathbf{A}(:, i_{k_2}))$ is not in the column space of $\mathbf{L}$.

Now, suppose that $\mathbf{A}(:, i_{k_2}))$ is linearly dependent to the columns in the current $\mathbf{L}$ matrix. The column $\mathbf{A}(:, i_{k_2}))$ must lie in the column space of $\mathbf{L}$. This is contra-positive, which completes the proof of 'P1'.

Finally, from 'P1', for each column $\vec{u} \in \{\mathbf{C_0} - \mathbf{L}\}$ (steps 5-7 of Alg. 1), there must exist a vector $\vec{\beta} = (\beta_1, ..., \beta_{\hat{c}})' = \mathbf{M}\mathbf{L}'\vec{u}$, such that $\vec{u} = \mathbf{L}\vec{\beta}$ holds. In other words, $\vec{u}$ must be in the column space of $\mathbf{L}$. Therefore, removing the column $\vec{u}$ from $\mathbf{L}$ will not change the column space of $\mathbf{L}$. This completes the proof of 'P2'. □

Notice that *Colibri-S* iteratively finds the linearly independent set of columns (i.e., the matrix $\mathbf{L}$). For the same initially sampled columns ($\mathbf{C}_0$), it might lead to a different $\mathbf{L}$ matrix if we use a different order in the index set $\mathcal{I}$. However, based on Theorem 1, this operation will not affect the subspace spanned by the columns of the matrix $\mathbf{L}$ since it is always the same as the subspace spanned by the columns of the matrix $\mathbf{C}_0$. Therefore, it will not affect the approximation accuracy for the original matrix $\mathbf{A}$.

4.3.2 *Efficiency of* Colibri-S

We have the following lemma for the speed of Alg. 1.

LEMMA 1. **Efficiency of *Colibri-S*.** *The computational complexity to output $\mathbf{M}$ and $\mathbf{L}$ in Alg. 1 is bounded by $O(c\tilde{c}^2 + c\tilde{m})$, where $\tilde{c}, \tilde{m}$ are the number of columns and edges in the matrix $\mathbf{L}$, respectively; and c is the number of columns in $\mathbf{C}_0$.*

PROOF. Omitted for brevity. □

4.3.3 Comparison with CUR/CMD

Next we compare *Colibri-S* against the state-of-art techniques, i.e. CUR [8] and CMD [28]. We compare with respect to accuracy, time and space cost.

LEMMA 2 (ACCURACY). *Using the same initial sampled columns $\mathbf{C}_0$, Alg. 1 has exactly the same approximation accuracy as CUR [8] and CMD [28].*

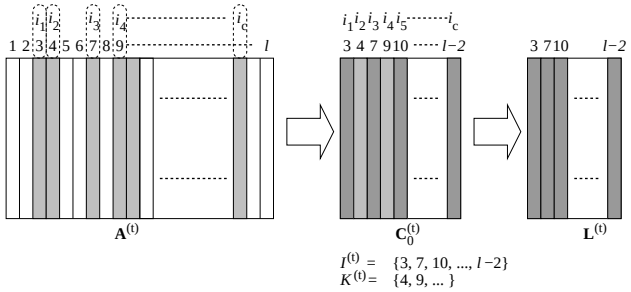


Figure 3: Illustration of notation and process for *Colibri-S*. Shaded columns are part of initial sample, dark shaded columns are linearly independent among those.

PROOF. Define $\tilde{\mathbf{A}}$ as $\tilde{\mathbf{A}} = \mathbf{L}\mathbf{M}\mathbf{R}$. By Theorem 1, the matrix $\tilde{\mathbf{A}}$ satisfies $\tilde{\mathbf{A}} = \mathbf{L}(\mathbf{L}'\mathbf{L})^{-1}\mathbf{L}'\mathbf{A}$. In other words, $\tilde{\mathbf{A}}$ is the projection of the matrix $\mathbf{A}$ into the column space of $\mathbf{L}$. On the other hand, by Theorem 1, the matrix $\mathbf{L}$ has the same column space as $\mathbf{C}_0$, i.e., $\tilde{\mathbf{A}} = \mathbf{C}_0(\mathbf{C}_0'\mathbf{C}_0)^{\dagger}\mathbf{C}_0'\mathbf{A}$, which is exactly how CUR/CMD [8, 28] tries to approximate the original matrix $\mathbf{A}$. □

LEMMA 3 (SPACE). *Using the same initial sampled columns $\mathbf{C}_0$, Alg. 1 is better than or equal to CUR in [8] and CMD in [28] in terms of space.*

PROOF. Notice that $\mathbf{L}$ is always a subset of $\mathbf{C}_0$. On the other hand, if there exist duplicate columns in $\mathbf{C}_0$, they will appear only once in $\mathbf{L}$. □

LEMMA 4 (TIME). *Using the same initial sampled columns $\mathbf{C}_0$, Alg. 1 is faster than, or equal to CUR ([8]) and CMD ([28]).*

PROOF. By Lemma 1, the computational complexity of Alg. 1 at the worst case is the same as the original CUR method in [8] ($O(cm)$ for multiplying $\mathbf{C}_0'$ and $\mathbf{C}_0$ together; and $O(c^3)$ for the Moore-Penrose pseudo-inverse of $\mathbf{C}_0'\mathbf{C}_0$. Also notice that $\tilde{c} \le c$ and $\tilde{m} \le m$). On the other hand, if there exist duplicate columns in $\mathbf{C}_0$, we can always remove them before step 3 in Alg.1 and then CMD in [28] will degenerate to CUR [8]. □

In particular, the complexity is proportional to the square of the "true" dimensionality $\tilde{c}$ of the approximating subspace. Since, as we shall see in the experimental evaluation, in real datasets $\tilde{c}$ is significantly smaller than c, this translates to substantial savings in computation time as well as space.

INTUITION. The intuition behind the above proofs and savings is shown in Figure 2, which gives a pictorial comparison of our *Colibri-S* with SVD/CUR/CMD. Figure 2 shows that: (1) SVD (Figure 2(a)) uses all data points (dark ones) and the resulting $\mathbf{L}$ matrix is dense. (2) CUR (Figure 2(b)) uses sampled columns (dark ones) but there may be many duplicate columns (the number next to each arrow stands for the multiplicity) The resulting $\mathbf{L}$ matrix of CUR is sparse but it has totally 16 columns. (3) CMD (Figure 2(c)) removes the duplicate columns in CUR and the resulting $\mathbf{L}$ (with 6 columns) is more compact. (4) Our *Colibri-S* (Figure 2(d)) further removes all the linearly dependent columns and the resulting $\mathbf{L}$ only contains 2 sparse columns. Therefore, while all these four methods leads to the same subspace, *Colibri-S* is most efficient in both time and space.

5. *Colibri-D* FOR DYNAMIC GRAPHS

In this section, we deal with problem 2 and propose *Colibri-D* for dynamic, time-evolving graphs. Our goal is to find the low rank approximation structure of the adjacency matrix at each time step t efficiently. As for static graphs, we first give the algorithm in subsection 5.1 and then provide theoretical justification and analysis in subsection 5.2.

5.1 Algorithm

Conceptually, we could call Alg. 1 to output the low rank approximation for each time step t. In this way, we will have to compute the core matrix $\mathbf{M}$, which is the most expensive part in Alg. 1, for each time step from the scratch. On the other hand, if the graph changes "smoothly" between two consecutive time steps (i.e., the number of affected edges is reasonably small) then, intuitively, we do not expect its low rank approximation structure to change dramatically. This is exactly the heart of our *Colibri-D*. We want to leverage the core matrix $\mathbf{M}^{(t)}$ to quickly get the core matrix $\mathbf{M}^{(t+1)}$ in the next time step, given that the graph changes "smoothly" from time step t to $(t+1)$.

For simplicity, we assume that the indices of the initial sampled columns $\mathbf{C}_0^{(t)}$ are fixed. That is, we will fix the index set $\mathcal{I} = \{i_1, ..., i_c\}$ over time, and we will always use the projection of the adjacency matrix $\mathbf{A}^{(t)}$ in the columns space of $\mathbf{C}_0^{(t)} = \mathbf{A}^{(t)}(:, \mathcal{I})$ as the low rank approximation of $\mathbf{A}^{(t)}$ for each time step[5]. Note that even if we use the same initial column indices, the content of matrix $\mathbf{C}_0^{(t)}$ keeps changing over time and so does the subspace it spans. Our goal is to efficiently update the non-redundant basis for the subspace spanned by the columns of $\mathbf{C}_0^{(t)}$ over time. Note that in Figure. 4, the column indices of $\mathbf{C}_0^{(t+1)}$ are exactly the same as those for $\mathbf{C}_0^{(t)}$ in Figure. 3. However, in this example, the contents of columns 3 and $l-2$ have changed.

The basic idea of our algorithm for dynamic graphs is as follows: once the adjacency matrix $\mathbf{A}^{(t+1)}$ at time step $(t+1)$ is updated, we will update the matrix $\mathbf{C}_0^{(t+1)}$. Then, we will try to identify those linearly independent columns $\mathbf{L}^{(t+1)}$ within $\mathbf{C}_0^{(t+1)}$ as well as the core matrix $\mathbf{M}^{(t+1)}$. To reduce the computational cost, we will leverage the core matrix from the current time step $\mathbf{M}^{(t)}$ to update $\mathbf{L}^{(t+1)}$ as well as $\mathbf{M}^{(t+1)}$, instead of computing them from the scratch. Finally, we will update the $\mathbf{R}$ matrix as $\mathbf{R}^{(t+1)} = \mathbf{L}^{(t+1)'}\mathbf{A}^{(t+1)}$.

Next, we will describe how to update $\mathbf{L}^{(t+1)}$ and $\mathbf{M}^{(t+1)}$ at time step $t+1$. At time step t, we might find some redundant columns in $\mathbf{C}_0^{(t)}$ which are linearly dependent wrt the remaining columns in $\mathbf{C}_0^{(t)}$. In Figure. 3, these were columns 4 and 9. We split the indices set $\mathcal{I}$ into two disjoint subsets: $\mathcal{J}^{(t)}$ and $\mathcal{K}^{(t)}$, as shown in Figure. 3. We require that $\mathcal{I} = \mathcal{J}^{(t)} \cup \mathcal{K}^{(t)}$, and $\mathbf{L}^{(t)} = \mathbf{A}^{(t)}(:, \mathcal{J}^{(t)})$. In other words, $\mathcal{J}^{(t)}$ corresponds to those columns in $\mathbf{C}_0^{(t)}$ that are actually used to construct the subspace; and $\mathcal{K}^{(t)}$ corresponds to those redundant columns in $\mathbf{C}_0^{(t)}$. Notice that even though we fix the index set $\mathcal{I}$ over time, the subsets $\mathcal{J}^{(t)}$ and $\mathcal{K}^{(t)}$ change over time. Updating the matrix $\mathbf{L}^{(t)}$ is equivalent to updating the subset $\mathcal{J}^{(t)}$. To simplify the description of the algorithm, we further partition $\mathcal{J}^{(t)}$ into two disjoint subsets $\mathcal{J}_a^{(t)}$ and $\mathcal{J}_b^{(t)}$, such that $\mathcal{J}^{(t)} = \mathcal{J}_a^{(t)} \cup \mathcal{J}_b^{(t)}$. We require that $\mathbf{A}^{(t)}(:, \mathcal{J}_a^{(t)}) = \mathbf{A}^{(t+1)}(:, \mathcal{J}_a^{(t)})$; and $\mathbf{A}^{(t)}(:, \mathcal{J}_b^{(t)}) \neq \mathbf{A}^{(t+1)}(:, \mathcal{J}_b^{(t)})$. In other words, $\mathcal{J}_a^{(t)}$ corresponds to those unchanged columns in $\mathbf{L}$ from t to $(t+1)$, while $\mathcal{J}_b^{(t)}$ corresponds to those changed columns from t to $(t+1)$. These sets are shown in Figure. 4 on the left: notice that their union is $\mathcal{I}^{(t)}$ from Figure. 3.

With the above notations, the complete pseudocode to update the low rank approximation from time step t to $(t+1)$ is given in Alg. 2.

Algorithm 2 *Colibri-D* for Dynamic Graphs

Input: The adjacency matrices $\mathbf{A}^{(t)}$ and $\mathbf{A}^{(t+1)}$, the indices set $\mathcal{I} = \mathcal{J}^{(t)} \cup \mathcal{K}^{(t)}$, tolerance ϵ, and the core matrix $\mathbf{M}^{(t)}$ at time step t

Output: Three matrices $\mathbf{L}^{(t+1)}$, $\mathbf{M}^{(t+1)}$, and $\mathbf{R}^{(t+1)}$; and updated indices partition $\mathcal{I} = \mathcal{J}^{(t+1)} \cup \mathcal{K}^{(t+1)}$.

1: Set $\mathcal{J}_a^{(t)}$ and $\mathcal{J}_b^{(t)}$ based on $\mathbf{A}^{(t)}$ and $\mathbf{A}^{(t+1)}$;
2: Initialize $\mathbf{L}^{(t+1)} = \mathbf{A}(:, \mathcal{J}_a^{(t)})$; $\mathcal{K} = \mathcal{J}_b^{(t)} \cup \mathcal{K}^{(t)}$
3: **if** $\mid \mathcal{J}_a^{(t)} \mid \leq \mid \mathcal{J}_b^{(t)} \mid$ **then**
4: Compute: $\mathbf{M}^{(t+1)} = (\mathbf{L}^{(t+1)'}\mathbf{L}^{(t+1)})^{-1}$
5: **else**
6: Compute: $\mathbf{\Lambda} = \mathbf{M}^{(t)}(\mathcal{J}_b^{(t)}, \mathcal{J}_b^{(t)})^{-1}$
7: Compute: $\mathbf{\Delta} = \mathbf{M}^{(t)}(\mathcal{J}_a^{(t)}, \mathcal{J}_b^{(t)})\mathbf{\Lambda}\mathbf{M}^{(t)}(\mathcal{J}_b^{(t)}, \mathcal{J}_a^{(t)})$
8: Compute: $\mathbf{M}^{(t+1)} = \mathbf{M}^{(t)}(\mathcal{J}_a^{(t)}, \mathcal{J}_a^{(t)}) - \mathbf{\Delta}$
9: **end if**
10: **for** each index k in $\mathcal{K}$ **do**
11: Compute the residual: $r\vec{e}s = \mathbf{A}^{(t+1)}(:,k) - \mathbf{L}^{(t+1)}\mathbf{M}^{(t+1)}\mathbf{L}^{(t+1)'}\mathbf{A}^{(t+1)}(:,k)$
12: **if** $\|r\vec{e}s\| \leq \varepsilon\|\mathbf{A}^{(t+1)}(:,k)\|$ **then**
13: Continue;
14: **else**
15: Compute: $\delta = \|r\vec{e}s\|^2$; and $\vec{y} = \mathbf{M}^{(t+1)}\mathbf{L}^{(t+1)'}\mathbf{A}^{(t+1)}(:,k)$
16: Update the core matrix $\mathbf{M}^{(t+1)}$: $\mathbf{M}^{(t+1)} \leftarrow \begin{pmatrix} \mathbf{M}^{(t+1)} + \vec{y}'\vec{y}/\delta & -\vec{y}/\delta \\ -\vec{y}'/\delta & 1/\delta \end{pmatrix}$
17: Expand $\mathbf{L}^{(t+1)}$: $\mathbf{L}^{(t+1)} \leftarrow [\mathbf{L}^{(t+1)},\ \mathbf{A}^{(t+1)}(:,k)]$
18: **end if**
19: **end for**
20: Compute $\mathbf{R}^{(t+1)} = \mathbf{L}^{(t+1)'}\mathbf{A}^{(t+1)}$;
21: Update $\mathcal{J}^{(t+1)}$ and $\mathcal{K}^{(t+1)}$.

Comparing Alg. 2 with its static version (Alg. 1), the main differences are (1) we do not need to test the linear dependence and build our core matrix from the scratch if the subset $\mathcal{J}_a$ is not empty (steps 3–9), since the columns in $\mathcal{J}_a$ are guaranteed to be linearly independent; (2) furthermore, if the change in $\mathcal{I}$ is relatively small (i.e. $\mid \mathcal{J}_a^{(t)} \mid > \mid \mathcal{J}_b^{(t)} \mid$), we do not need to initialize our core matrix $\mathbf{M}^{(t+1)}$ from the scratch. Instead, we can leverage the information in $\mathbf{M}^{(t)}$ to do fast initialization (steps 6–8). These strategies, as will be shown in the next subsection, will dramatically reduce the computational time, while the whole algorithm will give exactly the same low rank approximation as if we had called Alg. 1 for time step $(t+1)$. After we initialize the core matrix $\mathbf{M}^{(t+1)}$ (after step 9), we will recursively test the linear dependence for each column in $\mathcal{K}^{(t)}$ and $\mathcal{J}_b^{(t)}$ and possibly incorporate them to expand the core matrix $\mathbf{M}^{(t+1)}$, which is very similar to what we do for the static graphs in Alg. 1.

In our running example of Figure. 3 and 4, since columns 7 and 10 were linearly independent at time t and they have remained unchanged, we can safely initialize $\mathbf{L}^{(t+1)}$ to include these. However, since columns 3 and $l-2$ have changed, we need to re-test for linear independence. In this example, it turns out that 3 is still linearly

[5]How to update the indices set $\mathcal{I}$ over time is beyond the scope of this paper.

independent, whereas $l-2$ is not any more. Additionally, some of the columns that were previously excluded as linearly dependent (e.g., 4 and 9) may now have become linearly independent, so we need to re-test those as well. In this example, it turns out that they are still redundant.

5.2 Proofs and Analysis

5.2.1 *Correctness of* Colibri-D

We have the following lemma for the correctness of Alg. 2:

LEMMA 5. **Correctness of *Colibri-D*.** *Let the matrix $\mathbf{C}_0$ contain the initial sampled columns from $\mathbf{A}^{(t+1)}$ (i.e. $\mathbf{C}_0 = \mathbf{A}^{(t+1)}(:, \mathcal{I})$). With tolerance $\epsilon = 0$, the following facts hold for the matrices $\mathbf{L}^{(t+1)}$ and $\mathbf{M}^{(t+1)}$ in Alg. 2:*

P1: the columns of $\mathbf{L}^{(t+1)}$ are linearly independent;
P2: $\mathbf{L}^{(t+1)}$ shares the same column space as $\mathbf{C}_0$;
P3: the core matrix $\mathbf{M}^{(t+1)}$ satisfies $\mathbf{M}^{(t+1)} = (\mathbf{L}^{(t+1)'}\mathbf{L}^{(t+1)})^{-1}$.

PROOF. **:** Similar as for Theorem 1. Omitted for brevity □

By Lemma 5 and Theorem 1, the three matrices $\mathbf{L}^{(t+1)}$, $\mathbf{M}^{(t+1)}$, and $\mathbf{R}^{(t+1)}$ produced by Alg. 2 are exactly the same as if we had called Alg. 1 for time step $(t+1)$ from the scratch. Therefore, we have the following corollary:

COROLLARY 2. *Using the same index set $\mathcal{I}$ of initial sampled columns for all time steps, Alg. 2 has exactly the same approximation accuracy as Alg. 1, CUR [8] and CMD [28].*

5.2.2 *Efficiency of* Colibri-D

Since the three matrices $\mathbf{L}^{(t+1)}$, $\mathbf{M}^{(t+1)}$, and $\mathbf{R}^{(t+1)}$ by Alg. 2 are exactly the same as if we had called Alg. 1 for time step $(t+1)$, we have the following corollary for the space cost of Alg. 2:

COROLLARY 3. *Using a fixed indices set $\mathcal{I}$ of initial sampled columns, the space cost of Alg. 2 is the same as Alg. 1 and it is equal or better compared to CUR [8] and CMD [28].*

We have the following lemma about the speed of Alg. 2.

LEMMA 6. **Efficiency of *Colibri-D*.** *Let $r_1 = |\ \mathcal{J}_a^{(t)}\ |$, $r_2 = |\ \mathcal{J}_b^{(t)}\ |$ and $r_3 = |\ \mathcal{K}^{(t)}\ |$. The computational complexity of Alg. 2 is bounded by $O(\max(r_1, r_2, r_3)^3 + (r_2 + r_3)\tilde{m}^{(t+1)})$, where $\tilde{m}^{(t+1)}$ is number of edges in the matrix $\mathbf{L}^{(t+1)}$.*

PROOF. **:** Omitted for brevity. □

In terms of speed, the difference between Alg. 2 and Alg. 1 lies in the different way of initializing the matrix $\mathbf{M}^{(t+1)}$ (steps 3–9 of Alg. 2). More specifically, if $r_1 \leq r_2$, the computational cost for initializing $\mathbf{M}^{(t+1)}$ is asymptotically the same for both algorithms—both are $O(r_1^3)$. On the other hand, if $r_1 > r_2$, we only need $O(r_1^2 r_2)$ for Alg. 2 while Alg. 1 still requires $O(r_1^3)$. Based on this fact as well as Lemma 1, we have the following corollary.

COROLLARY 4. *Using a fixed set $\mathcal{I}$ of initial sampled columns, the running time of Alg. 2 is equal or better compared to Alg. 1, CUR [8] and CMD [28].*

To summarize, if we fix the index set $\mathcal{I}$ of initial sampled columns for all time steps, the proposed Alg. 2 will produce the low rank approximation at each time step t with the same accuracy as CUR/CMD and our own Alg. 1 for static graphs. For both speed and space cost, it is always equal or better than CUR/CMD as well as our Alg. 1.

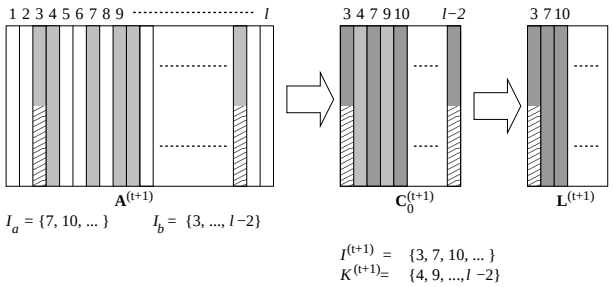


Figure 4: Illustration of notation and process for *Colibri-D*—compare with Figure. 3. Shaded and dark shaded columns as in Figure. 3, shaded and filled columns are those from the previous timestep that contain at least one new entry.

6. EXPERIMENTAL EVALUATIONS

Here we give experimental results for the proposed *Colibri*. Our evaluation mainly focuses on (1) the reconstruction accuracy, (2) the running time and (3) the space cost. After a brief introduction of the datasets and the evaluation criteria, we give the results for *Colibri-S* in subsection 6.2, and for *Colibri-D* in subsection 6.3.

6.1 Experimental Setup

We use a network traffic dataset from the backbone router of a class-B university network. We create a traffic matrix for every hour, with the rows and columns corresponding to the IP sources and IP destinations. We turn the matrix into a binary matrix, that is, a '1' entry means that there is some TCP flow from the corresponding IP source to the destination within that hour. In short, we ignore the volume of such traffic. Overall there are 21,837 different source/destination pairs, 1,222 consecutive hours and 22.8K edges per hour, on average.

Let $\tilde{\mathbf{A}} = \mathbf{LMR}$. We use the standard reconstruction accuracy to measure the approximation quality (exactly as in [28]), to estimate the SSE, the sum-squared-error, with sample size c=1,000 for both rows and columns:

$$\begin{aligned} \text{Accu} &= 1 - \text{SSE} \\ &= 1 - \sum_{i,j}(\mathbf{A}(i,j) - \tilde{\mathbf{A}}(i,j))^2 / (\sum_{i,j}\mathbf{A}(i,j)^2) \quad (3) \end{aligned}$$

For a given low rank approximation $\{\mathbf{L}_{n\times\tilde{c}},\ \mathbf{M}_{\tilde{c}\times\tilde{c}},\ \mathbf{R}_{\tilde{c}\times l}\}$, the matrices $\mathbf{L}$ and $\mathbf{R}$ are usually sparse, and thus we store them as adjacency lists. In contrast, the matrix $\mathbf{M}$ is usually dense, and we store it as a full matrix. Thus, the space cost is:

$$\text{SPCost} = \text{NNZ}(\mathbf{L}) + \text{NNZ}(\mathbf{R}) + \tilde{c}^2 \quad (4)$$

where $\text{NNZ}(.)$ is the number of non-zero entries in the matrix.

For the computational cost, we report the wall-clock time. All the experiments ran on the same machine with four 2.4GHz AMD CPUs and 48GB memory, running Linux (2.6 kernel). For each experiment, we run it 10 times and report the average.

Notice that for both Theorem 1 and Lemma 5, we require the tolerance $\varepsilon = 0$. In our experiments, we find by changing ε to be a small positive number (e.g., $\varepsilon = 10^{-6}$), it does not influence the approximation accuracy (up to 4 digits precision), while it makes the proposed algorithms more numerically stable[6]. Therefore, for

[6]this is an implementation detail. We omit the detailed discussion due to the space limit. How to choose an optimal ε is out-of the scope of this paper.

all the experiments we reported in this paper, we use $\varepsilon = 10^{-6}$ for both *Colibri-S* and *Colibri-D*.

6.2 Performance of *Colibri-S*

Here, we evaluate the performance of our *Colibri-S* for static graphs, in terms of speed and space.

We compare *Colibri-S* against the best published techniques, and specifically against CUR [8] and CMD [28]. For brevity and clarity, we omit the comparison against SVD, because CMD [28] was reported to be significantly faster and nimbler than SVD, with savings up to 100 times.

We aggregate the traffic matrices within the first 100 hours and then ignore the edge weights as the target matrix $\mathbf{A}$. Totally, there are 158,805 edges in this graph. We vary the sample size c from 1,000 to 8,000, and study how the accuracy changes with the running time and space cost for all three methods.

Figure 5 plots the mean running time vs. the approximation accuracy. Notice that the y-axes is in the logarithm scale. *Colibri-S* is significantly faster than both CUR and CMD, by 28x$\sim$353x and 12x$\sim$52x respectively.

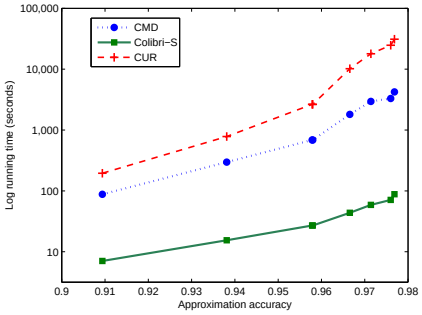


Figure 5: Running time vs. accuracy. Our *Colibri-S* (in green squares) is significantly faster than both CUR and CMD, for the same approximation accuracy. Note that the y-axis is in logarithmic scale.

With respect to space cost, CUR is always the most expensive among the three methods and therefore we use it as the baseline. Figure 6 plots the relative space cost of CMD and *Colibri-S*, vs. the approximation accuracy. Again, *Colibri-S* outperforms both CUR and CMD. Overall, *Colibri-S* only requires 7.4%$\sim$28.6% space cost of CUR, and 28.6%$\sim$59.1% space cost of CMD for the same approximation accuracy.

The reader may be wondering what causes all these savings. The answer is the reduction in columns kept: in *Colibri-S* we only keep those linearly independent columns, and discard all the other of the c columns that CUR chooses (and keeps). This idea eventually leads to significant savings. For example, with a sample size of $c = 8,000$ (the number of columns that CUR will keep), CMD discards duplicates, keeping on the average only $3,220$ unique columns, and *Colibri-S* further discards the linearly dependent ones, eventually keeping only $1,101$. And, thanks to our Theorem 1, the columns that *Colibri-S* discards have no effect on the desired subspace, and neither on the approximation quality.

6.3 Performance of *Colibri-D*

We use the same aggregated traffic matrix as in subsection 6.2; and initialize the algorithm by a sample size $c = 2,000$ (which gives an average accuacy of 93.8%). Then, we randomly perturb

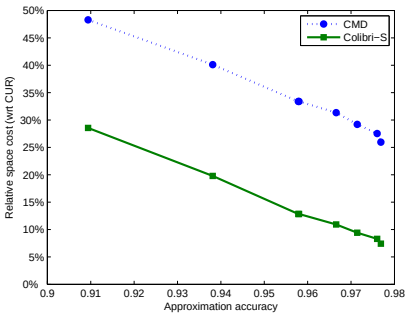


Figure 6: Relative space cost of *Colibri-S* and CMD, versus accuracy. Space costs are normalized by the space of CUR. *Colibri-S* consistently requires a fraction of the space by CUR/CMD, for same accuracy.

r out of these 2,000 sampled columns and update the low rank approximation of the updated adjacency matrix. Since *Colibri-D* has the same space cost as *Colibri-S*, we only present the results on the running time.

We compare our *Colibri-D* against both CMD and against our own *Colibri-S* We apply CMD and *Colibri-S* for each (static) instance of the graph and report the wall-clock times. For visual clarity, we omit the comparison against CUR, since it is consistently slower than both CMD and *Colibri-S* on static graphs, as shown in subsection 6.2.

Figure 7 plots the wall-clock time of CMD, *Colibri-S* and *Colibri-D*, versus r (the number of updated columns). *Colibri-D* is 2.5x$\sim$112x faster than CMD. Even compared against our own *Colibri-S* *Colibri-D* is still about 2x$\sim$5x faster. The computational savings of *Colibri-D* over *Colibri-S* come from the Sherman-Morrison Lemma: if the graph evolves smoothly, *Colibri-D* leverages the low rank approximation of the previous time step, and does a fast (but exact) update. We repeat that all three methods have *identical* approximation accuracy, if they use the same initial sampled columns.

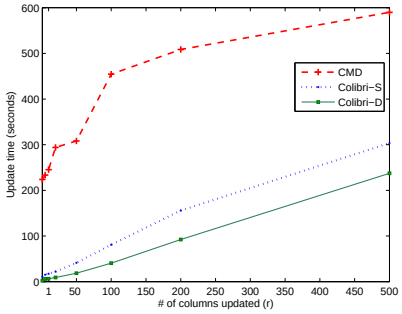


Figure 7: Performance for dynamic graphs: Speed versus number of updated columns. *Colibri-D* (in green squares) is 2.5x$\sim$112x faster than the best published competitor (CMD); and also faster than our own *Colibri-S*, applied on each individual graph instance.

7. CONCLUSION

In this paper, we propose the family of *Colibri* methods to do fast mining on large static and dynamic graphs. The main contributions of the paper are:

- A family of novel, low rank approximation methods (*Colibri-S*, *Colibri-D*) for static and dynamic graphs, respectively: *Colibri-S* saves space and time by eliminating linearly dependent columns; *Colibri-D* builds on *Colibri-S*, and performs incremental updates efficiently, by exploiting the "smoothness" between two consecutive time steps.
- Proofs and complexity analysis, showing our methods are provably equal or better compared to the best known methods in the literature, while maintaining exactly the same accuracy;
- Extensive experimental evaluation, showing that our methods are significantly faster and nimbler than the state of the art (up to 112 times faster). See Figure 1 for comparisons against CUR [8] and CMD [28].

8. ACKNOWLEDGEMENT

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 UCRL-CONF-231426. 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.

9. REFERENCES

[1] D. Achlioptas and F. McSherry. Fast computation of low-rank matrix approximations. *J. ACM*, 54(2), 2007.

[2] R. Albert, H. Jeong, and A.-L. Barabasi. Diameter of the world wide web. *Nature*, (401):130–131, 1999.

[3] 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.

[4] A. Broder, R. Kumar, F. Maghoul1, P. Raghavan, S. Rajagopalan, R. Stata, A. Tomkins, and J. Wiener. Graph structure in the web: experiments and models. In *WWW Conf.*, 2000.

[5] Y. Chi, X. Song, D. Zhou, K. Hino, and B. L. Tseng. Evolutionary spectral clustering by incorporating temporal smoothness. In *KDD*, pages 153–162, 2007.

[6] S. Dorogovtsev and J. Mendes. Evolution of networks. *Advances in Physics*, 51:1079–1187, 2002.

[7] P. Drineas, R. Kannan, and M. W. Mahoney. Fast monte carlo algorithms for matrices i: Approximating matrix multiplication. *SIAM Journal of Computing*, 2005.

[8] P. Drineas, R. Kannan, and M. W. Mahoney. Fast monte carlo algorithms for matrices iii: Computing a compressed approximate matrix decomposition. *SIAM Journal of Computing*, 2005.

[9] P. Drineas, M. W. Mahoney, and S. Muthukrishnan. Relative-error cur matrix decompositions. *CoRR*, abs/0708.3696, 2007.

[10] M. Faloutsos, P. Faloutsos, and C. Faloutsos. On power-law relationships of the internet topology. *SIGCOMM*, pages 251–262, Aug-Sept. 1999.

[11] G. Flake, S. Lawrence, C. L. Giles, and F. Coetzee. Self-organization and identification of web communities. *IEEE Computer*, 35(3), Mar. 2002.

[12] D. Gibson, J. Kleinberg, and P. Raghavan. Inferring web communities from link topology. In *Ninth ACM Conference on Hypertext and Hypermedia*, pages 225–234, New York, 1998.

[13] M. Girvan and M. E. J. Newman. Community structure is social and biological networks.

[14] G. H. Golub and C. F. Van-Loan. *Matrix Computations*. The Johns Hopkins University Press, Baltimore, 2nd edition, 1989.

[15] S. Guha, D. Gunopulos, and N. Koudas. Correlating synchronous and asynchronous data streams. In *KDD*, pages 529–534, 2003.

[16] P. Indyk. Stable distributions, pseudorandom generators, embeddings and data stream computation. In *FOCS*, pages 189–197, 2000.

[17] K. V. R. Kanth, D. Agrawal, and A. K. Singh. Dimensionality reduction for similarity searching in dynamic databases. In *SIGMOD Conference*, pages 166–176, 1998.

[18] D. Kempe, J. Kleinberg, and E. Tardos. Maximizing the spread of influence through a social network. *KDD*, 2003.

[19] J. Leskovec, J. M. Kleinberg, and C. Faloutsos. Graphs over time: densification laws, shrinking diameters and possible explanations. In *KDD*, pages 177–187, 2005.

[20] M. E. J. Newman. The structure and function of complex networks. *SIAM Review*, 45:167–256, 2003.

[21] H. Ning, W. Xu, Y. Chi, Y. Gong, and T. S. Huang. Incremental spectral clustering with application to monitoring of evolving blog communities. In *SDM*, 2007.

[22] J.-Y. Pan, H.-J. Yang, C. Faloutsos, and P. Duygulu. Automatic multimedia cross-modal correlation discovery. In *KDD*, pages 653–658, 2004.

[23] S. Papadimitriou, J. Sun, and C. Faloutsos. Streaming pattern discovery in multiple time-series. In *VLDB*, pages 697–708, 2005.

[24] W. Piegorsch and G. E. Casella. Inverting a sum of matrices. In *SIAM Review*, volume 32, pages 470–470, 1990.

[25] J. Sun, C. Faloutsos, S. Papadimitriou, and P. S. Yu. Graphscope: parameter-free mining of large time-evolving graphs. In *KDD*, pages 687–696, 2007.

[26] J. Sun, H. Qu, D. Chakrabarti, and C. Faloutsos. Neighborhood formation and anomaly detection in bipartite graphs. In *ICDM*, pages 418–425, 2005.

[27] J. Sun, D. Tao, and C. Faloutsos. Beyond streams and graphs: dynamic tensor analysis. In *KDD*, pages 374–383, 2006.

[28] J. Sun, Y. Xie, H. Zhang, and C. Faloutsos. Less is more: Compact matrix decomposition for large sparse graphs. In *SDM*, 2007.

[29] H. Tong, C. Faloutsos, and J.-Y. Pan. Random walk with restart: Fast solutions and applications. *Knowledge and Information Systems: An International Journal (KAIS)*, 2008.

[30] D. Xin, J. Han, X. Yan, and H. Cheng. Mining compressed frequent-pattern sets. In *VLDB*, pages 709–720, 2005.