CLUDE: An Efficient Algorithm for LU Decomposition Over a Sequence of Evolving Graphs

Chenghui Ren, Luyi Mo, Ben Kao, Reynold Cheng, David W. Cheung
Department of Computer Science, University of Hong Kong
Pokfulam Road, Hong Kong
{chren, lymo, kao, ckcheng, dcheung}@cs.hku.hk

ABSTRACT

In many applications, entities and their relationships are represented by graphs. Examples include the WWW (web pages and hyperlinks) and bibliographic networks (authors and co-authorship). A graph can be conveniently modeled by a matrix from which various quantitative measures are derived. Some example measures include PageRank and SALSA (which measure nodes' importance), and Personalized PageRank and Random Walk with Restart (which measure proximities between nodes). To compute these measures, linear systems of the form $\boldsymbol{Ax} = \boldsymbol{b}$, where $\boldsymbol{A}$ is a matrix that captures a graph's structure, need to be solved. To facilitate solving the linear system, the matrix $\boldsymbol{A}$ is often decomposed into two triangular matrices ($\boldsymbol{L}$ and $\boldsymbol{U}$). In a dynamic world, the graph that models it changes with time and thus is the matrix $\boldsymbol{A}$ that represents the graph. We consider a sequence of evolving graphs and its associated sequence of evolving matrices. We study how LU-decomposition should be done over the sequence so that (1) the decomposition is efficient and (2) the resulting LU matrices best preserve the sparsity of the matrices $\boldsymbol{A}$'s (i.e., the number of extra non-zero entries introduced in $\boldsymbol{L}$ and $\boldsymbol{U}$ are minimized.) We propose a cluster-based algorithm CLUDE for solving the problem. Through an experimental study, we show that CLUDE is about an order of magnitude faster than the traditional incremental update algorithm. The number of extra non-zero entries introduced by CLUDE is also about an order of magnitude fewer than that of the traditional algorithm. CLUDE is thus an efficient algorithm for LU decomposition that produces high-quality LU matrices over an evolving matrix sequence.

1. INTRODUCTION

Graphs are a powerful tool which model real world entities and their relationships through nodes and edges. For example, a graph can be used to model a social network for which users are represented by nodes while their interactions (such as friendship or whether they have recently communicated, etc.) are represented by edges. A graph can also model web pages and the hyperlinks connecting them, or model a bibliographic network capturing co-authorship between authors.

A number of measures have been proposed for analyzing graph structures. Examples include PageRank (PR) [22] and SALSA [18] (which measure the *importance* of nodes), and Personalized PageRank (PPR) [12], Discounted Hitting Time (DHT) [14] and Random Walk with Restart (RWR) [23] (which measure the *proximities* between nodes). These measures have extensive applications, especially in the structural analyses of the underlying information the graphs model. For example, Google employs PR to rank search results [17], and PPR is often used in node clustering and community detection [2].

A common property of these measures is that computing them requires solving certain linear systems. As an example, consider RWR: we start from a node u in a graph and at each step transit to another node. Specifically, with a probability d (called the damping factor), we transit to a neighboring node via an edge, and with a probability (1-d), we transit to the starting node u. We can compute the stationary distribution of the nodes (i.e., how likely that we are at a particular node at any time instant). Intuitively, a large stationary probability of a node v implies that v is *close* to node u under the RWR measure. Let $\boldsymbol{x}_u$ be a vector representing the stationary distribution such that $\boldsymbol{x}_u(v)$ represents the stationary probability of node v, then $\boldsymbol{x}_u$ can be determined by solving:

$$\boldsymbol{x}_u = d\boldsymbol{W}\boldsymbol{x}_u + (1-d)\boldsymbol{q}_u, \tag{1}$$

where $\boldsymbol{W}$ is the column normalized adjacency matrix of the graph[1] and $\boldsymbol{q}_u$ is a vector whose only non-zero entry is $\boldsymbol{q}_u(u) = 1$. Let $\boldsymbol{I}$ be the identity matrix, Eq. 1 can be rewritten as

$$\boldsymbol{A}\boldsymbol{x}_u = \boldsymbol{b}_u,$$

where $\boldsymbol{A} = \boldsymbol{I} - d\boldsymbol{W}$ and $\boldsymbol{b}_u = (1-d)\boldsymbol{q}_u$. In fact, each of the measures we have mentioned can be similarly determined by solving an equation of the form $\boldsymbol{Ax} = \boldsymbol{b}$ for $\boldsymbol{x}$ by composing a matrix $\boldsymbol{A}$ and a vector $\boldsymbol{b}$. In this formulation, the matrix $\boldsymbol{A}$ depends solely on the graph structure (and the measure to be determined), while the vector $\boldsymbol{b}$ can be considered as an *input query* for the measure. For example, by setting $\boldsymbol{b} = \boldsymbol{b}_u$

[1]If (i, j) is an edge in the graph, then $\boldsymbol{W}(j,i) = 1/\lambda(i)$, where $\lambda(i)$ is the out-degree of node i; $\boldsymbol{W}(j,i) = 0$ otherwise.

10.5441/002/edbt.2014.30

for various nodes u, we obtain the stationary distributions of different starting nodes u for the RWR measure.

In a dynamic world, the information modeled by the graph changes with time. For example, a hyperlink is added to a web page, or a Facebook link between two friends is established. The graph thus evolves with time. In [25], it was proposed that evolving graphs should be archived and analyzed as an *Evolving Graph Sequence* (or EGS). An EGS is a sequence of *snapshot graphs*, each of which captures the world's state at a particular instant. As we have discussed, a graph's structure can be conceptually represented by a matrix ($\boldsymbol{A}$) from which various measures can be computed. Hence, as the graph evolves, so are the matrices and the measures. An interesting question is "How shall all these matrices be processed so that graph-based measures can be computed efficiently to support EGS analysis?"

Before we discuss how we tackle the problem, let us first consider a few motivating examples to illustrate how graph-based measures over an EGS could lead to interesting analysis.

Example 1 PageRank (PR) [22] is a widely used metric to measure the importance of hyperlinked web pages. A web publisher who makes money by putting Google Ads on his web contents, for example, would be very interested in the PageRank score of his web site. To illustrate how PageRank scores change with time, we have collected 1000 daily snapshots of a set of 20,000 Wikipedia pages and their hyperlinks. Figure 1 shows how the PR score of a Wiki page labeled 152 changes over the 1000 snapshots. From the figure, we see a number of interesting *key moments* at which the PR score changes significantly. To illustrate, let us discuss a few of them, which are marked by arrows in Figure 1. These PR score changes are also illustrated in Figure 2.

First we see a sharp rise of the score at snapshot #197. On further investigation, we found that at that snapshot, new links pointing to Page 152 were added to two other pages (see Figures 2 (a) and (b)). Since these two pages (Pages 777 and 1169) had very large PR scores, they contributed much to the rise of Page 152's PR score at snapshot #197. Page 152's high PR score, however, was short-lived. A rapid decline of the score was observed at snapshot #247. We found that at that snapshot, a high-PR page (Page 8774), which only pointed to Page 152, was edited with 30 more new outgoing links added to it (see Figure 2 (c)). This drastically reduced Page 8774's contribution to Page 152's PR score, resulting in its sharp drop. Next, we see that the PR score of Page 152 steadily declined over a one-year period (from snapshot #585 to #912). We found that over that period, no new pages with large PR scores linked to Page 152 while at the same time the out-degrees of the pages that were pointing to Page 152 gradually increased. Hence, their contributions to Page 152's PR scores were gradually reduced. □

In this example, we see that interesting events occurred which led to significant changes to the measure. To discover such events, we need to identify the key moments in order to focus the investigation over a manageable set of snapshots. To achieve that, it is best to display the measure as a time series from which key moments are extracted. This, in turns, requires that the measure be evaluated over the whole EGS.

Example 2 In Google's official guide on improving a web page's PR scores[2], a number of actions are recommended. Some of these actions include translating the web page to other languages, publicizing the web site through newsletters, providing a rich site summary (RSS), and submitting the web site to various web directories, etc. How shall we evaluate the effectiveness of these actions? What are the usual lag times between the actions and their observable effects? To answer those questions, we need to systematically analyze web pages' PR scores as time series (such as the one shown in Figure 1) and discover any association between various actions taken and any observable changes to the measure. This again requires the PR score be computed at every snapshot. □

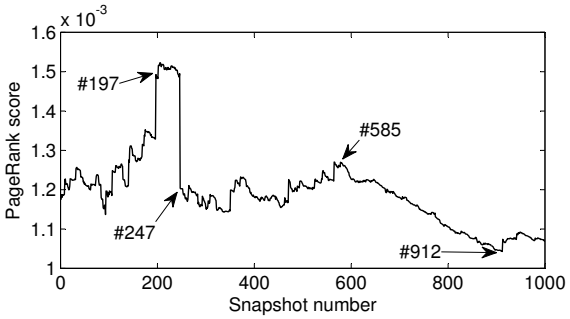


Figure 1: PR score of Wiki Page 152 over a 1000-day EGS.

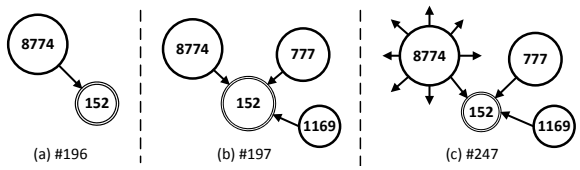


Figure 2: Graph structure at specific snapshot.

Example 3 Link prediction has been a popular topic especially in the data mining literature. Most of existing works on link prediction consider a static graph snapshot and evaluate certain "proximity" measure (e.g., RWR) with which "closest node pairs" are identified as potential endpoints of links. If one has access to not only, say, the RWR scores on a single graph snapshot, but the time series of the scores over an EGS, then the upward/downward trends of the scores provide an important dimension based on which link predictions could be made. The availability of the scores as time series allows a wealth of prediction techniques to be applied to predict links, such as those mentioned in [16] and [7]. □

An EGS is a sequence of graphs $\mathcal{G} = \{G_1, \ldots, G_T\}$. As we have mentioned, each graph G_i derives a matrix $\boldsymbol{A}_i$, which is composed based on the structure of G_i and the kind of measure to be evaluated. An EGS thus derives an *evolving matrix sequence* (EMS) $\mathcal{M} = \{\boldsymbol{A}_1, \ldots, \boldsymbol{A}_T\}$. To obtain a series of measure values (such as Figure 1), we need to solve the equation $\boldsymbol{A}_i\boldsymbol{x} = \boldsymbol{b}$ for each matrix $\boldsymbol{A}_i$. Our objective is to study how these equations can be solved efficiently.

A standard method to solve a linear system is to perform

[2] `http://www.googleguide.com/improving_pagerank.html`

Gaussian Elimination (or *GE*), which is a rather expensive operation for large graphs. Recall that the vector $\boldsymbol{b}$ is an *input* of the measure (e.g., we set $\boldsymbol{b} = \boldsymbol{b}_u$ to compute the RWR scores for the case where the starting node of the random walk is u). Repeatedly applying GE for each input $\boldsymbol{b}$ is very expensive. An alternative approach (*LU decomposition*) is to first decompose the matrix $\boldsymbol{A}_i$ as a product of a lower triangular matrix $\boldsymbol{L}_i$ and an upper triangular matrix $\boldsymbol{U}_i$ (i.e., $\boldsymbol{A}_i = \boldsymbol{L}_i\boldsymbol{U}_i$). Although LU decomposition is computationally similar to GE (and so they have similar cost), once the matrix $\boldsymbol{A}_i$ is decomposed, all linear systems of the same matrix $\boldsymbol{A}_i$ can be solved very efficiently using forward and backward substitution methods [9]. Hence using LU decomposition allows us to avoid performing expensive GE repeatedly for different input $\boldsymbol{b}$'s. As an example, with our Wikipedia dataset, once the matrix is LU-decomposed, solving the linear system is about 5,000 times faster than executing one GE. Hence, we reduce the problem of solving the linear system $\boldsymbol{A}_i\boldsymbol{x} = \boldsymbol{b}$ to decomposing the matrix $\boldsymbol{A}_i$.

To derive an efficient method to decompose all the matrices in an EMS, we first need to understand the properties of the EMS. For most applications of interest, the snapshot graphs (and hence the matrices of an EMS) are (1) *sparse* and (2) *gradually evolving*. As an example, for our Wikipedia dataset, the average out-degree of a node is 7. Also, successive snapshots share more than 99% of their edges. The first property calls for LU decomposition methods that are specialized for sparse matrices, while the second property suggests incremental LU decomposition be applied. Let us briefly review the two techniques.

Given a sparse matrix $\boldsymbol{A}$, decomposing it into $\boldsymbol{L}$ and $\boldsymbol{U}$ usually does not perfectly preserve its *sparsity*. That is, some 0 entries in $\boldsymbol{A}$ would become non-zero in $\boldsymbol{L}$ and $\boldsymbol{U}$. These entries are called *fill-ins*. A large number of fill-ins is undesirable because (1) it takes more space to store the matrices $\boldsymbol{L}$ and $\boldsymbol{U}$, (2) it slows down forward and backward substitutions in solving the linear systems, and most important of all, (3) it increases the decomposition time[3]. A common approach to reduce the number of fill-ins is to apply a *reordering technique*, which shuffles the rows and columns of $\boldsymbol{A}$ before the matrix is decomposed. Although finding the optimal ordering of $\boldsymbol{A}$ to minimize the number of fill-ins is an NP-Complete problem [26], there are a few heuristic reordering strategies, such as Markowitz [20] and AMD [1], which have been shown to be very effective in reducing the number of fill-ins in practice. The *quality* of an LU decomposition refers to the number of fill-ins. Intuitively, the fewer fill-ins, the higher the quality of the decomposition. Different orderings of the matrix induce different decomposition quality. As we have mentioned, a higher-quality decomposition generally gives faster decomposition time as well as faster equation solving time (in the execution of forward/backward substitutions).

There are previous studies on how to perform incremental LU decomposition. Specifically, given a matrix $\boldsymbol{A}_1$ and a low-rank matrix $\Delta\boldsymbol{A}_1$, Bennett's algorithm [5] computes the LU factors of $\boldsymbol{A}_2 = \boldsymbol{A}_1 + \Delta\boldsymbol{A}_1$ from the LU factors of $\boldsymbol{A}_1$ and the updates $\Delta\boldsymbol{A}_1$. The complexity of Bennett's algorithm is proportional to the rank of $\Delta\boldsymbol{A}_1$ and the number of non-zero entries in the LU factors of $\boldsymbol{A}_2$. It has been shown that Bennett's algorithm is much more efficient than directly computing the LU decomposition on $\boldsymbol{A}_2$ if the update matrix $\Delta\boldsymbol{A}_1$ has a low rank.

Ideally, to achieve fast LU decomposition over an EMS, we should perform reordering to reduce the number of fill-ins and apply incremental LU decomposition such as Bennett's algorithm. Unfortunately, integrating the two techniques is tricky. This is because to apply an incremental algorithm, the same ordering (if any) has to be applied to both matrices $\boldsymbol{A}_1$ and $\boldsymbol{A}_2$. However, an ordering that is best for $\boldsymbol{A}_1$ may not be so for $\boldsymbol{A}_2$. This issue is even more pronounced when we attempt to apply Bennett's algorithm onto a long EMS. This is because a good ordering for $\boldsymbol{A}_1$ could be badly unfit for the last matrix $\boldsymbol{A}_T$ of the EMS, resulting in large numbers of fill-ins and thus very slow LU decomposition.

Another issue of applying incremental LU decomposition over an EMS is how the various factors $\boldsymbol{L}_i$ and $\boldsymbol{U}_i$ are represented. Since these LU factors are typically sparse, they are implemented using adjacency lists, which store the non-zero entries of the factors. When we apply Bennett's algorithm to compute $\boldsymbol{L}_2$, $\boldsymbol{U}_2$ from $\boldsymbol{L}_1$, $\boldsymbol{U}_1$, the adjacency lists representing $\boldsymbol{L}_1$, $\boldsymbol{U}_1$ would have to be (structurally) modified to form the adjacency lists for $\boldsymbol{L}_2$, $\boldsymbol{U}_2$. This structural update of the data structures (such as node inserts and deletes in the linked lists) turns out to be a dominating cost of the incremental algorithm compared with the numerical computation.

In this paper we propose an algorithm CLUDE for performing LU decomposition on matrices of an EMS. CLUDE groups the matrices in an EMS into clusters and apply an incremental algorithm to decompose the matrices within each cluster. The idea is that if matrices of the same clusters are sufficiently similar with each other, then we may derive an ordering that generally fits all the matrices in the cluster. This cluster-based ordering allows the decomposition of the matrices to be of high quality, which leads to faster LU decomposition. Moreover, since the same ordering is applied to all the matrices in the cluster, an incremental decomposition algorithm can be applied. Finally, CLUDE applies symbolic computation to build an adjacency-lists structure that covers all the non-zero entries of all the LU factors of a cluster. This avoids the expensive structural changes to adjacency lists that happens when the incremental algorithm is straightforwardly applied.

Here we summarize our contributions.

- We propose to study the problem of LU decomposition over a sequence of evolving matrices, which finds many applications especially those that involve the sequential analysis of graph structural measures.
- We study the interaction between matrix ordering and incremental decomposition algorithm, with a focus on optimizing decomposition quality and speed.
- We propose the algorithm CLUDE which employs a clustering strategy to partition the matrices in an EMS so that a universal ordering and a universal data structure can be applied to all the matrices in a cluster.
- We perform an extensive experimental study using both real and synthetic datasets to evaluate CLUDE and compare it against other decomposition methods. Our experiment shows that CLUDE is up to an order of

[3]A number of factors affect the execution time of LU decomposition on a sparse matrix. A major factor is the number of non-zero entries in the resulting L, U matrices. The general complexity, however, is unknown [9].

Table 1: Glossary of symbols

Symbols	Meanings
n	number of nodes in a snapshot graph
$\boldsymbol{A}$, $\boldsymbol{L}$, $\boldsymbol{U}$	an $n \times n$ matrix and its LU factors
$\widehat{\boldsymbol{A}}$	decomposed version of $\boldsymbol{A}$
$\boldsymbol{x}$, $\boldsymbol{b}$	$n \times 1$ vectors
$sp(\boldsymbol{A})$	sparsity pattern of $\boldsymbol{A}$, i.e., $\{(i,j)\|\boldsymbol{A}(i,j) \neq 0\}$
$fp(\boldsymbol{A})$	fill-in pattern of $\boldsymbol{A}$
$\tilde{sp}(\boldsymbol{A})$	symbolic sparsity pattern of $\boldsymbol{A}$, i.e., $sp(\boldsymbol{A}) \cup fp(\boldsymbol{A})$
$\mathcal{O}$, $\mathcal{O}^*(\boldsymbol{A})$	a matrix ordering; the Markowitz order of $\boldsymbol{A}$
$\boldsymbol{A}^{\mathcal{O}}, \boldsymbol{L}^{\mathcal{O}}, \boldsymbol{U}^{\mathcal{O}}$	reordered matrix $\boldsymbol{A}$ and its LU factors
$\widehat{\boldsymbol{A}}^{\mathcal{O}}$	decomposed reordered version of matrix $\boldsymbol{A}$
$\boldsymbol{A}^*$	reordered matrix $\boldsymbol{A}$ with $\mathcal{O}^*(\boldsymbol{A})$, i.e., $\boldsymbol{A}^* = \boldsymbol{A}^{\mathcal{O}^*(\boldsymbol{A})}$
$ql(\mathcal{O}, \boldsymbol{A})$	quality-loss of applying $\mathcal{O}$ on $\boldsymbol{A}$
$\boldsymbol{A}_\cap, \boldsymbol{A}_\cup$	intersection and union of the matrices in a cluster in terms of their sparsity patterns

magnitude faster than the basic incremental algorithm and at the same time achieves up to an order of magnitude smaller number of fill-ins.

The rest of this paper is organized as follows. In Section 2 we cover the basics of traditional LU decomposition. We present a formal problem definition in Section 3 and describe the various algorithms in Section 4. In Section 5 we extend our solution to one that can guarantee decomposition quality. In Section 6 we present the experimental results. In Section 7 we present a case study showing how interesting observations can be obtained by analyzing certain measures on real data. In Section 8 we discuss some related works. Finally, we conclude the paper in Section 9.

2. PRELIMINARY

In this section we give some details of LU decomposition and matrix reordering. We will also define the various symbols and notations used in the paper. Figure 3 illustrates the various concepts and Table 1 lists the frequently used symbols.

2.1 LU decomposition

A system of linear equations $\boldsymbol{Ax} = \boldsymbol{b}$, where $\boldsymbol{A}$ is an $n \times n$ non-singular matrix, has a unique solution $\boldsymbol{x} = \boldsymbol{A}^{-1}\boldsymbol{b}$. A straightforward method to solve for $\boldsymbol{x}$ is to first invert the matrix $\boldsymbol{A}$. After that, $\boldsymbol{x}$ can be computed by multiplying $\boldsymbol{A}^{-1}$ to any input query $\boldsymbol{b}$. The problem of this approach is that when $\boldsymbol{A}$ is sparse, $\boldsymbol{A}^{-1}$ is usually dense [24] (e.g., see Figures 3(b) and (i) for a sparse matrix $\boldsymbol{A}$ and its dense inverse). It thus takes $O(n^2)$ space to store $\boldsymbol{A}^{-1}$, which is impractical for large graphs (and matrices). Besides, computing $\boldsymbol{A}^{-1}\boldsymbol{b}$ takes $O(n^2)$ time (because $\boldsymbol{A}^{-1}$ is dense), which is too expensive, considering that it has to be done for every input query $\boldsymbol{b}$. The matrix inversion method is thus impractical.

To facilitate solving $\boldsymbol{x}$ for various input query $\boldsymbol{b}$, we decompose a matrix $\boldsymbol{A}$ into its LU factors. The decomposition can be done by the Crout's method [9]. Figure 3(c) shows an example decomposition. Now, $\{\boldsymbol{Ax} = \boldsymbol{b}\} \Leftrightarrow \{\boldsymbol{L}(\boldsymbol{Ux}) = \boldsymbol{b}\}$. To find $\boldsymbol{x}$, we first perform forward substitution to get $\boldsymbol{Ux}$, followed by a backward substitution process to obtain $\boldsymbol{x}$ [9].

2.2 Preserving sparsity in LU decomposition

Let $\widehat{\boldsymbol{A}} = \boldsymbol{L}+\boldsymbol{U}$ be the *decomposed representation* of matrix $\boldsymbol{A}$. If the number of non-zero entries in $\widehat{\boldsymbol{A}}$ is k, then the complexity of forward/backward substitutions is $O(k)$. Also, the time complexity of Crout's method is a function of k. As we have explained in the introduction, a good decomposition should preserve the sparsity of the matrix $\boldsymbol{A}$ as much as possible. That is, k should be kept small, which is typically achieved by applying some reordering technique. Here, we briefly discuss reordering. First, some definitions.

DEFINITION 1 (SPARSITY PATTERN). *Given a matrix $\boldsymbol{A}$, its sparsity pattern, denoted by $sp(\boldsymbol{A})$, is the set of indices of $\boldsymbol{A}$ at which $\boldsymbol{A}$ contains non-zero values. That is, $sp(\boldsymbol{A}) := \{(i,j) \mid \boldsymbol{A}(i,j) \neq 0\}$.*

Figure 3(a) shows an illustration of a sparsity pattern. Note that decomposing a matrix $\boldsymbol{A}$ into its LU factors may introduce extra non-zero entries. This is illustrated by Figures 3(a) and 3(d), which show the sparsity pattern of the original matrix $\boldsymbol{A}$ and that of its decomposed form $\widehat{\boldsymbol{A}}$, respectively. These extra non-zero entries introduced by LU decomposition are called fill-ins. (In Figure 3(d), fill-ins are shown as dark grey entries.)

To reduce the number of fill-ins, we reorder the matrix $\boldsymbol{A}$ based on an ordering $\mathcal{O}$.

DEFINITION 2 (ORDERING). *An ordering $\mathcal{O} = (\boldsymbol{P}, \boldsymbol{Q})$ is a pair of n-by-n permutation matrices $\boldsymbol{P}$, $\boldsymbol{Q}$. Each row or column of a permutation matrix contains exactly one non-zero entry, whose value is 1. (Figure 3(j) shows an example.) We say that a matrix $\boldsymbol{A}$ is reordered by the ordering $\mathcal{O}$ into a matrix $\boldsymbol{A}^{\mathcal{O}}$ if $\boldsymbol{A}^{\mathcal{O}} = \boldsymbol{PAQ}$.*

Figure 3(f) shows a reordered matrix. Instead of decomposing the original matrix $\boldsymbol{A}$, we decompose the reordered matrix $\boldsymbol{A}^{\mathcal{O}}$ instead into two factors, denoted by $\boldsymbol{L}^{\mathcal{O}}$ and $\boldsymbol{U}^{\mathcal{O}}$ (see Figure 3(g)). The purpose of reordering the matrix is to reduce the number of fill-ins. Let $\widehat{\boldsymbol{A}}^{\mathcal{O}} = \boldsymbol{L}^{\mathcal{O}} + \boldsymbol{U}^{\mathcal{O}}$ be the decomposed ("$\widehat{\ }$") reordered ("$\mathcal{O}$") version of $\boldsymbol{A}$. Figure 3(h) shows its sparsity pattern, $sp(\widehat{\boldsymbol{A}}^{\mathcal{O}})$. Compared against the sparsity pattern of $sp(\boldsymbol{A}^{\mathcal{O}})$ (Figure 3(e)), there is only one fill-in brought about by the decomposition. The reordering step has thus resulted in much fewer fill-ins compared with the original decomposition (Figure 3(d)). One of the best reordering strategies is given by Markowitz, which has been shown to be very effective [20].

Given the LU factors, $\boldsymbol{L}^{\mathcal{O}}$ and $\boldsymbol{U}^{\mathcal{O}}$, of $\boldsymbol{A}^{\mathcal{O}}$, solving the original equation $\boldsymbol{Ax} = \boldsymbol{b}$ for $\boldsymbol{x}$ is simple. Note that,

$$\{\boldsymbol{Ax} = \boldsymbol{b}\} \Leftrightarrow \{\boldsymbol{P}^{-1}\boldsymbol{A}^{\mathcal{O}}\boldsymbol{Q}^{-1}\boldsymbol{x} = \boldsymbol{b}\} \Leftrightarrow \{\boldsymbol{A}^{\mathcal{O}}(\boldsymbol{Q}^{-1}\boldsymbol{x}) = \boldsymbol{Pb}\}.$$

Let $\boldsymbol{x}' = \boldsymbol{Q}^{-1}\boldsymbol{x}$ and $\boldsymbol{b}' = \boldsymbol{Pb}$, we have, $\boldsymbol{A}^{\mathcal{O}}\boldsymbol{x}' = \boldsymbol{b}'$. Given $\boldsymbol{L}^{\mathcal{O}}$ and $\boldsymbol{U}^{\mathcal{O}}$, $\boldsymbol{x}'$ can be solved efficiently using forward/backward substitutions. Finally, $\boldsymbol{x}$ is computed by $\boldsymbol{x} = \boldsymbol{Qx}'$. Note that the permutation matrices $\boldsymbol{P}$ and $\boldsymbol{Q}$ contain only one non-zero entry in each row or column. Therefore, computing $\boldsymbol{b}' = \boldsymbol{Pb}$ and $\boldsymbol{x} = \boldsymbol{Qx}'$ takes only $O(n)$ time.

2.3 Implementing LU decomposition on sparse matrices

For most applications of interest, the matrix $\boldsymbol{A}$ and its LU factors $\boldsymbol{L}$, $\boldsymbol{U}$ are sparse. They are thus typically represented

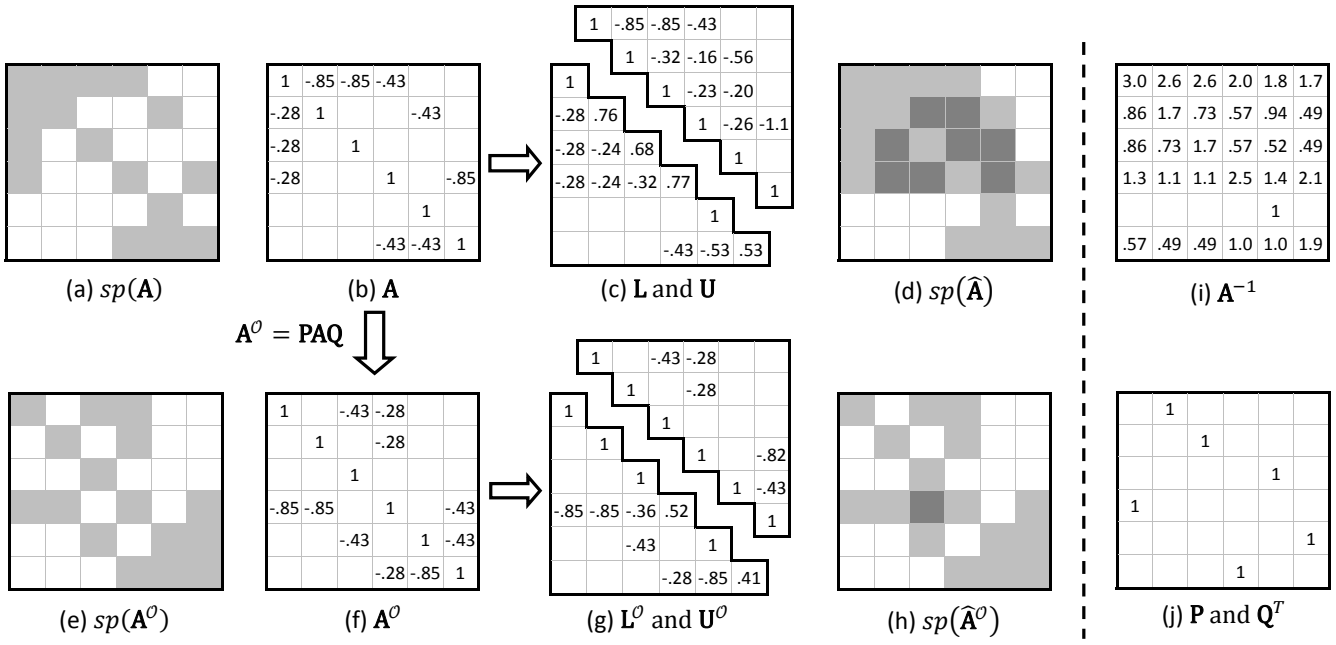


Figure 3: Illustration of LU decomposition, sparsity pattern, fill-ins, reordering, and matrix inverse.

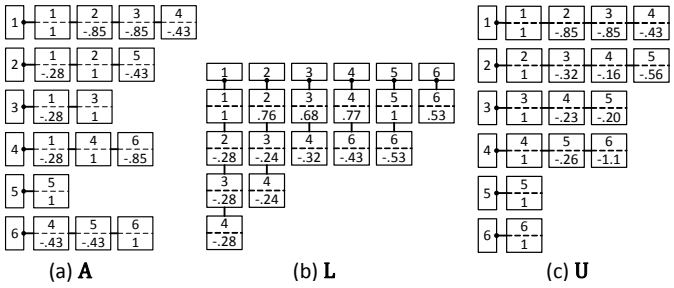


Figure 4: Data structures for storing a matrix and its LU factors

using adjacency lists. Figure 4 shows the data structures for representing the matrix and its factors. The decomposition process consists of two phases [9], namely, (1) symbolic decomposition (SD-phase) and (2) numerical decomposition (ND-phase). The purpose of the SD-phase is to determine the locations of all possible fill-ins so that the data structures for representing the LU factors (see Figures 4 (b) and 4(c)) can be efficiently created. In the ND-phase, the actual values of the entries are computed.

More specifically, in the SD-phase, we determine a fill-in pattern, $fp(\boldsymbol{A})$ [26], given by

$$\begin{aligned} fp(\boldsymbol{A}) = \{(u,v) \notin sp(\boldsymbol{A}) \mid \exists u_1, \ldots, u_k, \text{s.t.} \\ (1)\ k \geq 1, \\ (2)\ u_i < \min\{u,v\}\ \forall 1 \leq i \leq k, \\ (3)\ (u,u_1), (u_i,u_{i+1}), (u_k,v) \in sp(\boldsymbol{A})\ \forall 1 \leq i < k\}. \end{aligned} \tag{2}$$

In words, w.r.t. the graph from which the matrix $\boldsymbol{A}$ is derived, the node pair (u,v) is in $fp(\boldsymbol{A})$ if there is a path of length-2 or longer from u to v such that none of the nodes visited along this path has an index larger than those of u and v. We define the *symbolic sparsity pattern*, $\tilde{sp}(\boldsymbol{A})$ of a matrix $\boldsymbol{A}$ as the union of $\boldsymbol{A}$'s sparsity pattern and fill-in pattern, i.e.,

$$\tilde{sp}(\boldsymbol{A}) = sp(\boldsymbol{A}) \cup fp(\boldsymbol{A}). \tag{3}$$

It can be shown that $fp(\boldsymbol{A})$, as defined in Eq. 2, covers all fill-ins' locations and so $\tilde{sp}(\boldsymbol{A})$ covers all the locations in $sp(\widehat{\boldsymbol{A}})$ (Figure 3(d)), i.e., $sp(\widehat{\boldsymbol{A}}) \subseteq \tilde{sp}(\boldsymbol{A})$. Hence, by determining $\tilde{sp}(\boldsymbol{A})$ in the SD-phase, we get to cover all non-zero locations of the LU factors. So, the data structures for storing the LU factors can be prepared before the numerical decomposition.

Note that our discussion of the fill-in pattern and the symbolic sparsity pattern is orthogonal to whether reordering is done. In other words, if an ordering $\mathcal{O}$ is first applied to the matrix $\boldsymbol{A}$ before it is decomposed, then the fill-in pattern and the symbolic sparsity pattern are defined on the matrix $\boldsymbol{A}^{\mathcal{O}}$, giving $fp(\boldsymbol{A}^{\mathcal{O}})$ and $\tilde{sp}(\boldsymbol{A}^{\mathcal{O}})$.

3. PROBLEM DEFINITION

As we have discussed in the introduction, reordering and incremental decomposition are two techniques we can apply in decomposing the matrices in an EMS. Different orderings $\mathcal{O}_i$, when applied to a matrix $\boldsymbol{A}_i$, result in different symbolic sparsity patterns $\tilde{sp}(\boldsymbol{A}_i^{\mathcal{O}_i})$. Note that the larger $\tilde{sp}(\boldsymbol{A}_i^{\mathcal{O}_i})$ is, the larger is the data structure for storing the LU factors (see Section 2.3), and the longer does it take to perform the decomposition and to solve the linear system $\boldsymbol{A}_i\boldsymbol{x} = \boldsymbol{b}$. Therefore, it is important that a good ordering $\mathcal{O}_i$ for each matrix $\boldsymbol{A}_i$ be found, such that the size of $\tilde{sp}(\boldsymbol{A}_i^{\mathcal{O}_i})$ is small.

One of the best reordering strategies is given by Markowitz. For any matrix $\boldsymbol{A}$, let $\mathcal{O}^*(\boldsymbol{A})$ be the Markowitz order of $\boldsymbol{A}$, and let $\boldsymbol{A}^*$ be $\boldsymbol{A}$ reordered with $\mathcal{O}^*(\boldsymbol{A})$ (i.e., $\boldsymbol{A}^* = \boldsymbol{A}^{\mathcal{O}^*(\boldsymbol{A})}$). Ideally, each $\boldsymbol{A}_i$ in an EMS should be reordered into "*its best form*" $\boldsymbol{A}_i^*$ before it is decomposed. There are, however, two problems with this approach. First, determining the

Markowitz order of a matrix is generally as expensive as doing a Gaussian Elimination [9]. So, finding the Markowitz order for every matrix in an EMS is very expensive. Second, to apply an incremental LU decomposition algorithm on two successive matrices $\boldsymbol{A}_i$ and $\boldsymbol{A}_{i+1}$, if we apply an ordering on $\boldsymbol{A}_i$, the same ordering has to be applied to $\boldsymbol{A}_{i+1}$ as well. However, $\mathcal{O}^*(\boldsymbol{A}_i)$ and $\mathcal{O}^*(\boldsymbol{A}_{i+1})$ could be different. As a result, an algorithm for decomposing the matrices in an EMS has to be selective in determining what orderings are applied to the matrices, and which matrices in the sequence should share the same ordering so that efficient incremental decomposition can be performed on them. With this discussion, we are ready to formally define the problem of *LU Decomposition over an Evolving Matrix sequence* (LUDEM).

Definition 3 (The LUDEM Problem). *Given an EMS $\mathcal{M} = \{\boldsymbol{A}_1, \boldsymbol{A}_2, \ldots, \boldsymbol{A}_T\}$, where each $\boldsymbol{A}_i$ is an $n \times n$ sparse matrix, determine, for $1 \leq i \leq T$, an ordering $\mathcal{O}_i$ for $\boldsymbol{A}_i$ and compute the LU factors of $\boldsymbol{A}_i^{\mathcal{O}_i}$.*

We can evaluate an algorithm for solving the LUDEM problem by two metrics: (1) how fast it executes and (2) how good the orderings $\mathcal{O}_i$'s are. Since Markowitz is a known method for generating very good orderings. We use the Markowitz order $\mathcal{O}^*(\boldsymbol{A}_i)$ as a quality reference, and define the *quality-loss* of an ordering as follows.

Definition 4 (Quality-loss of an ordering). *Given an ordering $\mathcal{O}$ of a matrix $\boldsymbol{A}$, the quality-loss of $\mathcal{O}$ on $\boldsymbol{A}$, denoted by $ql(\mathcal{O}, \boldsymbol{A})$, is given by,*

$$ql(\mathcal{O}, \boldsymbol{A}) = \frac{|\tilde{sp}(\boldsymbol{A}^{\mathcal{O}})| - |\tilde{sp}(\boldsymbol{A}^*)|}{|\tilde{sp}(\boldsymbol{A}^*)|}. \quad (4)$$

That is, we compare the size of the symbolic sparsity pattern of $\boldsymbol{A}^{\mathcal{O}}$ against that of the Markowitz ordered $\boldsymbol{A}^*$. Note that a smaller $ql(\mathcal{O}, \boldsymbol{A})$ implies a higher ordering quality.

In general, $\tilde{sp}(\boldsymbol{A}^*)$ cannot be determined without determining the Markowitz ordering and decomposing $\boldsymbol{A}^*$. However, for the special case in which $\boldsymbol{A}$ is a symmetric matrix, it has been shown that its Markowitz ordering and $\tilde{sp}(\boldsymbol{A}^*)$ can be determined very efficiently without physically decomposing the matrix [1, 13]. In this case, an algorithm for solving the LUDEM problem can very efficiently evaluate (using Equation 4) the quality-loss of the orderings it produces. In particular, the algorithm can perform quality control on its own output. So, for the special case of symmetric matrices, we extend the LUDEM problem to one that has an additional quality constraint. We call this problem LUDEM-QC.

Definition 5 (The LUDEM-QC Problem). *Given an EMS $\mathcal{M} = \{\boldsymbol{A}_1, \boldsymbol{A}_2, \ldots, \boldsymbol{A}_T\}$, where each $\boldsymbol{A}_i$ is an $n \times n$ sparse symmetric matrix, and a quality requirement $\beta \geq 0$, determine, for $1 \leq i \leq T$, an ordering $\mathcal{O}_i$ for $\boldsymbol{A}_i$ such that $ql(\mathcal{O}_i, \boldsymbol{A}_i) \leq \beta$, and compute the LU factors of $\boldsymbol{A}_i^{\mathcal{O}_i}$.*

4. ALGORITHMS FOR LUDEM

In this section we describe algorithms for solving the LUDEM problem.

[Brute Force (BF)] The brute force method (BF) determines the Markowitz ordering $\mathcal{O}^*(\boldsymbol{A}_i)$ of each matrix $\boldsymbol{A}_i$, reorders $\boldsymbol{A}_i$ to the Markowitz ordered $\boldsymbol{A}_i^*$ and then decomposes $\boldsymbol{A}_i^*$. Under BF, $\mathcal{O}_i = \mathcal{O}^*(\boldsymbol{A}_i)$. BF is generally slow because it takes much time to determine the orderings of all matrices and it does not employ a fast incremental decomposition algorithm. However, BF achieves the best ordering quality because all matrices are Markowitz ordered. We will use BF as the baseline with which the performances of other algorithms are measured. In particular, we evaluate the ordering quality of other algorithms against Markowitz orderings (see Definition 4). Also, the execution times of other algorithms are expressed as speedup factors over BF.

[Straightly Incremental (INC)] The INC algorithm first determines the Markowitz ordering of $\boldsymbol{A}_1$ and applies the ordering to every matrix in the EMS to obtain $\boldsymbol{A}_i^{\mathcal{O}^*(\boldsymbol{A}_1)}$ for all $1 \leq i \leq T$. INC then decomposes $\boldsymbol{A}_1^{\mathcal{O}^*(\boldsymbol{A}_1)}$ followed by applying Bennett's algorithm to incrementally decompose the successive matrices $\boldsymbol{A}_2^{\mathcal{O}^*(\boldsymbol{A}_1)}, \ldots, \boldsymbol{A}_T^{\mathcal{O}^*(\boldsymbol{A}_1)}$. Hence, under INC, $\mathcal{O}_i = \mathcal{O}^*(\boldsymbol{A}_1)$. INC computes only one Markowitz ordering and performs only one full decomposition, in addition to executing Bennett's algorithm $T - 1$ times.

A problem with INC is that the ordering quality deteriorates as we move from $\boldsymbol{A}_1$ to $\boldsymbol{A}_T$ because the matrices deviate from $\boldsymbol{A}_1$ progressively. As we have explained, a bad ordering makes decomposition (full or incremental) slower because of a much larger number of fill-ins in the LU factors. However, to apply Bennett's algorithm, the matrices have to share the same ordering. Our next two algorithms attempt to strike a balance between ordering quality and the applicability of incremental decomposition. The idea is to partition the EMS into clusters such that matrices within the same cluster are sufficiently similar. With highly-similar cluster members, a single ordering can be shared by all members of a cluster and yet the ordering is of good enough quality. We call our next two algorithms *cluster-based* algorithms. Before their descriptions, we first give the details of the clustering procedure.

In order to group matrices in an EMS into clusters, we need to define a similarity measure. We measure two matrices' similarity by comparing the structures of their underlying graphs, which are conveniently captured by the sparsity patterns of the matrices (see Figure 3(a)). Specifically, we use a *normalized matrix edit similarity* (*mes*) measure that is based on the symmetric difference of the matrices' sparsity patterns:

Definition 6 (Matrix edit similarity). *Given two matrices $\boldsymbol{A}_a$ and $\boldsymbol{A}_b$,*

$$mes(\boldsymbol{A}_a, \boldsymbol{A}_b) := \frac{2|sp(\boldsymbol{A}_a) \cap sp(\boldsymbol{A}_b)|}{|sp(\boldsymbol{A}_a)| + |sp(\boldsymbol{A}_b)|}. \quad (5)$$

Let $\mathcal{C} = \{\boldsymbol{A}_1, ..., \boldsymbol{A}_t\}$ be a cluster of t matrices[4]. We derive two bounding matrices $\boldsymbol{A}_\cap$ and $\boldsymbol{A}_\cup$, which are the *intersection* and *union* of the matrices in $\mathcal{C}$ in terms of their sparsity patterns. Formally,

Definition 7 ($\boldsymbol{A}_\cap$, $\boldsymbol{A}_\cup$). *For all $1 \leq i, j \leq n$,*

$$\boldsymbol{A}_\cap(i,j) := \begin{cases} 1 & \text{if } (i,j) \in \bigcap_{k=1}^{t} sp(\boldsymbol{A}_k), \\ 0 & \text{otherwise;} \end{cases}$$

$$\boldsymbol{A}_\cup(i,j) := \begin{cases} 1 & \text{if } (i,j) \in \bigcup_{k=1}^{t} sp(\boldsymbol{A}_k), \\ 0 & \text{otherwise.} \end{cases}$$

It can be easily seen that,

[4]W.l.o.g., we assume that the cluster starts with the matrix index 1.

PROPERTY 1. $sp(\boldsymbol{A}_\cap) \subseteq sp(\boldsymbol{A}_i) \subseteq sp(\boldsymbol{A}_\cup) \quad \forall 1 \leq i \leq t.$

Hence, $\boldsymbol{A}_\cap$ and $\boldsymbol{A}_\cup$ *sandwich* the matrices in $\mathcal{C}$. We can thus measure the *compactness* of the cluster by the similarity between $\boldsymbol{A}_\cap$ and $\boldsymbol{A}_\cup$.

DEFINITION 8 (α-BOUNDEDNESS). *A cluster $\mathcal{C}$ of matrices is said to be α-bounded if and only if $mes(\boldsymbol{A}_\cap, \boldsymbol{A}_\cup) \geq \alpha$.*

Since typically the matrices in an EMS are progressively evolving, we use a simple segmentation strategy to partition the matrices of an EMS into clusters. Specifically, given a user-specified similarity threshold α, we start with an empty cluster $\mathcal{C}_1$ and incrementally insert the matrices into $\mathcal{C}_1$ starting with $\boldsymbol{A}_1$, then $\boldsymbol{A}_2$, etc., as long as $\mathcal{C}_1$ remains α-bounded. If the bounding requirement would have been violated by adding one more matrix, we start building the next cluster $\mathcal{C}_2$ and repeat the process. We call this α-clustering. Algorithm 1 shows the clustering algorithm.

Algorithm 1: α-clustering.

Input : EMS $\mathcal{M} = \{\boldsymbol{A}_1, \boldsymbol{A}_2, \ldots, \boldsymbol{A}_T\}$, Similarity threshold α
Output: Clusters $\{\mathcal{C}_1, \mathcal{C}_2, \ldots, \mathcal{C}_j\}$

```
1  j ← 1; C_j ← {A_1}
2  for i ← 2 to T do
3  |   Construct A_∩, A_∪ from C_j ∪ {A_i} based on Definition 7
4  |   if mes(A_∩, A_∪) ≥ α then
5  |   |   C_j ← C_j ∪ {A_i}
6  |   else                 // start building the next cluster
7  |   |   j ← j + 1; C_j ← {A_i}
8  |   end
9  end
10 return {C_1, C_2, ..., C_j}
```

Note that a larger α implies that $\boldsymbol{A}_\cap$ and $\boldsymbol{A}_\cup$ of a cluster are more similar, which then implies a tighter bounding requirement. This results in fewer matrices in a cluster and more clusters segmented from an EMS.

[Cluster-based Incremental (CINC)] Our next algorithm CINC applies INC on each cluster independently. More specifically, for each cluster $\mathcal{C}$, CINC determines the Markowitz ordering of the first matrix in $\mathcal{C}$ and applies that ordering to all the matrices in $\mathcal{C}$. After that, it decomposes the first matrix of $\mathcal{C}$ followed by applying Bennett's algorithm to incrementally decompose the other matrices in the cluster. Algorithm 2 shows the pseudo code of CINC.

Algorithm 2: CINC on one cluster.

Input : A cluster $\mathcal{C} = \{\boldsymbol{A}_1, \boldsymbol{A}_2, \ldots, \boldsymbol{A}_t\}$
Output: Ordering and LU factors of $\boldsymbol{A}_i$, for $1 \leq i \leq t$

```
1  O_1 ← O*(A_1)
2  (L_1^{O_1}, U_1^{O_1}) ← LU decomposition on A_1^{O_1}
3  for i ← 2 to t do
4  |   O_i ← O_1
5  |   ΔA ← A_i^{O_1} − A_{i−1}^{O_1}
6  |   (L_i^{O_i}, U_i^{O_i}) ← Bennett(A_{i−1}^{O_1}, ΔA, L_{i−1}^{O_1}, U_{i−1}^{O_1})
7  end
8  return {O_1, ..., O_t}, and {(L_1^{O_1}, U_1^{O_1}), ..., (L_t^{O_t}, U_t^{O_t})}
```

[Fast Cluster-based LU Decomposition (CLUDE)] Given two consecutive matrices $\boldsymbol{A}_i$ and $\boldsymbol{A}_{i+1}$ in a cluster $\mathcal{C}$, their symbolic sparsity patterns are typically different. The adjacency-lists structures for storing their LU factors are therefore different (see Section 2.3). As we apply Bennett's algorithm to obtain the LU factors of matrix $\boldsymbol{A}_{i+1}$ from those of $\boldsymbol{A}_i$, the list structures of $\boldsymbol{A}_{i+1}$ are dynamically created based on those of $\boldsymbol{A}_i$. We have profiled the execution of Bennett's algorithm. Interestingly, about 70% of its execution time is spent on constructing the list structures of $\boldsymbol{A}_{i+1}$, which involves frequent scanning and restructuring of various adjacency lists. Our next algorithm, CLUDE, takes advantage of the matrix cluster to determine a *universal symbolic sparsity pattern* (USSP). As we will show later, a USSP of a cluster $\mathcal{C}$ covers all the symbolic sparsity patterns of the matrices in $\mathcal{C}$. We can thus build a *universal adjacent-lists structure* to be commonly used to store the LU factors of all matrices in $\mathcal{C}$. Since this universal structure is static, we avoid the expensive dynamic construction of individual matrix's list structure, leading to much savings in execution time.

Before we describe the details of CLUDE, let us first explain the idea of USSP and prove some of its properties.

DEFINITION 9. (UNIVERSAL SYMBOLIC SPARSITY PATTERN). *Consider a cluster $\mathcal{C}$. A set of matrix indices, S, is a USSP of $\mathcal{C}$ iff $\tilde{sp}(\boldsymbol{A}) \subseteq S$, $\forall \boldsymbol{A} \in \mathcal{C}$.*

Recall that for any matrix $\boldsymbol{A}$, the data structures for storing $\boldsymbol{A}$'s LU factors are determined by its symbolic sparsity pattern ($\tilde{sp}(\boldsymbol{A})$) (see Figure 4). In particular, a node is created in an adjacency list for each matrix index that is present in $\tilde{sp}(\boldsymbol{A})$. We can likewise derive the data structures from a USSP S of a cluster. Since $\tilde{sp}(\boldsymbol{A}) \subseteq S$, $\forall \boldsymbol{A} \in \mathcal{C}$, the data structures for $\boldsymbol{A}$ are substructures of those derived from S. Hence the structures for S can act as static structures with which the the LU factors of the matrices in $\boldsymbol{A}$ are computed. In the following, we show how to obtain a USSP for a cluster based on $\boldsymbol{A}_\cup$ (see Definition 7). First, we prove a *monotonicity property* given by the following lemma.

LEMMA 1. *Given two matrices $\boldsymbol{A}_a$ and $\boldsymbol{A}_b$,*

$$(sp(\boldsymbol{A}_a) \subseteq sp(\boldsymbol{A}_b)) \Rightarrow (\tilde{sp}(\boldsymbol{A}_a) \subseteq \tilde{sp}(\boldsymbol{A}_b)).$$

PROOF. Assume $sp(\boldsymbol{A}_a) \subseteq sp(\boldsymbol{A}_b)$ and $(u, v) \in \tilde{sp}(\boldsymbol{A}_a)$, it suffice to show that (u, v) is also in $\tilde{sp}(\boldsymbol{A}_b)$. First, from Equation 3, $sp(\boldsymbol{A}_b) \subseteq \tilde{sp}(\boldsymbol{A}_b)$. Hence, if $(u, v) \in sp(\boldsymbol{A}_b)$, then $(u, v) \in \tilde{sp}(\boldsymbol{A}_b)$. The only case left to be considered is $(u, v) \notin sp(\boldsymbol{A}_b)$. Since $sp(\boldsymbol{A}_a) \subseteq sp(\boldsymbol{A}_b)$, we have $(u, v) \notin sp(\boldsymbol{A}_a)$. Now, $((u, v) \in \tilde{sp}(\boldsymbol{A}_a)) \wedge ((u, v) \notin sp(\boldsymbol{A}_a)) \Rightarrow (u, v) \in fp(\boldsymbol{A}_a)$ (Equation 3), i.e., $\exists u_1, \ldots, u_k$, s.t. the three conditions listed in Equation 2 are satisfied. In particular,

$$(3) \quad (u, u_1), (u_i, u_{i+1}), (u_k, v) \in sp(\boldsymbol{A}_a) \; \forall 1 \leq i < k.$$

Since $sp(\boldsymbol{A}_a) \subseteq sp(\boldsymbol{A}_b)$, we have (u, u_1), (u_i, u_{i+1}), $(u_k, v) \in sp(\boldsymbol{A}_b)$ $\forall 1 \leq i < k$. And thus, $(u, v) \in fp(\boldsymbol{A}_b)$. Hence, $(u, v) \in \tilde{sp}(\boldsymbol{A}_b)$ (Equation 3). □

THEOREM 1. *Given a cluster $\mathcal{C} = \{\boldsymbol{A}_1, \ldots, \boldsymbol{A}_t\}$. Let $\boldsymbol{A}_\cup$ be the matrix as defined in Defintion 7. $\tilde{sp}(\boldsymbol{A}_\cup)$ is a USSP of $\mathcal{C}$.*

PROOF. $\forall \boldsymbol{A}_i \in \mathcal{C}$, we have $sp(\boldsymbol{A}_i) \subseteq sp(\boldsymbol{A}_\cup)$ (by Property 1), which implies $\tilde{sp}(\boldsymbol{A}_i) \subseteq \tilde{sp}(\boldsymbol{A}_\cup)$ (by Lemma 1). Hence, by Definition 9, $\tilde{sp}(\boldsymbol{A}_\cup)$ is a USSP of $\mathcal{C}$. □

$\tilde{sp}(\boldsymbol{A}_\cup)$ can be obtained by performing symbolic decomposition on $\boldsymbol{A}_\cup$ (see Section 2.3). After that, a static data

structure is derived from $\tilde{sp}(\boldsymbol{A}_\cup)$ on which Bennett's algorithm operates. To reduce the size of the structure and thus decomposition time, we precede the above steps by finding the Markowitz ordering of $\boldsymbol{A}_\cup$ and applying the ordering to $\boldsymbol{A}_\cup$ as well as all matrices in the cluster. Algorithm 3 shows the pseudo code of CLUDE.

Algorithm 3: CLUDE on one cluster.

```
Input  : A cluster C = {A_1, A_2, ..., A_t}
Output: Ordering and LU factors of A_i, for 1 ≤ i ≤ t
Construct A_∪ from ⋃_{i=1}^{t} sp(A_i) based on Definition 7
O_∪ ← O*(A_∪)
Apply symbolic decomposition on A_∪^{O_∪} to obtain s̃p(A_∪^{O_∪})
Create static structure from s̃p(A_∪^{O_∪}) for LU factors
O_1 ← O_∪
(L_1^{O_1}, U_1^{O_1}) ← LU decomposition on A_1^{O_1}
for i ← 2 to t do
    O_i ← O_∪
    ΔA ← A_i^{O_∪} − A_{i−1}^{O_∪}
    (L_i^{O_i}, U_i^{O_i}) ← Bennett(A_{i−1}^{O_∪}, ΔA, L_{i−1}^{O_∪}, U_{i−1}^{O_∪})
end
return {O_1, ..., O_t}, and {(L_1^{O_1}, U_1^{O_1}), ..., (L_t^{O_t}, U_t^{O_t})}
```

5. ALGORITHMS FOR LUDEM-QC

We extend our cluster-based algorithms CINC and CLUDE to solve the LUDEM-QC problem for which an additional quality constraint $ql(\mathcal{O}_i, \boldsymbol{A}_i) \leq \beta$ has to be enforced. The key to enforcing the quality constraint is to control the size of the cluster. The smaller the cluster is, the higher the chance that the orderings produced by CINC or CLUDE satisfy the quality constraint. In the extreme case, when each cluster contains just one matrix, the ordering given by CINC or CLUDE for the (lone) matrix in the cluster is just Markowitz. Hence, $ql(\mathcal{O}_i, \boldsymbol{A}_i) = 0$ and so the constraint is vacuously satisfied. In the following, we discuss how the clustering algorithm should be modified under CINC and CLUDE so that the quality constraint is enforced. We call this clustering β-clustering. In the following discussion, we describe how to construct the first cluster of the EMS. Subsequent clusters are done similarly.

[**β-clustering CINC version**] Given a cluster $\mathcal{C} = \{\boldsymbol{A}_1, \ldots, \boldsymbol{A}_t\}$, CINC uses the Markowitz ordering of the first matrix in the cluster $\mathcal{O}_1$ as the ordering of all the matrices in the cluster. As we attempt to expand the current cluster by adding a matrix $\boldsymbol{A}_{t+1}$ from the EMS, we evaluate the quality-loss $ql(\mathcal{O}_1, \boldsymbol{A}_{t+1})$. If the quality constraint is violated, we start constructing a new cluster. Essentially, we replace the α-boundedness condition in α-clustering by the β quality-constraint. Algorithm 4 shows the clustering algorithm.

[**β-clustering CLUDE version**] CLUDE uses the Markowitz ordering $\mathcal{O}_\cup$ of $\boldsymbol{A}_\cup$ as the ordering of the matrices in the cluster. Checking the quality constraint as we attempt to add $\boldsymbol{A}_{t+1}$ to the cluster is trickier than in the CINC's case. This is because adding $\boldsymbol{A}_{t+1}$ to $\mathcal{C}$ changes $\boldsymbol{A}_\cup$ and thus $\mathcal{O}_\cup$. Hence, the quality constraints on all the t matrices that are already in the cluster have to be re-evaluated. To speed up constraint checking, we take a shortcut. Note that the constraint on $\boldsymbol{A}_i \in \mathcal{C}$ is equivalent to $\phi_i : \{|\tilde{sp}(\boldsymbol{A}_i^{\mathcal{O}_\cup})| - |\tilde{sp}(\boldsymbol{A}_i^*)| \leq \beta \cdot |\tilde{sp}(\boldsymbol{A}_i^*)|\}$. Also from Property 1

Algorithm 4: β-clustering (CINC version).

```
Input  : EMS M = {A_1, A_2, ..., A_T}, quality
         requirement β
Output: Clusters {C_1, C_2, ..., C_j}
j ← 1; C_j ← {A_1}
O ← O*(A_1)
for i ← 2 to T do
    if |s̃p(A_i^O)| − |s̃p(A_i^*)| ≤ β · |s̃p(A_i^*)| then
        C_j ← C_j ∪ {A_i}
    else                    // start building the next cluster
        j ← j + 1; C_j ← {A_i}
        O ← O*(A_i)
    end
end
return {C_1, C_2, ..., C_j}
```

and Lemma 1, we have $|\tilde{sp}(\boldsymbol{A}_i^{\mathcal{O}_\cup})| \leq |\tilde{sp}(\boldsymbol{A}_\cup^{\mathcal{O}_\cup})|$. Therefore the constraint $\phi_\cup : \{|\tilde{sp}(\boldsymbol{A}_\cup^{\mathcal{O}_\cup})| - |\tilde{sp}(\boldsymbol{A}_i^*)| \leq \beta \cdot |\tilde{sp}(\boldsymbol{A}_i^*)|\}$ implies ϕ_i. Hence, as we attempt to add $\boldsymbol{A}_{t+1}$ to the current cluster, we only need to compute one $|\tilde{sp}(\boldsymbol{A}_\cup^{\mathcal{O}_\cup})|$ instead of t $|\tilde{sp}(\boldsymbol{A}_i^{\mathcal{O}_\cup})|$'s. Algorithm 5 shows this clustering algorithm.

Algorithm 5: β-clustering (CLUDE version).

```
Input  : EMS M = {A_1, A_2, ..., A_T}, quality
         requirement β
Output: Clusters {C_1, C_2, ..., C_j}
j ← 1; C_j ← {A_1}
for i ← 2 to T do
    Construct A_∪ from C_j ∪ {A_i} based on Definition 7
    O_∪ ← O*(A_∪)
    if ∀A_l ∈ C_j ∪ A_i, |s̃p(A_∪^{O_∪})| − |s̃p(A_l^*)| ≤ β · |s̃p(A_l^*)|
    then
        C_j ← C_j ∪ {A_i}
    else                    // start building the next cluster
        j ← j + 1; C_j ← {A_i}
    end
end
return {C_1, C_2, ..., C_j}
```

6. EXPERIMENTAL EVALUATION

We conduct experiments to evaluate the algorithms INC, CINC, and CLUDE. We execute BF to obtain baseline performance numbers against which the other algorithms are evaluated. In particular, we execute BF to determine the Markowitz ordering of each matrix in the EMS to measure the quality-loss of the orderings given by other algorithms. Also, the execution times of the other algorithms are expressed as speedup factors over BF's execution time. All algorithms are implemented in Java and the experiments are conducted on a Linux machine with a 3.40GHz Octo-Core Intel(R) processor and 16GB of memory.

We conduct experiments on two EMS's that are derived from two real datasets[5] and also on a synthetic EMS. Here we briefly describe the datasets.

[Wiki] We collected a set of 1000 daily snapshots of 20,000 Wikipedia pages and their hyperlinks. The number of hyperlinks in the first and the last snapshots are 56,181 and 138,072, respectively. The average (mes) similarity (Eq. 5)

[5]http://socialnetworks.mpi-sws.org/, http://dblp.uni-trier.de/xml/.

between successive matrices derived from the snapshots is 99.88%.

[DBLP] The DBLP dataset consists of 70 years of publications. We extracted all publications in three areas (1) DB, (2) Vision, (3) Algorithms & Theory. Based on these publications, we constructed a sequence of co-authorship graphs. The snapshot graph of a date is derived from all the papers published before that date[6]. We used the latest 1000 daily snapshots for our experiments. There are 97,931 vertices; the number of edges in the first and the last snapshots are 387,960 and 547,164, respectively. The average similarity between successive matrices derived from the snapshots is 99.86%.

[Synthetic] We generated synthetic EGS's from which EMS's are derived. Our EGS generator takes five parameters (their default values are shown in parentheses):

- V (50,000): the number of vertices.
- $|EP|$ (450,000): the number of edges in an "edge pool" EP.
- d (5): the average vertex degree of the first snapshot.
- k (4): the ratio $\Delta E^+/\Delta E^-$, where ΔE^+ and ΔE^- are the number of edges added to and removed from a snapshot to generate the next snapshot, respectively.
- ΔE (500): $\Delta E^+ + \Delta E^-$.
- T (500): the number of snapshots in the EGS.

To generate an EGS, we first use the BA model [4] to generate a scale-free[7] base graph $\mathcal{G}$ that has V vertices and $|EP|$ edges. All the edges are collected in the edge pool EP. Next, we randomly pick $d \cdot V$ edges from EP to form the edge set E of the first snapshot. Then we repeat the following procedure to generate subsequent snapshot graphs:

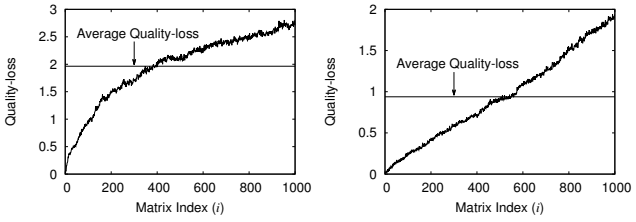
1. Randomly remove $\Delta E^- = \Delta E/(k+1)$ edges from E.
2. Randomly pick $\Delta E^+ = (k \cdot \Delta E)/(k+1)$ edges from $EP - E$ and add them to E.

We can prove that the snapshot graphs generated by the above procedure are scale-free. We omit the proof due to space limitation.

6.1 Ordering Quality Analysis

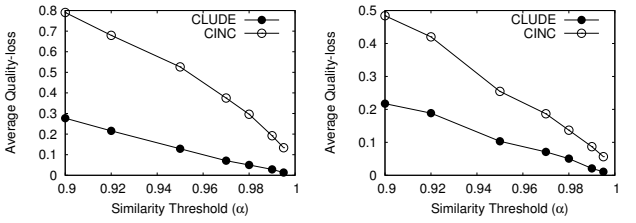
Our first set of experiments evaluate the algorithms in terms of their ordering qualities. Recall that INC finds the Markowitz ordering, $\mathcal{O}^*(\boldsymbol{A}_1)$, of $\boldsymbol{A}_1$ and applies that to all matrices $\boldsymbol{A}_i$'s in the whole EMS. The ordering quality degrades with i as $\boldsymbol{A}_i$ gradually deviates from $\boldsymbol{A}_1$. Figure 5 shows the quality-loss $ql(\mathcal{O}^*(\boldsymbol{A}_1), \boldsymbol{A}_i)$ vs. the matrix index i for the two real datasets. We see that the quality-loss increases with i as explained. Indeed, the ordering quality of INC is quite poor. For Wiki, the average quality-loss (over the 1000 matrices) is about 2. That means if a matrix $\boldsymbol{A}_i$ is ordered by $\mathcal{O}^*(\boldsymbol{A}_1)$, on average, the number of "extra" entries in $\boldsymbol{A}_i$'s LU factors is *twice* the size of $\boldsymbol{A}_i$'s LU factors if $\boldsymbol{A}_i$ were Markowitz-ordered! The quality-loss reaches 2.7 for the last snapshot of the EMS.

By grouping similar matrices into a cluster and applying the same ordering only to matrices of the same cluster, CINC and CLUDE give much better ordering qualities. Figure 6 shows the average quality-loss of the orderings given by CINC and CLUDE as the α-clustering similarity threshold varies. A larger α implies a more stringent similarity requirement and thus clusters are more *compact*. It is thus easier for the same ordering to cover all the matrices in the cluster and yet it gives good ordering quality. This explains why quality-loss drops as α increases. Comparing CINC and CLUDE, CLUDE gives much better ordering qualities. This is because while CINC uses the Markowitz ordering of the first matrix in the cluster, CLUDE uses the Markowitz ordering of $\boldsymbol{A}_\cup$, which covers *all* matrices in the cluster and thus fits them better. For example, for the Wiki dataset, when $\alpha = 0.95$, the quality-losses of CINC and CLUDE are 0.53 and 0.13, respectively. Compared with the average value 2 for INC, the quality-loss of CLUDE is 15 times better than that of INC.



(a) WIKIPEDIA (b) DBLP

Figure 5: INC: quality-loss vs. matrix index (i).



(a) WIKIPEDIA (b) DBLP

Figure 6: Average quality-loss vs. similarity threshold α.

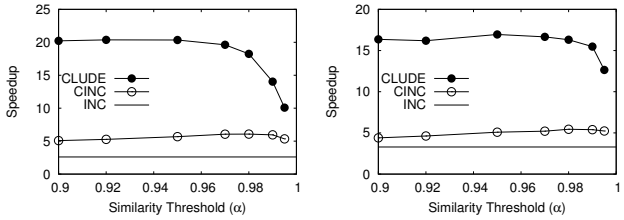
6.2 Efficiency Analysis

In this section, we compare the algorithms in terms of speed. We express algorithms' efficiency in terms of their speedup factors over BF's execution time. Figure 7 shows the speedups as α varies. Note that INC does not cluster the matrices and so its speedup is shown as straight lines in the graphs. From the figure, we see that among the three algorithms, INC is the slowest while CLUDE is the fastest. This is despite the fact that INC determines only one Markowitz ordering (on $\boldsymbol{A}_1$), performs only one full LU decomposition (on $\boldsymbol{A}_1$) and applies (the supposedly) fast Bennett's algorithm to incrementally LU decompose all the other matrices in the EMS.

The reason why INC is slow (only 2.6 times faster than BF for the Wiki dataset) is due to its poor ordering quality. As we have explained, the Markowitz ordering of $\boldsymbol{A}_1$ is unfit for most of the other matrices in the EMS. Hence, the LU factors computed by INC are huge. This significantly slows down the incremental decompositions (Bennett's). The speedups of CINC are generally above 5 for

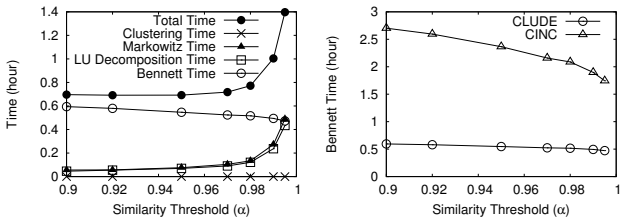
[6]For publications that only have publication year, we evenly distribute them to the dates of their corresponding years.

[7]A graph is scale-free if the distribution of vertices' degrees follows a power law: $P(t) \propto 1/t^\gamma$, where $P(t)$ is the probability that a vertex has a degree t, and γ is a constant. Following [4], we set $\gamma = 3$.



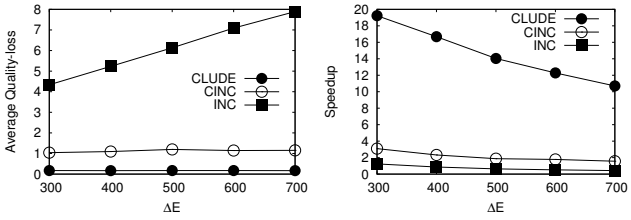
(a) WIKIPEDIA (b) DBLP

Figure 7: Speedup vs. similarity threshold α.



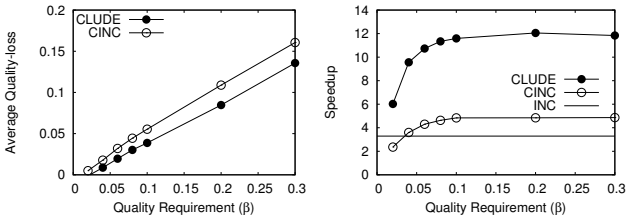
(a) Execution time (CLUDE) (b) Bennett time

Figure 8: CLUDE's execution time breakdown (Wiki dataset).



(a) Average quality-loss (b) Speedup

Figure 9: Varying ΔE (SYNTHETIC).



(a) Average quality-loss (b) Speedup

Figure 10: Varying quality requirement β (DBLP).

the Wiki set and CLUDE registers a speedup of 20. These significant speedups are brought about by their much higher ordering qualities. From Figure 7, we see how the performances of CINC and CLUDE change with α. In particular, their speedups drop when α is very close to 1. This is because a very large α value implies a very stringent clustering requirement. In the extreme case, when α is very large, each cluster contains only one matrix, which reduces CINC and CLUDE to BF. We observe that the speedups of CINC and CLUDE are very significant and quite stable unless α is very large. We remark that selecting the threshold α is an engineering effort as its best value depends on various factors such as the nature of the graphs. Fortunately, it is not very critical that the optimal α be found, as the algorithms perform very well over a wide range of α.

We further investigate the reasons behind the big performance gap between CINC and CLUDE as shown by their speedup curves. There are two factors that contribute to the improvement of CLUDE over CINC: (1) CLUDE gives better ordering quality than CINC, which leads to smaller LU factors and thus faster decomposition time (full or incremental). (2) CLUDE uses the universal symbolic sparsity pattern to prepare the data structures for storing matrices' LU factors. This greatly facilitates the incremental updating of the LU factors across matrices (see discussion in Section 4). Both of these factors improve the speed of Bennett's algorithm, which incrementally decompose matrices.

CLUDE's execution time consists of four components: (1) Clustering time (t_c): time to perform α-clustering on the EMS. (2) Markowitz time (t_M): time to compute the Markowitz orderings of matrices (done once per cluster). (3) LU decomposition time (t_d): time to perform full LU decompositions (done once per cluster on the first matrix of the cluster). (4) Bennett's time (t_B): time to perform incremental LU decompositions (done on all matrices but the first of each cluster). Figure 8(a) shows these four components when CLUDE is applied to the Wiki dataset over different α values.

First, we see that t_c is negligible and stays constant. Second, we note that as α increases, fewer matrices are collected in a cluster without violating the similarity constraint. Hence, clusters are smaller and there are more clusters. Consequently, t_M and t_d increase with α. Third, tighter clustering implies better ordering quality (see Figure 6(a)), which speeds up incremental decomposition. Therefore, t_B drops as α increases. Now, let us focus on the numbers when $\alpha = 0.95$, which is the case when CLUDE gives the best speedup. We see that t_B dominates CLUDE's execution time. In fact, t_B is also the dominating component of CINC's execution time. Figure 8(b) gives a head-to-head comparison between the t_B components of CINC and CLUDE. We see that CLUDE significantly outperforms CINC in t_B by the two factors mentioned above. This explains the big gap between their execution times.

6.3 Synthetic Dataset

Our next experiments evaluate the algorithms using the synthetic dataset. The synthetic dataset allows us to vary the various properties of the graphs (matrices) so that we can perform various sensitivity studies. Figures 9(a) and (b) compare the algorithms in terms of quality and speedup, respectively, as the number of edge changes between snapshot (ΔE) varies. Note that a larger ΔE causes the matrices in the EMS deviate more from $\boldsymbol{A}_1$. This makes INC's ordering more unfit for the matrices, leading to worse ordering quality. CINC and CLUDE, on the other hand, are very adaptive. Through α-clustering, they maintain the similarity of the matrices in the same cluster (by including more or fewer matrices in a cluster) and thus their ordering qualities remain stable as ΔE changes. However, faster evolving matrices means more and smaller clusters. This increases t_M and t_d. Also, a larger ΔE makes incremental decomposition slower, which increases t_B. Hence the algorithms'

speedups drop when ΔE increases. We remark that CLUDE gives very impressive speedups (10-20) compared with others (Figure 9(b)).

We have conducted many other experiments with the synthetic dataset varying the various parameters of the synthetic data generators. The general observations from these results are that CLUDE gives the best ordering quality and at the same time is much faster than INC and CINC. CLUDE typically registers a speedup from 10 to 20. Due to space limitations, we omit those results in the paper.

6.4 The LUDEM-QC Problem

Our last set of experiments compare the performance of CINC and CLUDE in solving the LUDEM-QC problem. Recall that the problem can be efficiently solved for symmetric matrices. Hence, we conducted the experiments on the DBLP dataset, whose matrices are symmetric. Figures 10(a) and 10(b) show the qualities and speedups of the algorithms as the quality requirement β varies.

From Figure 10(a), we see that both CINC and CLUDE are adaptive to β. In particular, when the requirement is looser (a larger β), the algorithms employ bigger clusters so that they can perform fewer full decompositions but more incremental decompositions. The result is trading ordering quality (increasing quality-loss, Figure 10(a)) for faster decomposition (increasing speedup, Figure 10(b)). We observe that both CINC and CLUDE are able to maintain an ordering quality that is well within the requirement. Between the two, CLUDE gives higher ordering quality. Again, this is because it uses the ordering of $\boldsymbol{A}_{\cup}$, which covers all the matrices in the same cluster. Moreover, CLUDE can provide more than 10 times speedup. It significantly outperforms the other algorithms.

7. CASE STUDY

To further illustrate the use of evaluating measures in a graph sequence, we conducted a case study on a Patent dataset [15]. This dataset contains information (e.g., patent name, year granted, company, etc.) of 3 million U.S. patents and the citations among them between 1975 and 1999. Analyzing the citations among patents can help us answer questions such as "How does company X depend on company Y in technology development?" "How does the dependency evolve over time?" These insights are useful in predicting new alliances and acquisitions, which have much impact on the companies' stock prices. We use IBM as an example subject of analysis.

We take the yearly snapshots of the patent citation graphs spanning 1979 to 1999. Based on a citation graph, we measure the proximity of company Y from company X by summing the PPR scores of Y's patent nodes using X's patent nodes as the set of starting seed nodes.

Taking IBM as company X, Figure 11 shows the proximity of a few representative companies from IBM over the years from 1979 to 1999. In the figure, we show the ranks of the companies based on their proximity scores. The figure reflects how much IBM depended on other companies in its technology development. For example, Xerox developed Alto (widely regarded as the first PC) and invented the Graphical User Interface (GUI), which are important components of IBM PC's development. Xerox thus maintained a high rank during those 20 years.

Among the seven companies shown in Figure 11, Harris, an international telecommunications equipment company, stands out. While the ranks of others were quite stable, Harris' rank increased steadily since 1979. This trend is a good predictor of a closer collaboration between the companies. In fact, in 1992, IBM and Harris announced their alliance to share technology and to capitalize their strengths in technology development. Harris' stock price hit a closing high shortly after the announcement. This case study shows that the trends of various graph measures over a graph sequence could provide interesting insights that are beyond what measures from a single graph can derive.

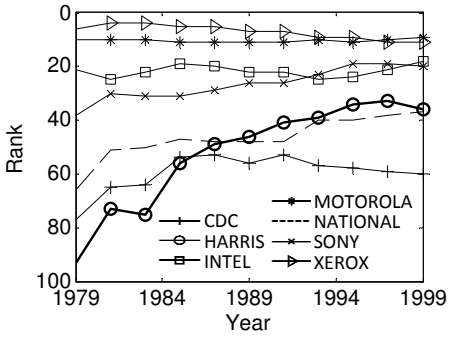


Figure 11: PPR score rankings (IBM patents as seed nodes).

8. RELATED WORK

EGS processing was first introduced in [25], which studies the computation of the shortest path distance between two nodes across a graph sequence. Our clustering approach shares some favor with that presented in [25].

There are a number of studies on efficient computation of the various measures, such as PR/SALSA/PPR/DHT/RWR, on single graphs [22, 18, 12, 14, 23, 10, 11]. One interesting approach is approximation methods. Two such popular methods are the *power iteration* (PI) method [6] and the *Monte Carlo* (MC) method [10]. For example, to compute RWR scores, PI iteratively refines the solution $\boldsymbol{x}$ based on the recurrence relation $\boldsymbol{x}_u^{(k+1)} = d\boldsymbol{W}\boldsymbol{x}_u^{(k)} + (1-d)\boldsymbol{q}_u$ and MC simulates random walks to approximate the stationary distributions. PI or MC have to be executed once for every input query q_u. In contrast, our problem is to decompose a matrix such that queries can be answered very efficiently. For example, with our Wiki dataset, answering queries after matrices are LU-decomposed is about two orders of magnitude faster than answering them using either PI or MC.

There are fast solutions for answering some very specific queries that exploit matrix sparsity [11, 12]. For example, in [11], sparse matrix decomposition is used to find the top-k nodes of the highest RWR scores in a graph. These studies focus on processing single graphs. Instead, our work focuses on processing graph sequences and to answer queries which involve general Gaussian Elimination.

There are algorithms (e.g.,[8, 3]) for incrementally maintaining specific measures when the underlying graph changes. For example, [3] employs the MC method and stores a number of random walk segments (RWS's) in a database. When the graph changes, the stored RWS's are updated accordingly. PPR scores are then approximated based on the stored RWS's. Our algorithms compute exact measures in-

stead of approximation and they are not restricted to a specific measure. Also, after matrices are LU-decomposed, query answering can be done much faster than those incremental measure maintenance solutions. For example, with our Wiki dataset, our approach is at least an order of magnitude faster.

In recent years, there are works on processing graph streams [19, 21]. Their focus is on how to detect interesting changes and how to perform fast aggregations as graphs arrive. For example, [19] studies how to detect sub-graphs that change rapidly over a small window of the stream. Like other studies on stream processing, the data (graphs) that arrives in the stream is not archived. This limits the kind of analyses that can be performed. In contrast, we focus on decomposing the matrices in a graph sequence such that more complex analytical tasks can be done efficiently.

9. CONCLUSIONS

In this paper we studied the LUDEM problem and its quality-constraint variant LUDEM-QC. We illustrated that by decomposing the matrices in an EMS into their LU factors, interesting structural analyses on a sequence of evolving graphs can be carried out efficiently. We gave an in-depth discussion on matrix reordering and incremental LU decomposition, based on which we designed our solutions for the LUDEM problem. Through extensive experiments, we analyzed our algorithms and showed that CLUDE outperformed the rest. Over a wide range of settings, CLUDE significantly outperformed the straightforward incremental algorithm (INC) both in terms of ordering quality and speed. Typically, CLUDE's quality-loss was more than 10 times smaller than that of INC. Also, CLUDE registered a speedup that in most cases was at least an order of magnitude faster than the brute-force approach.

Acknowledgment

This research is partly supported by Hong Kong Research Grants Council grant HKU712712E, HKU711309E, and the University of Hong Kong (Project 201211159083). We would like to thank the reviewers for their insightful comments.

10. REFERENCES

[1] P. R. Amestoy, T. A. Davis, and I. S. Duff. An approximate minimum degree ordering algorithm. *SIAM J. Matrix Anal. Appl.*, 17(4):886–905, Oct. 1996.

[2] R. Andersen, F. Chung, and K. Lang. Local graph partitioning using pagerank vectors. In *FOCS '06*, pages 475–486, Washington, DC, USA, 2006. IEEE Computer Society.

[3] B. Bahmani, A. Chowdhury, and A. Goel. Fast incremental and personalized pagerank. *Proc. Very Large Data Base Endow.*, 4(3):173–184, Dec. 2010.

[4] A.-L. Barabĺćsi and R. Albert. Emergence of scaling in random networks. *Science*, 286(5439):509–512, 1999.

[5] J. Bennett. Triangular factors of modified matrices. *Numerische Mathematik*, 7(3):217–221, 1965.

[6] P. Berkhin. A survey on pagerank computing. *Internet Mathematics*, 2:73–120, 2005.

[7] G. E. Box, G. M. Jenkins, and G. C. Reinsel. *Time series analysis: forecasting and control*. Wiley. com, 2013.

[8] P. K. Desikan, N. Pathak, J. Srivastava, and V. Kumar. Incremental page rank computation on evolving graphs. In *WWW (Special interest tracks and posters)*, pages 1094–1095, 2005.

[9] I. S. Duff, A. M. Erisman, and J. K. Reid. *Direct methods for sparse matrices*. Oxford University Press, Inc., New York, NY, USA, 1986.

[10] D. Fogaras and B. Rácz. Towards scaling fully personalized pagerank. In *WAW*, pages 105–117. 2004.

[11] Y. Fujiwara, M. Nakatsuji, M. Onizuka, and M. Kitsuregawa. Fast and exact top-k search for random walk with restart. *Proc. Very Large Data Base*, 5(5):442–453, 2012.

[12] Y. Fujiwara, M. Nakatsuji, T. Yamamuro, H. Shiokawa, and M. Onizuka. Efficient personalized pagerank with accuracy assurance. In *KDD*, pages 15–23, 2012.

[13] A. George, M. Heath, J. Liu, and E. Ng. Solution of sparse positive definite systems on a hypercube. *Journal of Computational and Applied Mathematics*, 27(1-2):129–156, 1989.

[14] Z. Guan, J. Wu, Q. Zhang, A. K. Singh, and X. Yan. Assessing and ranking structural correlations in graphs. In *ACM SIGMOD Conference*, pages 937–948, 2011.

[15] B. Hall, A. Jaffe, and M. Trajtenberg. The NBER patent citation data file: Lessons, insights and methodological tools. Technical report, NBER, 2001.

[16] Z. Huang and D. K. J. Lin. The time-series link prediction problem with applications in communication surveillance. *INFORMS Journal on Computing*, 21(2):286–303, 2009.

[17] A. N. Langville and C. D. Meyer. *Google's PageRank and beyond: The science of search engine rankings*. Princeton University Press, 2006.

[18] R. Lempel and S. Moran. Salsa: the stochastic approach for link-structure analysis. *ACM Trans. Inf. Syst.*, 19(2):131–160, 2001.

[19] Z. Liu and J. X. Yu. Discovering burst areas in fast evolving graphs. In *Database Systems for Advanced Applications (1)*, pages 171–185, 2010.

[20] H. M. Markowitz. The elimination form of the inverse and its application to linear programming. *Management Science*, 1957.

[21] A. McGregor. Graph mining on streams. In *Encyclopedia of Database Systems*, pages 1271–1275, 2009.

[22] L. Page, S. Brin, R. Motwani, and T. Winograd. The pagerank citation ranking: bringing order to the web. 1999.

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

[24] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery. *Numerical Recipes 3rd Edition: The Art of Scientific Computing*. 2007.

[25] C. Ren, E. Lo, B. Kao, X. Zhu, and R. Cheng. On querying historical evolving graph sequences. *Proc. Very Large Data Base*, 4(11):726–737, 2011.

[26] D. J. Rose and R. E. Tarjan. Algorithmic aspects of vertex elimination. In *STOC*, pages 245–254, 1975.