Fully Dynamic Quasi-Biclique Edge Covers via Boolean Matrix Factorizations

Pauli Miettinen
Max-Planck-Institut für Informatik
Saarbrücken, Germany
pauli.miettinen@mpi-inf.mpg.de

ABSTRACT

An important way of summarizing a bipartite graph is to give a set of (quasi-) bicliques that contain (almost) all of its edges. These quasi-bicliques are somewhat similar to clustering of the nodes, giving sets of similar nodes. Unlike clustering, however, the quasi-bicliques are not required to partition the nodes, allowing greater flexibility when creating them. When we identify the bipartite graph with its bi-adjacency matrix, the problem of finding these quasi-bicliques turns into the problem of finding the Boolean matrix factorization of the bi-adjacency matrix – a problem that has received increasing research interest in data mining in recent years. But many real-world graphs are dynamic and evolve over time. How can we update our bicliques without having to re-compute them from the scratch?

An algorithm was recently proposed for this task (Miettinen, ICMD 2012). The algorithm, however, is only able to handle the case where the new 1s are added to the matrix – it cannot handle the removal of existing 1s. Furthermore, the algorithm cannot adjust the rank of the factorization.

This paper extends said algorithm with the capability of working in fully dynamic setting (with both additions and deletions) and with capability of adjusting its rank dynamically, as well. The behaviour and performance of the algorithm is studied in experiments conducted with both real-world and synthetic data.

Categories and Subject Descriptors

H.2.8 [**Database Management**]: Database Applications—*data mining*

1. INTRODUCTION

A binary matrix can always be identified with a bipartite graph by considering it as a bi-adjacency matrix. If the binary matrix is square, it can also be identified with a directed graph; if it is symmetric, it can be identified with an undirected graph. Many graph mining problems can then be represented as problems on the corresponding binary matrices. This paper focuses on bipartite graphs (i.e. general binary matrices), and in there, on the special problem on *quasi-biclique edge covers*. Informally, given a bipartite graph G, the goal is to find a set of quasi-bicliques that contain almost all of the edges of G and are not too sparse (a *quasi-biclique* is an almost-complete bipartite graph; the sparser it is, the further it is from being complete).

Why quasi-biclique coverings? In short, a quasi-biclique edge cover is a powerful method to summarize and analyse the graph based on the shared neighbours of nodes. Consider, for example, a bipartite graph of users and web pages, with an edge between a user and web page if the user has 'liked' the web page. A quasi-biclique in such a graph gives us a set of users that like similar web pages, and a set of web pages that are liked by similar people.

The above example also highlights an important feature of many such real-world graphs: they are constantly evolving as users 'like' new web pages, new users and web pages get added to the system, and old users and web pages are removed. The problem considered in this paper is how to dynamically update given quasi-biclique edge covering when edges or nodes are added or removed. We will also consider the problem of how to adjust the size of the cover (i.e. the number of quasi-bicliques) dynamically. It should be emphasized that the goal is *not* to predict which edges will appear in the future (the *link prediction* problem); rather, the goal is to have a good cover at any given point of time. Therefore, say, overfitting is not a problem per se, as long as the algorithm is able to adjust the cover when it sees new edges.

We will consider the quasi-biclique covering problem using its equivalent representation as *Boolean matrix factorization* (BMF) problem for the bi-adjacency matrix. The algorithm presented in this paper is an extension of a recent algorithm for dynamic Boolean matrix factorizations [10]. The algorithm of [10] has two shortcomings: 1) edges and nodes can only be added, never removed and 2) it cannot dynamically adjust the size of the covering (rank of the factorization). This paper will address both of these shortcomings.

The next section will give the formal definitions of the problems this paper considers. Section 3 gives a brief introduction to the existing algorithm before explaining the extensions. The experiments are reported in Section 4, followed by related work and conclusions.

2. BACKGROUND AND DEFINITIONS

Before defining the dynamic problems studied in this paper, we will first explain the notation and define the offline versions of the dynamic problems.

DyNetMM'13, June 23, 2013, New York, New York, USA

2.1 Notation

We identify bipartite graphs as binary matrices. Matrices are denoted by upper-case bold letters ($\boldsymbol{A}$). Vectors are lower-case bold letters ($\boldsymbol{a}$). If $\boldsymbol{A}$ is an n-by-m binary matrix, $|\boldsymbol{A}|$ denotes the number of 1s in it, i.e. $|\boldsymbol{A}| = \sum_{i,j} a_{ij}$. We extend the same notation to binary vectors.

Let $\boldsymbol{X}$ and $\boldsymbol{Y}$ be n-by-m binary matrices. We have the following element-wise matrix operations: The *Boolean sum* $\boldsymbol{X} \vee \boldsymbol{Y}$ is the normal matrix sum with addition defined as $1 + 1 = 1$. The *Boolean subtraction* $\boldsymbol{X} \ominus \boldsymbol{Y}$ is the normal element-wise subtraction with $0 - 1 = 0$. Notice that this does not define an inverse of Boolean sum, as $1 + 1 - 1 = 0$. The *Boolean element-wise product* $\boldsymbol{X} \wedge \boldsymbol{Y}$ is defined as normal element-wise matrix product. The *exclusive or* $\boldsymbol{X} \oplus \boldsymbol{Y}$ is the normal matrix sum with addition defined as $1 + 1 = 0$ (i.e. addition is done over the field $\mathbb{Z}_2$). Furthermore, if $\boldsymbol{Z} \in \{-1, 0, 1\}^{n \times m}$, we define $\boldsymbol{X} \boxplus \boldsymbol{Z}$ to be the normal element-wise addition except that $1 + 1 = 1$ and $0 - 1 = 0$. The complement of $\boldsymbol{X}$ is denoted by $\bar{\boldsymbol{X}}$.

Let $\boldsymbol{X}$ be n-by-k and $\boldsymbol{Y}$ be k-by-m binary matrices. Their *Boolean matrix product*, $\boldsymbol{X} \circ \boldsymbol{Y}$, is the binary matrix $\boldsymbol{Z}$ with $z_{ij} = \bigvee_{l=1}^{k} x_{il} y_{lj}$, that is, Boolean matrix product is the normal matrix product using the Boolean addition.

The *Boolean rank* of an n-by-m binary matrix $\boldsymbol{A}$, $\operatorname{rank}_B(\boldsymbol{A})$, is the least integer k such that there exists an n-by-k binary matrix $\boldsymbol{B}$ and a k-by-m binary matrix $\boldsymbol{C}$ for which $\boldsymbol{A} = \boldsymbol{B} \circ \boldsymbol{C}$. Matrices $\boldsymbol{B}$ and $\boldsymbol{C}$ are the *factor matrices* of $\boldsymbol{A}$, and the pair $(\boldsymbol{B}, \boldsymbol{C})$ is the (exact) *Boolean factorization* of $\boldsymbol{A}$. If $\boldsymbol{A} \neq \boldsymbol{B} \circ \boldsymbol{C}$, the factorization is approximate. The columns of $\boldsymbol{B}$ and rows of $\boldsymbol{C}$ are the *factor vectors* (*factors* for short); the fth pair of factor vectors (fth column of $\boldsymbol{B}$ and fth row of $\boldsymbol{C}$) is denoted $\boldsymbol{b}_f$ and $\boldsymbol{c}_f$.

If $p = (i, j)$ is a pair of nonnegative integers and $\boldsymbol{A} = (a_{ij})$ is an n-by-m binary matrix (with $i \leq n$ and $j \leq m$), we write $p \in \boldsymbol{A}$ if $a_{ij} = 1$ and $p \notin \boldsymbol{A}$ otherwise. If $p \in \boldsymbol{A}$, we say that $\boldsymbol{A}$ *covers* element p.

Let $\mathtt{s} = (s_1, s_2, \ldots)$ be an ordered sequence of arbitrary items. We use the following slice notation: $\mathtt{s}(1 : n)$ stands for the first n elements of $\mathtt{s}$ while $\mathtt{s}(n :)$ stands for the elements of $\mathtt{s}$ from element n onwards, including n.

Now let $\mathtt{s}$ be an ordered sequence of k tuples from $\mathbb{N} \times \mathbb{N} \times \{+, -\}$, $\mathtt{s} = \big((i_1, j_1, \pm), (i_2, j_2, \pm), \ldots, (i_k, j_k, \pm)\big)$. We define operator $\mathcal{M}_{n \times m}(\mathtt{s})$ to produce an n-by-m matrix $\boldsymbol{M} = (m_{ij})$ such that

$$m_{ij} = \begin{cases} 1 & \text{if } (i, j, +) \in \mathtt{s} \\ -1 & \text{if } (i, j, -) \in \mathtt{s} \\ 0 & \text{otherwise.} \end{cases}$$

We omit the subscript of $\mathcal{M}$ when it is clear from the context.

2.2 Background on Quasi-Biclique Covers and Boolean Matrix Factorizations

Before discussing the dynamic problems, let us consider the standard offline problems. The problem of finding a quasi-biclique cover of size k is formally as follows.

PROBLEM 2.1. *Given a bipartite graph $G = (U \cup V, E)$ and an integer k, find k pairs (B_i, C_i), where $B_i \subseteq U$ and $C_i \subseteq V$ for all $i = 1, \ldots, k$, such that the pairs minimize*

$$\left| E \triangle \left(\bigcup_{i=1}^{k} \tilde{E}_i \right) \right| , \tag{1}$$

where $\triangle$ is the symmetric difference between two sets (i.e. $A \triangle B = (A \setminus B) \cup (B \setminus A)$) and $\tilde{E}_i = \{e = \{b, c\} : b \in B_i \text{ and } c \in C_i\}$, that is, $\tilde{E}_i$ contains the edges of the complete bipartite subgraph between B_i and C_i.

As explained in the introduction, this paper mainly works with an alternative formulation of Problem 2.1, namely, the problem of Boolean matrix factorization, defined as follows:

PROBLEM 2.2 (BMF). *Given an n-by-m binary matrix $\boldsymbol{A}$ and integer k, find an n-by-k binary matrix $\boldsymbol{B}$ and a k-by-m binary matrix $\boldsymbol{C}$ such that $\boldsymbol{B}$ and $\boldsymbol{C}$ minimize*

$$|\boldsymbol{A} \oplus (\boldsymbol{B} \circ \boldsymbol{C})| . \tag{2}$$

That Problems 2.1 and 2.2 are equivalent is easy to see. We can identify the bipartite graph G with its bi-adjacency matrix $\boldsymbol{A}$ (a bi-adjacency matrix has $|U|$ rows and $|V|$ columns and has 1 in position (i, j) if E contains an edge between u_i and v_j). The ith column of $\boldsymbol{B}$ and ith row of $\boldsymbol{C}$ give the incidence vectors for sets B_i and C_i, respectively, and their outer product gives the bi-adjacency matrix of the biclique $\tilde{E}_i$. The Boolean matrix product $\boldsymbol{B} \circ \boldsymbol{C}$ gives the incidence matrix of the graph $\bigcup_{i=1}^{k} \tilde{E}_i$, and finally, the modulo-2 summation $\boldsymbol{A} \oplus (\boldsymbol{B} \circ \boldsymbol{C})$ is equivalent to the symmetric difference between E and $\bigcup_{i=1}^{k} \tilde{E}_i$.

This change of context from graphs to matrices furnishes us with a good number of existing research. For example, it is known that finding the matrices $\boldsymbol{B}$ and $\boldsymbol{C}$ is NP-hard even to approximate [11] and that given matrices $\boldsymbol{A}$ and $\boldsymbol{B}$, finding $\boldsymbol{C}$ that minimizes (2) is also NP-hard even to approximate well [7].

2.3 The Problems

Informally, the *Fully Dynamic Boolean Matrix Factorization* problem (FDBMF) asks us to keep up a factorization of changing data that provides a good approximation at any time. Equivalently, we are asked to keep up a collection of bicliques that approximate well the set of edges in our bipartite matrix. Formally, FDBMF is defined as follows:

PROBLEM 2.3 (FDBMF). *Given an n-by-m binary matrix $\boldsymbol{A}$, its (approximate) rank-k Boolean factorization $(\boldsymbol{B}, \boldsymbol{C})$, and, for any given time t, a prefix $\mathtt{s}(1 : t)$ of an unknown sequence $\mathtt{s} \in \mathbb{N} \times \mathbb{N} \times \{+, -\}$, find a rank-$k$ Boolean factorization $(\boldsymbol{B}_t, \boldsymbol{C}_t)$ that minimizes*

$$\big| \big(\boldsymbol{A} \boxplus \mathcal{M}(\mathtt{s}(1 : t))\big) \oplus (\boldsymbol{B}_t \circ \boldsymbol{C}_t) \big| . \tag{3}$$

It is worth to emphasize again that in FDBMF, the goal is not to predict where the 1s will appear, but to have a good factorization at any given time t.

In the definition of Problem 2.4, both the size of the data matrix and the rank of the factorization are fixed. Both constraints can be relaxed, though. Adding or removing new rows and columns to the data matrix is straight forward, and all algorithms we are going to present can handle that. Allowing the rank of the decomposition to change, however, is a more complex issue. Given that we do not try to predict anything (and thence do not have overfitting issues), if the algorithm is allowed to adjust the rank, it can represent any given matrix exactly using $\min\{n, m\}$ factors. Therefore, minimizing the error (3) cannot be our goal if we let the rank undefined.

This problem can be solved by using the *minimum description length* (MDL) principle [16]: instead of minimizing the error, the goal is to minimize the number of bits it takes to represent the matrix $\boldsymbol{A} \boxplus \mathcal{M}(\mathtt{s}(1:t))$ using the factorization $(\boldsymbol{B}_t, \boldsymbol{C}_t)$. To this end, we need to encode three binary matrices, $\boldsymbol{B}_t$, $\boldsymbol{C}_t$ (the factorization), and $\big(\boldsymbol{A} \boxplus \mathcal{M}(\mathtt{s}(1:t))\big) \oplus (\boldsymbol{B}_t \circ \boldsymbol{C}_t)$ (the error). Denote the total encoding length of these three matrices by $L_t(\boldsymbol{A}, \mathtt{s}(1:t), \boldsymbol{B}_t, \boldsymbol{C}_t)$. Actual encoding schemes for BMF have been studied earlier [12, 13], and we will use the method that was considered the best: the *Typed XOR Data-to-Model* encoding [13]. Taking L_t to refer to that encoding model, we can define the problem of *Dynamic-Rank Boolean Matrix Factorization* (DRBMF) as:

PROBLEM 2.4 (DRBMF). *Given an n-by-m binary matrix $\boldsymbol{A}$, its (approximate) rank-k Boolean factorization $(\boldsymbol{B}, \boldsymbol{C})$, and, for any given time t, a prefix $\mathtt{s}(1:t)$ of an unknown sequence $\mathtt{s}$, find a Boolean factorization $(\boldsymbol{B}_t, \boldsymbol{C}_t)$ of any rank that minimizes*

$$L_t(\boldsymbol{A}, \mathtt{s}(1:t), \boldsymbol{B}_t, \boldsymbol{C}_t)\,. \tag{4}$$

3. ALGORITHMS

This section will present the extensions to allow the earlier algorithm of [10] to handle the removal of 1s and dynamic adjustments of the rank. We will start by presenting the algorithm of [10] before explaining how to extend it.

3.1 Algorithm for Additive DBMF

The algorithm of [10] works only in the *additive dynamic BMF* setting, i.e. when it is only possible to add 1s, rows and columns. Here we will give a brief outline of the algorithm; for more information, see [10].

The basic algorithm in [10] is *online*, that is, it can only add 1s in the factor matrices, and can never remove them. When a 1 is added that is not yet covered by the factorization, the algorithm tries to find a factor to extend to cover the added 1. If no factor can be extended without increasing the error, the 1 is left uncovered. Otherwise, the factor that decreases the reconstruction error the most is extended to cover it. If a factor is extended, the algorithm tries to extend it into other rows and columns iteratively, until it cannot be extended anymore.

Whether the factor can be extended is defined by the function cover. If $\boldsymbol{A}$ is the data matrix, $(\boldsymbol{B}, \boldsymbol{C})$ the factorization before the extension, and $(\boldsymbol{B}', \boldsymbol{C}')$ the factorization after the extension, the cover function is defined as

$$\begin{aligned}\mathsf{cover}(\boldsymbol{A}, \boldsymbol{B}, \boldsymbol{C}, \boldsymbol{B}', \boldsymbol{C}') = \\ \big|\boldsymbol{A} \wedge \big((\boldsymbol{B}' \circ \boldsymbol{C}') \ominus (\boldsymbol{B} \circ \boldsymbol{C})\big)\big| \\ - \big|\big(\bar{\boldsymbol{A}} \ominus (\boldsymbol{B} \circ \boldsymbol{C})\big) \wedge \big(\boldsymbol{B}' \circ \boldsymbol{C}'\big)\big|\,.\end{aligned} \tag{5}$$

That is, cover measures how many not-yet-covered 1s the extended factor would cover, minus the number of uncovered 0s that are covered by the extended factor.

To allow the removal of unnecessary 1s from the factors, it is proposed in [10] to use iterative updates on the factor matrices: $\boldsymbol{B}'$ is set to $\boldsymbol{B}$ and $\boldsymbol{C}'$ is find such that it minimizes (5). The updated $\boldsymbol{C}$ is then held fixed and $\boldsymbol{B}'$ minimizing (5) is found. This is continued until there is no improvement on the error.

3.2 Extension to Removals

In case a 1 is removed from the matrix, there are three different cases we need to consider. Let (i, j) be the element that is turned from 1 to 0. The first case is that $(i, j) \notin \boldsymbol{B} \circ \boldsymbol{C}$. In this case, we do not need to do anything, as the error can only decrease. The second case is when row i and column j become empty after the deletion. In this case, we remove all 1s in the corresponding row and column in the factor matrices. If only either a row or a column becomes empty, we remove all 1s in the corresponding row/column from the factor matrices, respectively, but continue to the next case.

The third case is the one where all the work is. In this case we need to decide if we can remove row i or column j from *any* factor currently containing it. If (i, j) is only covered by single factor f, this is easy: we first try to set $b_{if} = 0$ and see if that improves the error, and then to set $c_{fj} = 0$ and see if that improves the error (with $b_{if} = 1$). We do either (or both) of the changes if they reduce the error. But it can be that (i, j) is covered by multiple factors. To properly test which factors should be changed, we should try every possible combination of them. But obviously, that quickly becomes infeasible. Rather, we will check for every factor independently whether we can remove the rows or columns from it (keeping all other factors unchanged). Then we will edit those factors for which the change yield smaller reconstruction error.

After we have edited a factor (by removing a row or a column), we will see if we can extend it to new rows or columns. Assume we removed row i_t from the factor f. We now see if this smaller f can be used in a new column. For this, we study all columns that are not yet included in f (i.e. columns j for which $c_{fj} = 0$) and see if including some column would yield non-negative change in the cover function. After that, we use similar technique to check if we can extend the (possibly extended) factor to new rows. We continue to alternatively extending the factor to new columns and rows until there are no new columns or rows that would give non-negative cover value. The process is guaranteed to end as in each iteration we reduce the error by at least 1.

A pseudo-code for the extension is presented as Algorithm 1. This algorithm is called for every removal of index pair (i_t, j_t) and the result is used as the input for the next call. In case the edit is an addition, Algorithm 1 from [10] is used.

3.3 Dynamically Adjusting the Rank

To adjust the rank, we have two options: we can either remove existing factors or add new ones. We will try to do them both, finding what would be the best factor to remove, and what would be the best factor to add. We then commit the better of these two actions, provided that it reduces the description length, and re-compute the best addition and deletion. This is repeated until neither the addition of a new factor nor the deletion of an old one improves the description length anymore.

Computing what would be the description length after removing factor f is straight forward: we just remove the factor and re-compute the description length (taking into account the new factorization and new error). We try this for every factor, and select the best as a candidate factor to be deleted.

But how to decide what factor to add? Unlike with deletions, we cannot just look into some existing list of factors,

Algorithm 1 Removing elements in FDBMF

Input: A binary matrix $\boldsymbol{A}$, its rank-k Boolean factorization $(\boldsymbol{B}, \boldsymbol{C})$, and index pair (i_t, j_t).
Output: Updated factorization $(\boldsymbol{B}_t, \boldsymbol{C}_t)$.

```
1: function UpdateAfterRemoval(A, B, C, (i_t, j_t))
2:     if (i_t, j_t) ∉ B ∘ C then
3:         return B and C
4:     end if
5:     if a_{i_t j} = 0 for all j ≠ j_t then
6:         set b_{i_t f} = 0 for all f
7:     end if
8:     if a_{i j_t} = 0 for all i ≠ i_t then
9:         set c_{f j_t} = 0 for all f
10:    end if
11:    for all factors f s.t. b_{i_t f} c_{f j_t} = 1 do
12:        if cover improves after setting b_{i_t f} or c_{f j_t} to 0 then
13:            add f to the list of factors to be edited
14:        end if
15:    end for
16:    for all factors f to be edited do
17:        edit f so as to maximize the cover
18:    end for
19:    for all factors f that were edited do
20:        repeat
21:            try to extend f to new rows and columns
22:        until no new extensions are possible
23:    end for
24:    return B_t and C_t
25: end function
```

but we have to generate a new factor from the scratch. We do that using the following heuristic: We consider each column of $\boldsymbol{A} \ominus (\boldsymbol{B} \circ \boldsymbol{C})$, that is, columns of 1s that we have not yet covered. Each of them is a candidate to be added to $\boldsymbol{B}$. To compute the corresponding row of $\boldsymbol{C}$, we see in which columns of $\boldsymbol{A}$ the candidate column has a positive cover value, and include those columns. This gives us m candidate factors (with possible repetitions). To select from these, we could re-compute the description length for each candidate. This, however, is expensive, as we need to compute the new error for each of the candidates. Rather, we select the factor with the largest area ($|\boldsymbol{b}| \cdot |\boldsymbol{c}|$) as our candidate factor to add, and only compute the description length for that.

There are some decisions in this heuristic that need to be discussed. First, why use the columns of $\boldsymbol{A} \ominus (\boldsymbol{B} \circ \boldsymbol{C})$? The motivation is that we want to cover uncovered 1s: any part of the new factor that overlaps with existing factors will not reduce the description length. The new factor can overlap with the existing ones, though, as we select the columns in which it appears based on the original data, not the reduced one. The other question is, why consider only the largest-area factor? The way how the new factors are build (using $\boldsymbol{A} \ominus (\boldsymbol{B} \circ \boldsymbol{C})$ and cover) ensures that they will never increase the error. So the factor with the largest area covers the largest number of 1s, which we consider a desirable feature.

As updating the rank is rarely meaningful after every edit, we only check the description length after user-defined number of changes. Further, we can heuristically decide that if the description length is going down, we do not need to do anything to the rank; we only see if we can make any changes if the description length has increased.

3.4 Time Complexity

There are some implementation details that greatly affect the (practical) time and space complexity of the algorithm. The first is how the matrices are stored. Most (or all) real-world applications to FDBMF use very sparse data, so the matrices (data and factor) should be stored using some sparse representation. But due to the nature of the dynamic algorithms, all matrices will be changed during the computation, and the sparse representation should make the updates reasonably effective. For our implementation, we selected a hybrid of list-of-lists and dictionary-of-keys: each column is represented by a set of indices, and the sets are stored in a list. The sets of indices can be implement using standard techniques, for example, cuckoo hashing [15] giving amortized constant addition and deletion times and constant query time.

Assuming we store the product $\boldsymbol{B} \circ \boldsymbol{C}$, we can handle the first three if-clauses of Algorithm 1 in $O(k)$ time. The for-loop of line 11 takes at most k iterations, each iteration requiring to re-compute the cover for the factor. But we only need to compute the cover for the rows and columns the factor contains, taking time $O(|\boldsymbol{b}_f| + |\boldsymbol{c}_f|)$ for factor f, and thus the overall time of the for-loop is $O(|\boldsymbol{B}| + |\boldsymbol{C}|)$.

Extending the factors can be very costly, though. For n-by-m data matrix, computing the cover-values for factor f and each data column can take $O(m\,|\boldsymbol{b}_f|)$ time. As this computation might need to be done after (almost) each edition, and for rows too, we need to find a way to make it faster. To that end, we do some more clever bookkeeping. When initializing the algorithm, we compute and store the cover values for each factor and each row and column of the data not included in the factor. Along these values, we store the timestamp when we computed the value and we order them in decreasing order by the cover value.

At each time step the cover value can change by at most 1 for the rows and columns that were not included in or excluded from some factor. When considering a factor, we traverse the list of pre-computed cover values, starting from the largest one. If c is the value stored and Δt is the difference in time between when the value was computed and the current time, we consider this column (or row) only if $c + \Delta t \geq 0$. If the condition holds, we compute the actual cover value: if it is non-negative, the column (or row) is added into the factor; if it is negative, the value along with the updated timestamp is put back to the list. After we see the first column (or row) for which $c + \Delta t < 0$, we know that no more columns (rows) can have non-negative cover value. After the first iteration, the list is kept sorted by $c + \Delta t$.

The only complication for this scheme comes from those rows and columns that were included in or excluded from a factor. We keep a list of those, and when considering a factor, first check if we need to re-compute the cover value for some of those rows and columns. If so, we do that first, and return the updated values back to the list with updated timestamps.

At first glance it might look like keeping the list sorted is going to be very expensive ($O(m \log m)$ for columns). Note, however, that we only need to move the elements for which we re-compute the cover. As moving these elements in a pre-ordered list is (in practice) much faster than $O(m)$ (they rarely move to the very end of the list), this bookkeeping actually saves us considerable time. The list of cover-values is re-computed after every change in the rank and every iterative update of the factor matrices.

Updating the rank is more expensive operation. Computing the description length takes time

$$O(|\boldsymbol{B}| + |\boldsymbol{C}| + \max\{|\boldsymbol{A}|, |\boldsymbol{B} \circ \boldsymbol{C}|\}),$$

which is the same time considering the removal and addition of a factor takes. The slowest part here, then, is the recomputation of the cover-values.

Finally, it should be noted that many parts of the algorithm are embarrassingly parallel. For example, whenever the cover-values need to be computed for multiple rows or columns (generally the slowest part of the algorithm), it can be done in parallel.

4. EXPERIMENTAL EVALUATION

We tested our algorithm using both synthetic and real-world data. Before going to the results, we present the algorithms and error measures we used in the experiments.

4.1 Algorithms and Error Measures

The experiments were conducted using the algorithm of [10] extended in the aforementioned ways. This algorithm is called `Dynamic`. For some experiments, the factors were iteratively updated after a fixed number of edits (see [10] for details); this method is called '`Dynamic` w/ `iter`'. To compute the initial solution, the `Asso` algorithm [11] was used, as it generally was the best-performing algorithm in the experiments of [10]. With synthetic data we also know the factor matrices used to create the initial matrix $\boldsymbol{A}$. In these cases, `Opt` refers to the method that uses these factors as the initial solution. The dynamic rank adjustment was only used when specifically mentioned.

To measure the quality of the factorizations we used the simple *reconstruction error* (3) at the end of the input sequence. In addition, we also computed the *relative error*: if e_i is the error caused by the initialization, e_f is the final error, and $|\mathtt{s}|$ the length of the input sequence, the relative error is defined as $(e_f - e_i)/|\mathtt{s}|$. That is, the relative error explains how much error we do, on average, per each edit.

We also computed the offline factorization, that is, the factorization of the data after the full sequence of additions has been applied. We used this to compute the *empirical competitive factor*: the reconstruction error of the dynamic method divided by the reconstruction error of the offline method. Note that we cannot compute the true competitive factor as even the offline version of the problem is NP-hard.

The presented algorithms were implemented using Matlab and Python and the source code together with the synthetic data generators is freely available for research purposes.[1]

4.2 Synthetic Data

The purpose of these experiments is to test the effects various data characteristics have on the algorithms in a controlled manner. We studied three characteristics: the Boolean rank of the factorization, the density of the data, and the amount of data revealed at the initialization phase. All synthetic matrices were 500-by-700 and for each data point, we generated 10 random binary matrices with identical parameters. In figures, we report the mean over these 10 matrices as well as the standard deviation. The rank parameter was set to the correct rank of the synthetic data.

Rank. The rank of the matrices varied from 5 to 25 in steps of 5. The data was generated so that the expected density of the data matrix was 10% and the edit sequence had 50 additions followed by 50 removals. The results are in Figure 1(a).

[1] `http://www.mpi-inf.mpg.de/~pmiettin/dbmf/`

Generally speaking, the higher the rank, the better the algorithm does. This is contrary to normal BMF (see, e.g., [11]), where synthetic data with higher rank usually yields to worse results. Here the reason for improvement with higher rank is probably that the larger number of factors lets the algorithm adjust itself better.

In the relative error (results omitted due to space constraints), the method based on `Asso` initialization always gave smaller error than the `Opt` initialization, meaning that with `Asso`, the error did not increase that much. Further, the relative error was mostly negative for the `Asso` initialization, indicating that the dynamic method in fact *improved* the reconstruction accuracy compared to the initial solution.

Density. This data had rank 20 and the edit sequence had 50 additions followed by 50 deletions. The density of the data is varied, from 1% to 30%. In the results (Figure 1(b)), the behaviour is similar, albeit reversed, of Figure 1(a): `Opt` is somewhat better than `Dynamic`, and both method's results decrease with increased density.

Sequence length. Here all matrices had rank 20 and expected density of 10%. The edit sequence contained either 25, 50, or 75 additions followed by the same number of deletions. The results (Figure 1(c)) mirror the other results: `Opt` is better than `Dynamic`, but both behave similarly.

4.3 Real-World Data

4.3.1 Data Sets

For real-world experiments, we used three timestamped data sets from the HetRec 2011 collection.[2]

I. The `Delicious` data[3] contains information about which bookmarks users have tagged. As each user can tag a bookmark with multiple tags, we only considered the first tag each user gave to each bookmark. After removing all bookmarks with less than five tags and all users with less than 3 bookmarks tagged, we were left with a 1053-by-1203 binary matrix with 7 717 ones (density 0.6%).

II. The `LastFM` data[4] contains information about which artist which user has tagged. We again removed repeated tags and artists with less than five tags and users with less than five tagged artists. This left us with a 1348-by-3708 binary matrix with 53 676 ones (density 1%).

III. The `Movielens` data[5] contains information about which user has rated which movie. We removed movies with less than ten reviews and users with less than five movies reviewed to get a 2113-by-6829 matrix with 841 910 ones (density 5.8%).

All elements in these data sets come with a timestamp. To generate the initial matrix, we took the first two-thirds of the elements. The edit sequence contained the deletion of the oldest one-third of the elements randomly mixed with the addition of the youngest one-third of the elements. This data generation process was used in oder to simulate the 'decay', where the older ratings/likings are ignored while new ones are being added.

When iterative update of the factors was used (see [10]), it was triggered after every 1 000 (for `Delicious`), 10 000 (for `LastFM`), or 100 000 (for `Movielens`) edits.

[2] `http://www.grouplens.org/node/462`

[3] `http://www.delicious.com`

[4] `http://www.lastfm.com`

[5] `www.grouplens.org`, `www.imdb.com`, and `http://www.rottentomatoes.com`

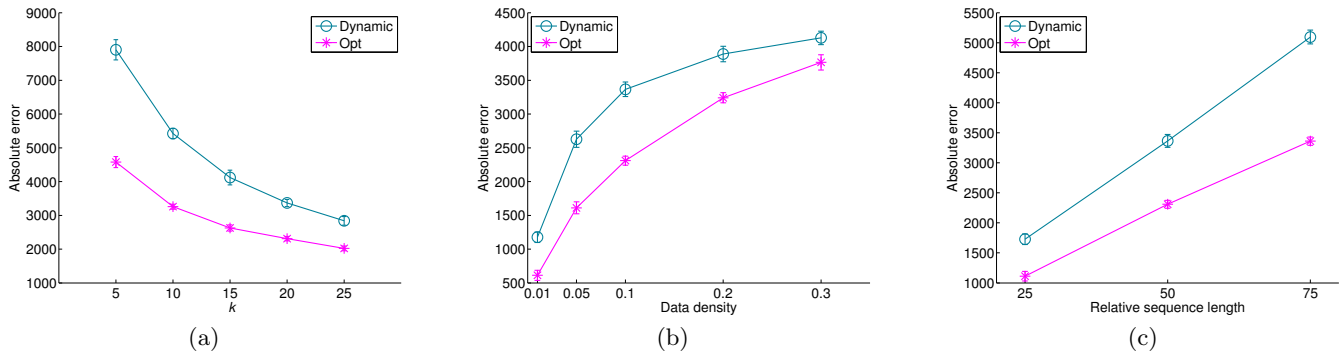

Figure 1: Results for synthetic data using absolute errors and two different methods for finding the initial factorization. (a) Using different rank k. (b) Using different data density. (c) Using different length of online sequence. The markers show the mean over ten random samples and the width of the error bars is twice the standard deviation.

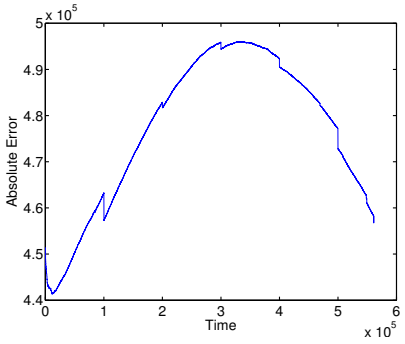

Figure 2: The behaviour of the error in each iteration for `Movielens` data. The initial factorization is computed using `Asso` and the factors are iteratively updated in every 100 000 additions.

4.3.2 Behaviour of the Error over Time

We start by examining how the reconstruction error behaves as the data is changed over time. For this, we report the results with `Movielens` data using `Asso` as the initialization method and updating the factors iteratively after every 100 000 edits. The results are in Figure 2.

The sharp drops in Figure 2 are due to the iterative updates. The other improvements are a combination of the algorithm adapting better to the data and the changes removing the 1s that were not covered with the algorithm. Overall, even at its peak, the error is no more than about 10% higher than the initial error, signaling that the algorithm is able to adjust well to the changing data. The reasons for the hill-shape of the curve are not obvious, but it is possible that the underlying latent factors change over time: at the first half of the updates the algorithm is still adapting the factorization, and after about 300 000 updates, the factorization is properly adapted, and the subsequent updates fit well on the found factorization.

4.3.3 Reconstruction Error and Time

The main result here is the reconstruction error with the real-world data sets, presented in Table 1. Three methods were used: the first was the standard FDBMF algorithm (`Dynamic`), the second used iterative updates to update the factors in regular intervals, and the third was the comparison point, the offline BMF algorithm. In addition, we also compared how well the initial factors represented the final matrix. Both the initial solution and the offline solution were computed using the `Asso` algorithm.

Table 1: Results for real-wold data sets, absolute error. `Dynamic` is the FDBMF algorithm presented here, 'w/ `iter`' means it uses the iterative updates presented in [10]. `Offline` uses `Asso` to solve the BMF in the final matrix and '`Initial`' refers to using the initial factors as the factorization of the final matrix.

	Data set		
Algorithm	`Delicious`	`LastFM`	`Movielens`
`Dynamic`	4 792	30 488	467 729
w/ `iter`	4 755	29 829	456 777
`Offline`	4 601	27 252	472 611
`Initial`	5 414	37 123	685 765

The Table 1 shows clearly that even without the iterative updates, the dynamic algorithm is competitive with the offline algorithm (the competitive factor is never more than 12%). With the `Movielens` data the dynamic algorithm is in fact *better* than the offline algorithm. Unsurprisingly, the iterative updates always improve the reconstruction error, but the effect is mostly very minor.

The iterative updates do have a major effect in one measure: the running time. (The timing information is in Table 2.) The iterative method is always at least an order of magnitude slower than the normal version (a similar observation was also made in [10]). But the running times of the standard FDBMF algorithm (`Dynamic`) and the offline algorithm (here, `Asso`) are roughly equal (this does not include the time it takes to compute the initial factorization).

4.3.4 Adjusting the Rank

We end with the results concerning the DRBMF algorithm. We computed the results using the `Delicious` data, and testing the change in the description length after every 100 changes. If the description length had increased by more than 0.1%, the procedure to update the rank was initiated. We also computed the offline rank and description lengths,

Table 2: Running times in seconds with different real-world data sets

Algorithm	Data set		
	Delicious	LastFM	Movielens
Dynamic	4	213	4 452
w/iter	585	1 504	11 295
Offline	43	200	4 210

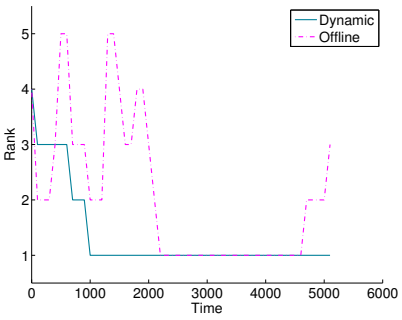

Figure 3: The ranks of dynamic and offline factorizations for the Delicious data computed after every 100 changes.

taking the matrix after every 100 changes and computing the MDL-optimal factorization using the algorithm from [13]. The ranks of the factorizations can be seen in Figure 3.

The two algorithms do not constantly agree with each other. Over time, the dynamic algorithm reduces the rank while the offline algorithm changes it more often. That the dynamic algorithm behaves more smoothly is to be expected, as it cannot compute the factorization from the scratch. After about 2 000 edits, the offline algorithm agrees with the dynamic algorithm that the data has Boolean rank 1, and the two agree almost until the end.

The data, then, has extremely low rank. Why is that? Looking at the factorization it seems obvious that the algorithms (both online and offline) consider the data having only very little structure. Such low ranks are not universal, though. The MDL-optimal rank of the LastFM data, for example, is over 300.

The ranks alone do not tell the whole story, however. Per the MDL principle, the factorization of the data that obtains the smaller description length is the optimal. The description lengths for the algorithms are presented in Figure 4.

From Figure 4 we see that while the offline algorithm starts with much lower encoding length, it quickly increases to the same level as the dynamic algorithm, with the dynamic algorithm typically being slightly better. This is again an interesting behaviour. The offline algorithm from [13] is a heuristic, so it is not guaranteed to give optimal results. Yet, as it can compute its factorization from the scratch, it should be able to obtain at least as good results as the dynamic method. In this light, the dynamic method's performance seems very strong.

In experiments done with the other data sets (results omitted), the algorithm mostly kept the rank untouched.

4.3.5 Conclusions

Overall, the results with real-world data are very good. The absolute reconstruction errors might look high, but compared to the error caused by the offline method – the only reasonable comparison point – they are very competitive. That a dynamic method is better than the comparable offline method is rather surprising, but as we have seen, with the heuristics involved here, it sometimes is the case.[6] The algorithm also behaves very well when the it is allowed to adjust the rank. Furthermore, its running time is essentially equivalent to that of the offline algorithm *for a single factorization*. Given that the initial factors clearly do not work well in the dynamic setting, the proposed algorithm is the fastest (and sometimes the most accurate) method for having constantly a good factorization.

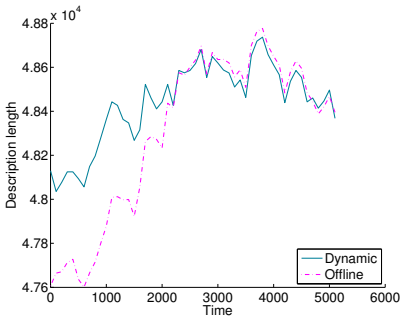

Figure 4: Description lengths of the dynamic and offline factorizations of the Delicious data.

5. RELATED WORK

Boolean matrix factorizations have gained interest in data mining community during the past few years. The use of Boolean matrix factorizations in data mining was proposed in [11], although related concepts, such as tiles and formal concepts, were studied much earlier. *Tiling* a database [6] refers to the task of covering all 1s of a binary matrix using few[7] itemsets. The Boolean matrix factorization can be seen as a generalization of this task, each rank-1 binary matrix defining a 'tile'. The difference is that tiling does not allow any 0s to be represented as 1s, whereas the Boolean matrix factorization allows this type of errors. Before that, Boolean matrix factorizations were mostly studied by combinatorics; see [14] and references therein. For some applications and variations of Boolean matrix factorizations, see [8].

Boolean matrix factorization is not the only type of matrix factorization dealing with binary matrices. Methods using normal algebra [19] or probabilistic modeling [2, 18], for example, have been proposed. The characteristics and behaviour of such methods are very different to Boolean matrix factorization, though.

Extending a matrix factorization is a common problem in Information Retrieval (IR) when latent factor models, such as Latent Semantic Indexing [3], are used. These models represent the given corpus as a (non-negative) matrix, and

[6]Obviously, we can use the dynamic algorithm in offline setting to obtain better offline algorithm. But as it requires the initial factorization, the benefits are less obvious in purely offline situation.

[7]When the goal is to cover all 1s and minimize the number of tiles, it is equivalent to computing the Boolean rank [9]; when the number of tiles is given and the goal is to minimize the number of uncovered 1s, the problem is more akin to standard Boolean matrix factorization.

apply a factorization on it. When a new document arrives to the corpus, it has to be *fold in*. The folding-in is performed by projecting the document vector into the lower-dimensional latent factor space (e.g. by multiplying it with the inverse of one of the factor matrices). As noted earlier, this folding-in is an NP-hard problem with the Boolean matrix factorization.

Recently Saha and Sindhwani [17] proposed an algorithm for dynamic non-negative matrix factorization. Their method, as most of those in IR, allows adding new rows and columns (typically, terms and documents), but not changing the already-observed values. This restriction makes sense in the framework of IR, as the contents of the documents rarely gets changed. Our setup, however, asks specifically for handling the changes in the already-factored part of the matrix.

Boolean matrix factorization can be seen as a type of relaxed (bi-) clustering under the Boolean algebra (see [11]), and previous methods have been proposed for dynamic clustering (see, for example, [4] and citations therein). The relaxation BMF does, however, seems so fundamental that the methods proposed in that line of work do not seem to be applicable in dynamic BMF.

Link prediction is the problem of predicting which edges are added to the graph in the future. As we have emphasized, our setting differs from that of link prediction as we do not aim at predicting anything. Moreover, a typical link prediction algorithm works with undirected graphs (i.e. symmetric binary matrices), though methods for handling bipartite graphs do also exist (e.g. [1, 5]).

Finally, the work most related to the present one is of course [10].

6. CONCLUSIONS

We have presented algorithms for doing fully dynamic Boolean matrix factorizations with dynamic ranks. This problem is equivalent to finding a quasi-biclique edge covering a bipartite graph, and can be used in many graph analysis tasks. As already the offline version of the BMF problem is NP-hard even to approximate well, our algorithm is naturally a heuristic. Yet, our tests with real-world data show that the dynamic and online algorithms are, not only as fast as, but in some cases more accurate than their offline counterparts. The ability to dynamically adjust the rank allows the fast dynamic algorithm to be used longer before the factorization has to be re-computed from the scratch.

In terms of binary matrices, we have considered the most general case, corresponding to bipartite graphs. If the original graph is undirected, however, the adjacency matrix is more restricted. Could this fact be used to improve the presented algorithm's performance with undirected graphs is an interesting question for future research.

The algorithm presented here is fully sequential. This will naturally reduce its scalability to large graphs. As mentioned above, many parts of the algorithm are embarrassingly parallel, though, so a shared-memory parallel implementation is straight forward. Whether this or similar algorithm can be efficiently implemented in a distributed setting is again a problem for future work.

7. REFERENCES

[1] N. Benchettara, R. Kanawati, and C. Rouveirol. Supervised machine learning applied to link prediction in bipartite social networks. In *ASONAM '10*, pages 326–330. IEEE, 2010.

[2] E. Bingham, A. Kabán, and M. Fortelius. The aspect Bernoulli model: multiple causes of presences and absences. *Pattern Anal. Appl.*, 12(1):55–78, 2009.

[3] S. C. Deerwester et al. Indexing by latent semantic analysis. *J. Am. Soc. Inform. Sci.*, 41(6):391–407, 1990.

[4] D. Duan, Y. Li, R. Li, and Z. Lu. Incremental K-clique clustering in dynamic social networks. *Artif. Intell. Rev.*, May 2011.

[5] D. Dunlavy, T. G. Kolda, and E. Acar. Temporal Link Prediction Using Matrix and Tensor Factorizations. *ACM TKDD*, 5(2), 2011.

[6] F. Geerts, B. Goethals, and T. Mielikäinen. Tiling databases. In *DS '04*, pages 77–122, 2004.

[7] P. Miettinen. On the positive-negative partial set cover problem. *Information Processing Letters*, 108(4):219–221, 2008.

[8] P. Miettinen. *Matrix Decomposition Methods for Data Mining: Computational Complexity and Algorithms*. PhD thesis, Department of Computer Science, University of Helsinki, 2009.

[9] P. Miettinen. Sparse Boolean Matrix Factorizations. In *ICDM '10*, pages 935–940, 2010.

[10] P. Miettinen. Dynamic Boolean Matrix Factorizations. In *ICDM '12*, pages 519–528, Dec. 2012.

[11] P. Miettinen, T. Mielikäinen, A. Gionis, G. Das, and H. Mannila. The Discrete Basis Problem. *IEEE TKDE*, 20(10):1348–1362, Oct. 2008.

[12] P. Miettinen and J. Vreeken. Model Order Selection for Boolean Matrix Factorization. In *KDD '11*, pages 51–59, 2011.

[13] P. Miettinen and J. Vreeken. MDL4BMF: Minimum Description Length for Boolean Matrix Factorization. Technical Report MPI-I-2012-5-001, Max-Planck-Institut für Informatik, June 2012.

[14] S. D. Monson, N. J. Pullman, and R. Rees. A Survey of Clique and Biclique Coverings and Factorizations of (0, 1)-Matrices. *Bull. ICA*, 14:17–86, 1995.

[15] R. Pagh and F. F. Rodler. Cuckoo hashing. *J. Algorithm*, 51(2):122–144, 2004.

[16] J. Rissanen. Modeling by shortest data description. *Automatica*, 14(5):465–471, Sept. 1978.

[17] A. Saha and V. Sindhwani. Learning evolving and emerging topics in social media: A dynamic NMF approach with temporal regularization. In *WSDM '12*, pages 693–702, 2012.

[18] A. Streich, M. Frank, D. Basin, and J. M. Buhmann. Multi-assignment clustering for Boolean data. In *ICML '09*, 2009.

[19] Z.-Y. Zhang, T. Li, C. Ding, X.-W. Ren, and X.-S. Zhang. Binary matrix factorization for analyzing gene expression data. *Data Min. Knowl. Discov.*, 20(1):28–52, 2010.