Biclustering and Boolean Matrix Factorization in Data Streams

Stefan Neumann
University of Vienna
Faculty of Computer Science
Vienna, Austria
stefan.neumann@univie.ac.at

Pauli Miettinen
University of Eastern Finland
School of Computing
Kuopio, Finland
pauli.miettinen@uef.fi

ABSTRACT

We study clustering of bipartite graphs and Boolean matrix factorization in data streams. We consider a streaming setting in which the vertices from the *left* side of the graph arrive one by one together with all of their incident edges. We provide an algorithm which after one pass over the stream recovers the set of clusters on the *right* side of the graph using sublinear space; to the best of our knowledge this is the first algorithm with this property. We also show that after a second pass over the stream the left clusters of the bipartite graph can be recovered and we show how to extend our algorithm to solve the Boolean matrix factorization problem (by exploiting the correspondence of Boolean matrices and bipartite graphs). We evaluate an implementation of the algorithm on synthetic data and on real-world data. On real-world datasets the algorithm is orders of magnitudes faster than a static baseline algorithm while providing quality results within a factor 2 of the baseline algorithm. Our algorithm scales linearly in the number of edges in the graph. Finally, we analyze the algorithm theoretically and provide sufficient conditions under which the algorithm recovers a set of planted clusters under a standard random graph model.

PVLDB Reference Format:
Stefan Neumann, Pauli Miettinen. Biclustering and Boolean Matrix Factorization in Data Streams. *PVLDB*, 13(10): 1709-1722, 2020.
DOI: https://doi.org/10.14778/3401960.3401968

1. INTRODUCTION

Bipartite graphs appear in many areas in which interactions of objects from two different domains are observed. Hence, finding interesting clusters (also called communities) in bipartite graphs is a fundamental and well-researched problem with many applications; this problem is often called *biclustering*. For example, in social networks the two domains could be users and hashtags and an interaction corresponds to a user using a certain hashtag; finding clusters in such a graph corresponds to finding groups of hashtags used by the same users and groups of users using the same hashtags [41]. Biclustering has many applications across many domains such as computational biology [15, 27], text mining [14] and finance [20].

Many real-world bipartite graphs have three natural properties. First, the numbers of vertices on both sides of the graphs are very large, while at the same time their density is extremely low, i.e., the graphs are very sparse. For example, consider a bipartite graph consisting of users on the left side of the graph and movies on the right side of the graph, where an edge indicates that a user rated a movie. Such graphs often consist of millions of users and movies, but the average degree is constant. Second, the degrees on one side of the graph are usually bounded by a small constant, while on the other side of the graph a few vertices have extremely large degrees. Continuing the example from above, note that users usually do not rate more than 1000 movies, but a small number of popular movies is rated by millions of users. The third property is that the clusters on the high-degree side of the graph are usually relatively small. Again continuing the above example, those groups of movies which are watched by the same users typically do not consist of more than 50 movies.

Furthermore, for many real-world bipartite graphs, it is natural to assume that the left-side vertices appear in a data stream, for example, in Natural Language Processing [17], market basket analysis, network traffic analysis and stock price analysis [10]. For instance, in market basket analysis the left-side vertices correspond to transactions in a supermarket and incident edges indicate which products were bought. To efficiently find interesting clusters in such datasets, we need to develop streaming algorithms for the biclustering problem. Another motivation to study the streaming setting is that current static algorithms do not scale to the previously mentioned real-world graphs with millions of vertices, because these methods have prohibitively high memory consumptions and running times. Streaming algorithms could mitigate this issue due to improved memory efficiency and speed.

Our Contributions. In this paper, we address this question and provide the first streaming algorithms for the biclustering problem. In particular, we study a streaming setting in which the vertices from the left side of the bipartite graph arrive one after another, together with all of their incident edges. Then after a single pass over the stream, the algorithm must output the set of right-side clusters of the graph. The algorithm is then allowed a second pass over the stream in order to output the left-side clusters. See Section 2.1 for the formal definition of the problem.

This work is licensed under the Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License. To view a copy of this license, visit http://creativecommons.org/licenses/by-nc-nd/4.0/. For any use beyond those covered by this license, obtain permission by emailing info@vldb.org. Copyright is held by the owner/author(s). Publication rights licensed to the VLDB Endowment.
Proceedings of the VLDB Endowment, Vol. 13, No. 10
ISSN 2150-8097.
DOI: https://doi.org/10.14778/3401960.3401968

To obtain our algorithms, we heavily exploit the previously mentioned properties of real-world bipartite graphs. Formally, we assume that there exists a number s such that the degree of all left-side vertices and the size of all right-side clusters is at most s. This implies that, in total, the graph contains $O(ms)$ edges, where m is the number of vertices on the left side of the graph.

We introduce the sofa algorithm which returns the right-side clusters of the graph after a single pass over the stream and using sublinear memory. To the best our of knowledge, sofa is the first algorithm with this property. The running time of sofa is $O(ms{\cdot}k \lg m)$, where k is the number of clusters to be recovered; note that this running time is within a $O(k \lg m)$ factor of the size of the graph. During its running time, sofa uses $O(ks \lg m)$ space; observe that this space usage sublinear in the size in the size of the graph as long as $s = o(m/ \lg m)$ which is realistic in practice (as we argued before). Furthermore, we show that the left-side clusters of the graph can be computed using a second pass over the stream and using space $O(m)$, which is optimal since we have to output a cluster assignment for each of the m left-side vertices of the graph.

We also provide theoretical guarantees for a version of sofa. We show that under a standard random graph model, a version of sofa returns a set of planted ground-truth clusters with information-theoretically optimal memory usage; see Theorem 1 for details. We also provide similar yet weaker guarantees for the practical version of sofa.

Next, we show how sofa can be extended to solve the Boolean matrix factorization problem, which is popular in the data mining and machine learning communities. We obtain similar guarantees on space and run-time as above. Unfortunately, we cannot provide any quality guarantees here, because the lower bounds from [11] rule out obtaining non-trivial approximation ratios for practical BMF algorithms (see Section 7 for details). Thus, sofa is a heuristic for BMF, but our experiments show that it works well in practice.

We evaluate sofa on synthetic as well as on real-world datasets. On synthetically generated random graphs, our experiments show that sofa returns clusters, that are close to the planted ground-truth clusters and that its running time scales linearly in the number of edges in the graph. On real-world datasets, our experiments show that sofa is orders of magnitudes faster and more memory-efficient than a static baseline algorithm, while at the same time achieving objective function values within factor 2 of the baseline. In concrete terms, sofa can process a graph with millions of vertices, for which the static baseline algorithm runs out of memory, using only 500 MB of RAM and, further, sofa can process a graph with hundreds of thousands of edges within less than three hours, while the baseline algorithm requires several days to finish.

Outline of the Paper. Our paper is arranged as follows. In Section 2 we formally define the problems we study. Then in Section 3 we introduce sofa, which performs a single pass over the left side of a bipartite graph and then returns the right-side clusters. We show how the left-side clusters can be recovered during a second pass over the stream in Section 4. In Section 5, we discuss certain adjustments of the algorithms that we made during the implementation and then we evaluate sofa experimentally in Section 6. We discuss related work in Section 7 and conclude the paper in Section 8. Appendix A contains our theoretical analysis.

2. PRELIMINARIES

In this section, we formally introduce the problems we study, we discuss their relationship and we introduce an important subroutine of our algorithms.

2.1 Biclustering in Random Graphs

We study biclustering of random bipartite graphs. Let $G = (U \cup V, E)$ be a bipartite graph, where U is the set of vertices on the left side of the graph and V is the set of vertices on the right side of the graph. We assume that U is partitioned into subsets $U_1, \dots, U_k$ for $k > 1$ and $V_1, \dots, V_k$ are subsets of V (it is not necessary that the V_j are mutually disjoint or that their union is the set V).

Now let $p, q \in [0, 1]$ be probabilities with $p > q$. In our random model, vertices $u \in U_i$ have edges to vertices $v \in V_i$ with "large" probability and to vertices in $v \in V_j$ with $i \neq j$ with "low" probability. More concretely, we assume that

$$\mathbf{Pr}\left((u, v) \in E\right) = \begin{cases} p, & \text{if } u \in U_i, v \in V_i, \\ q, & \text{if } u \in U_i, v \in V_j, i \neq j. \end{cases} \quad (1)$$

Now the computational problem is as follows. We assume that our algorithms obtain as input a graph G generated from the random model above and the parameters k, p and q (but have no knowledge about the sets U_i and V_j). The task is to recover the clusters U_i and V_j from G; that is, the algorithm must output clusters $\tilde{U}_1, \dots, \tilde{U}_k \subseteq U$ and $\tilde{V}_1, \dots, \tilde{V}_k \subseteq V$, such that $\{\tilde{U}_1, \dots, \tilde{U}_k\} = \{U_1, \dots, U_k\}$ and $\{\tilde{V}_1, \dots, \tilde{V}_k\} = \{V_1, \dots, V_k\}$.

We decided to study the above random graph model for two reasons. First, the model has been widely studied theoretically, e.g., in machine learning [32, 46] and in mathematics [2, 45], and similar models have been used to derive practical algorithms [36, 38]. Second, when dropping the random graph assumption and assuming worst-case inputs, biclustering problems are NP-hard [33] and require prohibitively high running times [11].

In the streaming setting, the algorithm's input is a stream of the left-side vertices $u \in U$, where each vertex arrives together with all of its incident edges. We further assume that for some parameter s, each $u \in U$ has at most s incident edges and that $|V_i| \leq s$ for all i. Note that the stream only contains left-side vertices $u \in U$ and does *not* contain the vertices $v \in V$. After single pass over the stream, the algorithm must return the right-side clusters $\tilde{V}_i$. Then, the algorithm is allowed a second pass over the stream to output the left-side clusters $\tilde{U}_i$.

Next, we state our theoretical guarantees. We prove that after a single pass over the left-side vertices of a bipartite graph, the planted right-side clusters can be recovered if some conditions hold. We write $A \triangle B = (A \setminus B) \cup (B \setminus A)$ to denote the symmetric difference.

THEOREM 1. *Let $G = (U \cup V, E)$ be a random bipartite graph with planted clusters $U_1, \dots, U_k$ and $V_1, \dots, V_k$ as above. Let $p \in [1/2, 0.99]$ and $s = \max_i |V_i|$. There exist constants K_1, K_2, K_3, K_4 such that if*

$$q \leq K_1 ps/n, \qquad |U_i| \geq K_2 \lg n \text{ for all } i,$$
$$|V_i| \geq K_3 \lg n \text{ for all } i, \qquad |V_i \triangle V_{i'}| \geq K_4 s \text{ for } i \neq i',$$

then there exists an algorithm which w.h.p. returns clusters $\tilde{V}_1, \dots, \tilde{V}_k$ such that $\{\tilde{V}_1, \dots, \tilde{V}_k\} = \{V_1, \dots, V_k\}$. The algorithm uses $O(ks)$ space and has a running time of $O(mks)$.

Let us briefly discuss this result and for simplicity assume that the V_i are disjoint and have size $|V_i| = s = \Omega(\lg n)$. Then the bounds for p and q essentially require that $p > 1/2$, $q \approx ps/n$ and $|U_i| = \Omega(\lg n)$. While this is much weaker than bounds derived for static algorithms for this type of random graph model (e.g., [32, 46]), the static algorithms do not use sublinear space. Furthermore, the bounds on p and q are almost optimal when one wants to ensure that a greedy clustering of the left-side vertices succeeds.

We also show that *any* algorithm recovering the planted right-side clusters must use space $\Omega(ks)$. Thus, the space usage of the algorithm from the theorem is optimal. We prove the theorem and the proposition in Appendix A.

PROPOSITION 2. *Any algorithm solving the above biclustering problem requires space $\Omega(ks)$.*

2.2 Boolean Matrix Factorization (BMF)

In the Boolean Matrix Factorization (BMF) problem, the input is a matrix $B \in \{0,1\}^{m \times n}$ and the task is to find factor matrices $L \in \{0,1\}^{m \times k}$ and $R \in \{0,1\}^{k \times n}$ such that $\|B - L \circ R\|_2$ is minimized. Here, $\circ$ denotes matrix multiplication under the Boolean algebra, i.e., for all $i = 1, \dots, m$ and $j = 1, \dots, n$,

$$(L \circ R)_{ij} = \bigvee_{r=1}^{k} (L_{ir} \wedge R_{rj}) .$$

In the streaming setting, the algorithm's input is a stream consisting of the rows B_i of B, where we assume that each row B_i has at most s non-zero entries. After a single pass over the stream, the algorithm must output the right factor matrix R. Then, the algorithm is allowed a second pass over the stream to compute the left factor matrix L.

While the biclustering problem and the BMF problem might appear quite different at first glance, they are tightly connected. Indeed, there is a one-to-one correspondence between bipartite graphs $G = (U \cup V, E)$ with $U = \{u_1, \dots, u_m\}$ and $V = \{v_1, \dots, v_n\}$ and Boolean matrices $B \in \{0,1\}^{m \times n}$: The rows of B correspond to the vertices $u_i \in U$ and the columns of B correspond to the vertices $v_j \in V$; now one sets $B_{ij} = 1$ iff $(u_i, v_j) \in E$. This is yields a bijective mapping between bipartite graphs and Boolean matrices; B is often called the *biadjacency matrix* of G.

Furthermore, there exists a correspondence between clusterings $U_1, \dots, U_k \subseteq U$ and $V_1, \dots, V_k \subseteq V$ and the factor matrices L and R: The clusters U_i correspond to the columns of L and the clusters V_j correspond to the rows of R. More precisely, consider the r'th column of L and set it to the indicator vector of U_k, i.e., we set $L_{ir} = 1$ iff $u_i \in U_r$. Simililary, we set $R_{rj} = 1$ iff $v_j \in V_r$.

There are two main differences between the problems. First, while in biclustering we try to recover a set of planted ground-truth clusters, in BMF we try to optimize an objective function. However, when $p > 1/2 > q$, a "good" biclustering solution will also provide a good BMF solution and vice versa. Second, in biclustering each vertex $u \in U$ belongs to exactly one cluster U_i (since the U_i partition U). This would correspond to the constraint in BMF that each column of the factor matrix L must contain exactly one non-zero entry. However, in BMF we do not make this assumption and allow each column of L to contain arbitrarily many non-zero entries. Thus, in BMF the vertices $u \in U$ are allowed to be member of multiple clusters $U_{i_1}, \dots, U_{i_t}$ (and the clusters U_i do not have to be mutually disjoint). To address these differences, in Section 4 we use different algorithms for computing the left-side clusters U_i for biclustering and for BMF.

2.3 Mergeable Heavy Hitters Data Structures

Next, we recap mergeable heavy hitters data structures, which we will use as subroutines in our algorithms.

Let $X = (e_1, \dots, e_N)$ be a stream of elements from a discrete domain A. The *frequency* f_a of an element $a \in A$ is its number of occurrences in the stream, i.e., $f_a = |\{i : e_i = a\}|$. In the *heavy hitters* problem the task is to output all elements with $f_a \geq \varepsilon N$ and none with $f_a < \varepsilon N/2$ after a single pass over the stream for $\varepsilon > 0$.

Misra and Gries [31] provided a data structure which solves the heavy hitters problem using $O(1/\varepsilon)$ space. In fact, their data structure can approximate the frequency of each element $a \in A$ with additive error at most $\varepsilon N/2$. For the rest of the paper, we will denote Misra–Griess data structures by MG.

Agarwal et al. [4] showed that Misra–Gries data structures are *mergeable*: Let MG_1 and MG_2 be two Misra–Gries data structures which were constructed on two different streams X_1 and X_2. Then there exists a merge algorithm which on input MG_1 and MG_2 constructs a new data structure, that satisfies the same guarantees as a Misra–Gries data structure which was built on the concatenated stream $X_1 \cup X_2$. We write $\mathsf{MG}_1 \cup \mathsf{MG}_2$ to denote such a merged data structure.

Remark. While we use the mergeable version of the Misra–Gries data structure, we could as well other mergeable heavy hitters data structures such as the count-min sketch [12]. See [4] for more details on mergeable data structures.

3. PASS 1: RECOVER RIGHT CLUSTERS

We describe two algorithms for computing the right clusters $\tilde{V}_j$. As described in Section 2, we assume that the algorithms obtain as input a stream $U = (u_1, \dots, u_m)$ consisting of vertices from the left side of the graph, where each u_i arrives together with all of its at most s edges to vertices on the right side of the graph. After a single pass over U, the algorithm must return clusters $\tilde{V}_1, \dots, \tilde{V}_k$ on the right side of the graph.

It will be convenient to identify the vertices $u \in U$ with bitvectors $x_u \in \{0,1\}^n$, where we set $x_u(j) = 1$ iff $(u, v_j) \in E$, i.e., $x_u(j) = 1$ iff vertex u is a neighbor of $v_j \in V$. For two vertices $u, u' \in U$, we let $d(x_u, x_{u'}) = |\{j : x_u(j) \neq x_{u'}(j)\}|$ denote the Hamming distance of x_u and $x_{u'}$, i.e., $d(x_u, x_{u'})$ measures the number of vertices in V which are incident upon u or u' but not both.

We will first describe a simplified greedy algorithm to highlight our main ideas; this is the algorithm mentioned in Theorem 1. Then we provide a second, more practical, algorithm in Section 3.2, which we implement and evaluate in Sections 5 and 6.

3.1 Warm Up: Greedy Biclustering

We start by discussing a simplified greedy algorithm to explain the main idea of our approach. This greedy algorithm has the guarantees stated in Theorem 1.

Before describing the algorithm, let us first make two observations about the properties of the random graph model in Section 2.1: (1) Suppose we know a planted left-side cluster U_i and we want to recover its corresponding right-side cluster V_i. Then observe that by Equation 1 every vertex $v \in V_i$ has $p|U_i|$ neighbors in U_i and every vertex $v \notin V_i$

has $q|U_i|$ neighbors in U_i. Thus, if U_i is large enough and p is sufficiently larger than q, we can find a threshold θ such that with high probability all $v \in V_i$ have more than $\theta|U_i|$ neighbors in U_i and all $v \notin V_i$ have less than $\theta|U_i|$ neighbors in U_i. Hence, recovering the cluster V_i essentially boils down to identifying those vertices in V_i which are frequently neighbors of vertices in U_i. In other words, we want to find the heavy hitters among the neighbors of vertices in U_i. (2) The second insight is that when processing the stream, the vertices $u, u' \in U_i$ from the same cluster will have similar neighborhoods in V and, hence, $d(x_u, x_{u'})$ is small. More concretely, assume that $d(x_u, x_{u'}) < \alpha$ for some suitable parameter α. On the other hand, if $u \in U_i$ and $u'' \in U_j$ with $i \neq j$, their neighborhoods will be quite different and $d(x_u, x_{u''}) > \alpha$ is large. Thus, a greedy clustering of the vertices $u \in U$ based on the distances of their corresponding vectors x_u suffices to recover the U_i. In Appendix A, we show how θ and α can be picked under the conditions from Theorem 1.

Roughly speaking, the algorithm works as follows. It assumes that it obtains parameters θ and α with the above properties as input. Now the algorithm greedily forms clusters of all left-side vertices which have distance at most α; this corresponds to Observation (2) above. To save memory, the algorithm only stores *a single vertex* for each cluster. Furthermore, for each cluster consisting of left-side vertices, the algorithm keeps track how many of its edges are incident upon each right-side vertex $v \in V$. Since we do not have enough memory to store a counter for each vertex $v \in V$, the algorithm uses the mergeable heavy hitters data structure from Section 2.3 to approximately keep track of how many times each right-side vertex appeared; this corresponds to Observation (1) above.

Now we describe the algorithm more formally and present its pseudocode in Algorithm 1. The algorithm obtains as input U, a distance parameter α and a rounding threshold θ. It maintains a set of *centers* C which is initially empty. For each center $c \in C$, the algorithm stores a heavy hitters data structure $\mathsf{MG}(c)$ with $O(s)$ counters and a counter n_c denoting how many vertices have been assigned to c.

Now the algorithm processes the vertices $u \in U$ as follows. First, it checks whether x_u has Hamming distance more than α from all centers $c \in C$. If this is the case, the algorithm opens u as a new center. That is, it sets $C \leftarrow C \cup \{u\}$ and sets $n_u \leftarrow 1$. Else, there exists a center $c(u) \in C$ with $d(x_u, x_{c(u)}) \leq \alpha$ and the algorithm *assigns* u to $c(u)$. When assigning u to $c(u)$, the algorithm first creates a heavy hitters data structure $\mathsf{MG}(u)$ containing all j such that $(u, v_j) \in E$ (note that the algorithm has access to this information since u arrives together with all of its incident edges). Then it merges $\mathsf{MG}(c(u))$ and $\mathsf{MG}(u)$ and updates $\mathsf{MG}(c(u))$ to this merged heavy hitters data structure. Furthermore, the algorithm increases the counter $n_{c(u)}$ by 1. Then it proceeds with the next point from the stream.

When the algorithm finished processing the stream, it performs a postprocessing step. It iterates over all centers $c \in C$ and sets $\tilde{V}_c$ to all vertices $v_j \in V$ such that the counter of j in $\mathsf{MG}(C)$ is at least θn_c, where θ is the rounding threshold from the input and n_c is the number of vertices that were assigned to c. Then the algorithm outputs the clusters $\tilde{V}_c$ as its solution.

Remark. Note that Algorithm 1 only delivers good results when the parameters α and θ provide exactly those guarantees which we discussed at the beginning of the subsection. In Section A we show how α and θ can be set when the parameters p, q and k are known for random graph models as introduced in Section 2.1; under this assumption we show that the algorithm indeed returns the planted clusters $V_1, \dots, V_k$ after a single pass over the stream and using essentially optimal space. However, in practice it is unrealistic that one has knowledge about these parameters. Especially setting the parameter α seems troublesome; for example, when setting α incorrectly, one cannot even guarantee to obtain k clusters in total. We show how to resolve this issue in the next subsection.

Algorithm 1 Greedy-clustering (U, α, θ)

```
1: C ← ∅
2: for u ← next vertex from stream
3:     d ← min_{c∈C} d(x_u, x_c)
4:     if d > α                                  ▷ open u as center
5:         C ← C ∪ {u}
6:         n_u ← 1
7:     else                     ▷ Assign u to its closest center c(u)
8:         c(u) ← arg min_{c∈C} d(x_u, x_c)
9:         MG(c(u)) ← MG(c(u)) ∪ MG(u)
10:        n_{c(u)} ← n_{c(u)} + 1
11: for all c ∈ C                                ▷ Postprocessing
12:     Ṽ_c ← {v_j ∈ V : the counter of j in MG(c) is at least θn_c}
```

3.2 Biclustering Using Importance Sampling

We introduce the sofa algorithm which constitutes the main contribution of our paper; sofa is short for *Streaming bOolean FactorizAtion*. sofa performs a single pass over the vertices $u \in U$ and afterwards returns clusters $\tilde{V}_1, \dots, \tilde{V}_k$. One can view sofa as the more practical version of Algorithm 1, since it it does not require the parameter α which is not available in practice. In a nutshell, we will replace the greedy clustering from Algorithm 1 by the streaming k-Medians algorithm from Braverman et al. [9] which is based on importance sampling. The pseudocode of sofa with all details is presented in Algorithm 2.

Roughly speaking, sofa works as follows. sofa maintains a set of *centers* C which is initially empty; we impose that C is never allowed to contain more than $c_{\max}$ vertices, where $c_{\max}$ is a user-defined parameter. As before, for each center $c \in C$, the algorithm maintains a heavy hitters data structure $\mathsf{MG}(c)$. When sofa processes the vertices from the stream and a new vertex u arrives, sofa computes the distance $d = d(x_u, x_{c(u)})$ from u to the closest center $c(u)$ in C. It then opens u as new center with probability proportional to d; if u is not opened as a center, sofa assigns u to $c(u)$. Thus, if u is "close" to $c(u)$ then u is unlikely to become a new center and more likely to be assigned to $c(u)$; on the other hand, if u is "far away" from $c(u)$ (and, hence, all centers), then u is likely to become a new center. As before, when a vertex u is assigned to $c(u)$, the indices of all neighbors of u are added to $\mathsf{MG}(c(u))$. Next, suppose that after opening a new center, the set C contains $c_{\max}$ centers. Then sofa restarts on the stream which only consists of the $c_{\max}$ centers in C and all unprocessed vertices of the stream. When sofa restarts on the centers of C and one of the previous centers c_i is assigned to another previous center c_j, then sofa merges their corresponding heavy hitters data structures $\mathsf{MG}(c_i)$ and $\mathsf{MG}(c_j)$ as described in Section 2. Finally, after processing all vertices from the stream and obtaining a set of centers C together with their heavy hitters

data structures, we run a postprocessing step. At this point C can contain more than k centers (but at most $c_{\max}$). We run a static k-Medians algorithm on the vectors x_c for $c \in C$ to obtain a clustering of C into subsets $C_1, \dots, C_k$. For each C_i, we merge the heavy hitters data structures of the centers in C_i and denote this merged data structure as MG_i. As before, we set $\tilde{V}_i$ to all vertices $v_j \in V$ which have a counter of value at least $\theta|C_i|$ in MG_i.

We now elaborate on the details of sofa. At the beginning, sofa initializes a lower bound LB on the k-Medians clustering cost of the points x_u in the stream to 1. It also maintains an approximation of the current cost of the clustering which we denote *cost* and initialize to 0. After that, sofa starts processing the vertices from the stream. We maintain a set of centers C for which we ensure that $|C| < c_{\max}$ at all times. For each center we store a heavy hitters data structure from Section 2.3 with $O(s)$ counters.

When starting to process the vertices from the stream, sofa computes a weight $f \leftarrow LB/(k(1 + \lg n))$. As long as there are unread vertices in the stream, $|C| < c_{\max}$ and $cost < 2LB$, sofa proceeds as follows. It reads the next vertex u from the stream and sets d to the distance $d(x_u, x_{c(u)})$ of u to its closest center $c(u)$. Now it opens u as a new center with probability $\min\{w(u) \cdot d/f, 1\}$, where $w(u)$ is the weight of u. sofa maintains as invariant that if u was a previously unprocessed vertex from the stream, then $w(u) = 1$, and, if u was a center before, then $w(u)$ is the number of vertices which were previously assigned to u. If u is opened as a new center, we set $C \leftarrow C \cup \{u\}$. If u is assigned to its closest center $c(u)$, then we increase *cost* by $w(u) \cdot d$, increase the weight of $c(u)$ by $w(u)$ and set $\mathsf{MG}(c(u))$ to the merged heavy hitters data structures of $\mathsf{MG}(c(u))$ and $\mathsf{MG}(u)$.

If at some point $|C| = c_{\max}$ or $cost > 2LB$, then sofa doubles LB. Furthermore, sofa restarts on the stream which consists of the $c_{\max}$ vertices of C and all unprocessed vertices from U (in this order). Note that the vertices $c \in C$ still have their previously assigned weights $w(c)$, whereas the vertices in the unprocessed part of U all have weight 1.

After sofa finished processing all vertices from the stream, we perform a postprocessing step. We start by running a static $O(1)$-approximate k-Medians algorithm on the points x_c for $c \in C$ which uses only $O(|C| \cdot s)$ space and which runs in time $\mathrm{poly}(|C| \cdot s)$; this can be done, for example, using the local search algorithm by Arya et al. [6]. This provides us with a clustering of C into disjoint subsets $C_1, \dots, C_k$. Now for each $i = 1, \dots, k$, we set MG_i to the merged heavy hitters data structure of all vertices in C_i and $|C_i|$ to the sum of the weights of all vertices in C_i. Finally, we set $\tilde{V}_i$ to all vertices $v_j \in V$ such that the counter of j in MG_i is at least $\theta|C_i|$.

Space Usage and Running Time. We briefly argue that sofa uses space $O(ks \lg m)$ and its running time is bounded by $O(mks \lg m)$. Observe that the main space usage comes from storing the set of centers C together with a heavy hitters data structure for each center. Recall that we ensure that $|C| \leq c_{\max}$ at all times. Furthermore, each center has $O(s)$ incident edges (by assumption on our input stream) and we set the number of counters for each heavy hitters data structure to $O(s)$. Thus, the total space usage is $O(c_{\max} s)$. In an upcoming online appendix, we show that we can obtain provable guarantees for sofa if $c_{\max} = O(k \lg |U|)$.

Remark. We use the streaming k-Medians clustering algorithm from [9], because the centers it maintains are points from the stream. Thus, if these points are sparse, the space usage of sofa for storing centers directly benefits from this. Algorithms for streaming k-Means (e.g., [40]) often include steps, which cause the centers to become dense. Thus, if we used such an algorithm as a subroutine, sofa would require more space. Here, however, we focused on setting close to the information-theoretically minimum space usage and, hence, we decided to use the algorithm by [9].

Algorithm 2 sofa (U, k, $c_{\max}$, θ)

```
1: LB ← 1, cost ← 0  ▷ Process the vertices from the stream
2: while there exist unread vertices in U
3:     C ← ∅
4:     f ← LB/(k(1 + lg n))
5:     for u ← next vertex from stream
6:         d ← min_{c∈C} d(x_u, x_c)
7:         openCenter ← True, with probability min{w(x) · d/f, 1},
               and False, otherwise
8:         if openCenter = True                    ▷ open u as center
9:             C ← C ∪ {u}
10:            w(u) ← 1
11:        else              ▷ Assign u to its closest center c(u)
12:            cost ← cost + w(u) · d
13:            c(u) ← arg min_{c∈C} d(x_u, x_{c(u)})
14:            w(c(u)) ← w(c(u)) + w(u)
15:            MG(c(u)) ← MG(c(u)) ∪ MG(u)
16:        if |C| = c_max or cost > 2LB
17:            break and raise flag
18:    if flag raised
19:        U ← the stream consisting of the (weighted) vertices
               in C and all unread vertices of U
20:        LB ← 2LB
21: (C_1, ..., C_k) ← clustering of C using an O(1)-approximate
    k-Medians algorithm                       ▷ Postprocessing
22: for all i = 1, ..., k
23:     MG_i ← ⋃_{x∈C_i} MG(x)
24:     |C_i| ← Σ_{c∈C_i} w(c_i)
25:     Ṽ_i ← {v ∈ V : the counter of v in MG_i is at least θ|C_i|}
```

4. PASS 2: RECOVER LEFT CLUSTERS

In this section, we present algorithms for computing a clustering $\tilde{U}_1, \dots, \tilde{U}_k \subseteq U$ of the left side of the graph during a second pass over the stream U. We assume that our algorithms obtain as input a set of clusters $\tilde{V}_1, \dots, \tilde{V}_k \subseteq V$ from the right side of the graph. We will present two different algorithms for biclustering and BMF, respectively.

4.1 Biclustering

We now present an algorithm which performs a single pass over the stream U and assigns each $u \in U$ to exactly one cluster $\tilde{U}_i$. We will use this algorithm for the biclustering problem, where each vertex $u \in U$ belongs to a unique planted cluster U_i (see Section 2.1).

To obtain the clustering $\tilde{U}_1, \dots, \tilde{U}_k$, the algorithm initially sets $\tilde{U}_i = \emptyset$ for all $i = 1, \dots, k$. Now the algorithm performs a single pass over the stream of left-side vertices $u \in U$. For each u, let $\Gamma(u)$ denote the set of neighbors of u in V, i.e., $\Gamma(u) = \{v \in V : (u, v) \in E\} \subseteq V$. Now the algorithm assigns u to the cluster $\widetilde{U_{i^*}}$ such that the overlap of $\Gamma(u)$ and $\widetilde{V_{i^*}}$ is maximized relative to the size of $\widetilde{V_{i^*}}$. More concretely, the algorithm computes

$$i^* = \arg\max\{|\Gamma(u) \cap \tilde{V}_i| / |\tilde{V}_i| : i = 1, \dots, k\} \tag{2}$$

and then assigns u to $\widetilde{U_{i^*}}$.

Space Usage and Running Time. Observe that the algorithm uses space $O(m)$ (where $m = |U|$), since for each vertex $u \in U$, we need to store to which cluster U_i it was assigned. Furthermore, the running time of the algorithm is $O(mks)$: For each of the m vertices, we need to compute i^* as per Equation (2). Since we assume that each vertex u has at most $O(s)$ neighbors and that all $\tilde{V}_i$ have size $O(s)$, it takes time $O(s)$ to compute $|\Gamma(u) \cap \tilde{V}_i|/|\tilde{V}_i|$ for fixed i. Thus, computing i^* can be done in time $O(ks)$.

4.2 BMF

Next, we present an algorithm, which performs a single pass over the stream and computes clusters $\tilde{U}_1, \dots, \tilde{U}_k$, where every vertex $u \in U$ may be contained in multiple clusters $U_{i_1}, \dots, U_{i_T}$. Recall from Section 2.2 that this corresponds to computing a factor matrix L for the the BMF problem.

Our approach for computing the sets $\tilde{U}_i$ is similar to the greedy covering scheme used in [29]. The main idea is that for every $u \in U$, we greedily cover the set $\Gamma(u) \subseteq V$ using the clusters $\tilde{V}_1, \dots, \tilde{V}_k$ similar to the classic set cover problem. However, unlike in standard set cover, we do allow for some amount of "overcovering". Note that this greedily minimizes the symmetric difference of $\Gamma(u)$ and the sets $\tilde{V}_i$ used for covering $\Gamma(u)$; thus, also their Hamming distance is minimized.

Before we present our algorithm, let us first define our score function for the covering process. For sets A, X, Y, we define the *score of A for covering X given that Y was already covered* as $\mathit{score}(A \mid X, Y) = |(X \setminus Y) \cap A| - |A \setminus (X \cup Y)|$.

To better understand the score function, consider the case that no elements of X were covered before, i.e., $Y = \emptyset$. Then $\mathit{score}(A \mid X, \emptyset) = |X \cap A| - |A \setminus X|$ is the number of elements in X, which get covered by A, minus the number of those elements in A, which do not appear in X (these elements "overcover" X). Now suppose that $Y \neq \emptyset$, i.e., some elements of X were already covered before and these elements are stored in the set Y. Then the score function takes this into account by not adding score for elements in $A \cap X \cap Y$ that are in A and X, but were already covered before. Also, the score function does not subtract score for elements in A that are not in X, but which were already overcovered before (and, hence, are in Y); more precisely, it does not subtract score for the elements in $(A \cap Y) \setminus X$.

We now describe our greedy algorithm for computing the clusters $\tilde{U}_i$. Initially, we set $\tilde{U}_i = \emptyset$ for all i. Now we perform a single pass over the stream U and for each $u \in U$, we do the following. We initialize $Y_u = \emptyset$ and, as before, let $\Gamma(u)$ denote the set of neighbors of u in V. Now, while there exists an i such that $\mathit{score}(\tilde{V}_i \mid \Gamma(u), Y_u) > 0$, we compute

$$i^* = \arg\max_{i=1,\dots,k} \mathit{score}(\tilde{V}_i \mid \Gamma(u), Y_u). \quad (3)$$

If $\mathit{score}(\widetilde{V_{i^*}} \mid \Gamma(u), Y_u) > 0$, we assign u to $\widetilde{U_{i^*}}$ and we set $Y_u = Y_u \cup \widetilde{V_{i^*}}$. Otherwise, we stop covering u and proceed with the next vertex from the stream.

Space Usage and Running Time. The space usage is $O(km)$ since each vertex can be assigned to as many as k clusters. The running time of the algorithm is $O(mk^2 s)$: First, note that evaluating $\mathit{score}(\tilde{V}_i \mid \Gamma(u), Y_u)$ takes time $O(s)$ because all sets have size $O(s)$. Second, for a single iteration of the while-loop we need to evaluate the score function $O(k)$ times to obtain i^* and there are at most k iterations. Hence, we need to spend time $O(k^2 s)$ for each of the m vertices in U.

5. IMPLEMENTATION

We implemented the sofa from Section 3.2 for recovering the right-side clusters and the two algorithms from Section 4 for recovering the left-side clusters. Now we present certain adjustments that we made to improve the results of the algorithms and we discuss how to set certain parameters.

We implemented all algorithms in Python. To speed up the computation, the subroutines for finding the closest centers (Line 6 in Algorithm 2) and for finding the clusters with maximum score (Equation (3)) were implemented in CPython. We did not use any parallelization, i.e., our implementations are purely single-threaded. Our code is available online[1].

5.1 Asymmetric Weighted Hamming Distance

During preliminary tests of sofa on real-world data, we realized that sofa picked extremely sparse centers which often only had a single non-zero entry. This resulted in almost all vertices being assigned to this particular center (because the Hamming distance of a vertex u to a center with a single non-zero entry is the degree of u plus/minus 1 and, due to the low degrees of the left-side vertices u, these distances are usually small) which made the cluster recovery fail.

Hence, to promote denser centers, we introduce the following asymmetric weighted version of the Hamming distance. Let $c \in C$ be a center maintained by sofa and let u be a vertex which needs to be clustered. For each entry i of x_c and x_u, we assign the following costs: If $x_c(i) = x_p(i)$, then the cost is 0; if $x_p(i) = 1$ and $x_c(i) = 0$ then the cost is 1; if $x_p(i) = 0$ and $x_c(i) = 1$ then the cost is $\alpha < 1$. Now the *asymmetric weighted Hamming distance* of c and p is simply the sum over the costs for all entries of x_c and x_p.

Note that by setting $\alpha = 1$ the above results in the classic (symmetric) Hamming distance. Furthermore, setting $\alpha < 1$ promotes denser centers because the case of $x_c(i) = 1$ and $x_u(i) = 0$ is penalized less than in classic Hamming distance.

For example, consider the vectors $x_{c_1} = (1, 1, 1, 1, 0)$, $x_{c_2} = (0, 0, 0, 0, 1)$ and $x_u = (1, 0, 0, 0, 0)$. In vanilla Hamming distance, u would be assigned to c_2 since their distance is 2 and the distance of c_1 and p is 3. With asymmetric weighted Hamming distance and $\alpha = 0.1$, u is assigned to c_1 because their distance is 0.3 and the distance is u and c_2 is 1.1. Note the assignment of u to c_1 instead of c_2 is also much more suitable for the thresholding step in Line 25 of sofa.

In practice, our experiments showed that setting $\alpha = 0.1$ was a good choice for all datasets and sofa benefitted heavily from using asymmetric weighted Hamming distance.

5.2 Biclustering Algorithm

To solve the biclustering problem from Section 2.1, we implemented sofa (Algorithm 2) together with the biclustering algorithm from Section 4.1 for recovering the left clusters. The only adjustment that we made was to use the k-Means implementation of scikit-learn [35] in order to implement the $O(1)$-approximate k-Medians algorithm in Line 21 of sofa.

5.3 BMF Algorithm

To solve the BMF problem from Section 2.2, we implemented sofa (Algorithm 2) together with the BMF algorithm from Section 4.2 for recovering the left clusters.

During preliminary tests we observed that on some datasets we achieved better results when we completely skipped the

[1] https://cs.uef.fi/~pauli/bmf/sofa/

k-Median algorithm in Line 21 of sofa. Instead, we compute a cluster $\tilde{V}_c$ for each center $c \in C$. Note that this might lead to more than k clusters $\tilde{V}_c$ but to at most $c_{\max}$. Then we use the BMF algorithm from Section 4.2 to compute a cluster $\tilde{U}_c$ for each of the (potentially more than k) clusters $\tilde{V}_c$. While computing the clusters $\tilde{U}_c$, we keep track of the total score of each cluster $\tilde{V}_c$; this can be done by maintaining a counter s_c for each $c \in C$ and increasing s_c by $score(\tilde{V}_c \mid \Gamma(u), Y_u)$ whenever we compute i^* in Equation (3). To ensure that our algorithm only returns k clusters when it finishes, we sort the clusters $\tilde{V}_c$ by their score values s_c in non-increasing order and only keep the k clusters with the highest total scores. This ensures that at the end we only return k clusters.

While sofa and the algorithm from Section 4.2 return clusters $\tilde{U}_i$ and $\tilde{V}_i$ instead of Boolean factor matrices L and R as required for the BMF problem, we can transform the clusters into factor matrices L and R as discussed in Section 2.2. This gives raise to a matrix $\tilde{B} = L \circ R$ which approximates the biadjacency matrix B of the input graph G.

5.4 Setting the Rounding Threshold θ

A Heuristic for Determining θ. The supplemental material of [32] presents a heuristic for setting θ. It essentially works by observing that θ is a function of the parameters p and q of the random graph model from Section 2.1. Then it performs a grid search over different values of p and q and picks the pair (p^*, q^*) for which the resulting rounding threshold θ^* maximizes the likelihood of the counters observed in the heavy hitters data structure from Line 23 of sofa. Due to lack of space we refer to the supplemental material of [32] for the details of the heuristic. We will refer to the version of sofa which uses this heuristic as sofa-auto.

Using Multiple Thresholds. Note that the only place in sofa, where the rounding threshold θ is used, is in the postprocessing step. Thus, given multiple rounding thresholds $\theta_1, \dots, \theta_T$, it is possible to compute a set of clusters $\tilde{V}_1^{(t)}, \dots, \tilde{V}_k^{(t)}$ for each θ_t. Then for each $t = 1, \dots, T$, we can compute corresponding left-side clusters $\tilde{U}_1^{(t)}, \dots, \tilde{U}_k^{(t)}$ using the algorithms from Section 4. Note that computing the clusters $\tilde{U}_i^{(t)}$ for all values of $t = 1, \dots, T$ still only requires a single pass over the stream: For each $u \in U$ of the stream, we can run the algorithms for computing $\tilde{U}_1^{(t)}, \dots, \tilde{U}_k^{(t)}$ in parallel for all $t = 1, \dots, T$.

In our experiments we will use the above strategy to generate clusters for multiple thresholds. Then we will evaluate their quality in a separate postprocessing step (see Section 6.2). We will refer to the version of sofa which uses multiple thresholds simply as sofa.

5.5 Static to Streaming Reduction

Since many static algorithms do not scale to datasets of the size considered in this paper, we describe a reduction for turning *static* biclustering/BMF algorithms into *2-pass streaming* algorithms. We will use this reduction to compare sofa against static algorithms in our experiments.

The reduction works as follows. First, we sample a subgraph with $\tilde{m}$ left-side vertices and $\tilde{n}$ right-side vertices, where $\tilde{m} \ll m$ and $\tilde{n} \ll n$ are parameters of the reduction. Then we run the static algorithm on the sampled subgraph to determine a set of right-side clusters $\tilde{V}_1, \dots, \tilde{V}_k$ (details below). In the second pass over the stream, we use the procedure from Section 4 to infer the left-side clusters $\tilde{U}_1, \dots, \tilde{U}_k$.

Now, we elaborate on the first pass over the stream. First, we use reservoir sampling to obtain $\tilde{m}$ left-side vertices from the graph uniformly at random; let $U' = \{u'_1, \dots, u'_{\tilde{m}}\}$ denote this set of left-side vertices. Let V' be the set of right-side vertices which are adjacent to vertices in U'. Note that possibly $|V'| > \tilde{n}$ and let V'' be the set of $\tilde{n}$ vertices in V' with highest degree to vertices in U' (breaking ties arbitrarily). Now we run the static algorithm on the subgraph with the $\tilde{m}$ left-side vertices U' and $\tilde{n}$ right-side vertices V''. This gives raise to clusters $\tilde{V}_1, \dots, \tilde{V}_k$. Next, we add the (low-degree) vertices $v \in V' \setminus V''$ to the clusters $\tilde{V}_i$ by assigning each v to the cluster $\tilde{V}_i$ which "on average" has the most similar left-side neighborhood compared to v. More concretely, for each vertex $v \in V'$ we define the vector $x_v \in \{0,1\}^{\tilde{m}}$ such that $x_v(i) = 1$ iff $(u'_i, v) \in E$. Next, for each cluster $\tilde{V}_i$ define the vector $x_i = \sum_{v \in \tilde{V}_i} x_v / |\tilde{V}_i|$ which describes the "average left-side neighborhood" of the vertices in $\tilde{V}_i$. Now we assign each $v \in V' \setminus V''$ to $\tilde{V}_{i^*}$ with $i^* = \arg\min_i d(x_i, x_v)$. This yields the final clusters $\tilde{V}_1, \dots, \tilde{V}_k$.

6. EXPERIMENTS

We evaluate sofa on synthetic and on real-world datasets. We conducted the experiments on a workstation with 4 Intel i7-3770 processors at 3.4 GHz and 16 GB of main memory.

6.1 Synthetic Datasets

We start by evaluating our biclustering version of sofa from Section 5.2 on synthetic data. We ran sofa with different numbers of centers $c_{\max} \in \{100, 200\}$ and with 100 and 200 counters in the heavy hitters data structures.

We compare sofa against three different algorithms. First, a version of the algorithm from [32] which does not use any spectral preprocessing; this algorithm is denoted static sofa. static sofa can be viewed as a non-streaming version of sofa, i.e., it performs the clustering offline using k-Means (instead of streaming k-Median) and then it performs the thresholding step (Line 25) using the exact frequency counts (instead of the approximate frequency counts from the heavy hitters data structures). Thus, static sofa essentially provides an upper bound on how good the streaming version of sofa can potentially get. Next, we turn the static biclustering algorithms by Dhillon [14] and Zha et al. [44] into streaming algorithms via the reduction from Section 5.5, where we set $\tilde{m} = \tilde{n} = 5000$, i.e., we sample subgraphs with 5000 vertices on both sides. We denote these algorithms RSdhillon and RSzhaEtAl, where RS stands for *random subgraph*.

Data Generation and Quality Measure. We generated the synthetic data as follows. We start with an empty graph and then for each ground-truth cluster U_i, we insert ℓ vertices (see below for which values of ℓ we used in the experiments). Then we inserted 8000 vertices on the right side of the graph (i.e., $|V| = n = 8000$). To generate the ground-truth clusters V_i, we simply picked r vertices uniformly at random from V for each i (see below for how r was set in the experiments). Now the random edges were inserted exactly as described in the random graph model from Section 2.1.

When not mentioned otherwise, we have set the parameters for the graph generation as follows: $n = 8000$, $k = 50$, $\ell = 200$ (and, hence, $|U| = m = k \cdot \ell = 10\,000$), $p = 0.7$, $r = 30$. Furthermore, we set q such that in expectation every left-side vertex obtains 20 random neighbors.

To evaluate the output of the algorithms, let $U_1, \ldots, U_k$ be the planted ground-truth clusters and let $\tilde{U}_1, \ldots, \tilde{U}_k$ be the clusters returned by one of the algorithms. We define the *quality* Q of the clustering $\tilde{U}_1, \ldots, \tilde{U}_k$ as

$$Q = \frac{1}{k} \sum_{i=1}^{k} \max_{j=1,\ldots,s} J(U_i, \tilde{U}_j) \in [0,1],$$

where $J(A, B) = |A \cap B|/|A \cup B|$ is the Jaccard coefficient. That is, for each ground-truth cluster U_i, we find the cluster $\tilde{U}_j$ which maximizes the Jaccard coefficient of U_i and $\tilde{U}_j$. The quality is then simply the sum over the Jaccard coefficients for all ground-truth clusters U_i, normalized by k. Clearly, higher values for Q imply a clustering closer to the planted clustering. For example, if the clusters $\tilde{U}_j$ match *exactly* the ground-truth clusters U_i then $Q = 1$. We evaluate the quality of the clusters $\tilde{V}_i$ in exactly the same way.

Experiments. Next, let us discuss the outcomes of our experiments in different scenarios, where each time we vary one of the parameters. For each set of parameters we generated 15 different datasets and we will be reporting averages and standard deviations for the recovery quality of the algorithms. Our results are reported in Figure 1.

Varying Amount of Signal. First, let us consider a varying amount of signal, i.e., we set $p \in \{0.5, 0.6, 0.7, 0.8, 0.9\}$. One can see in Figures 1(a) and 1(b) that the quality of all sofa-versions improves as p increases. Furthermore, static sofa achieves the best quality for recovering the left and right clusters. The second-best sofa-version is sofa with 200 counters and 200 centers and achieves between 0.05 and 0.1 less quality than static sofa; we ran significance tests and these differences are significant. When only providing 100 centers, sofa has some problems for values $p \in \{0.5, 0.6\}$; this is not surprising since we planted 50 clusters and thus only maintaining 100 centers is quite restrictive for sofa. The right-side recovery of RSdhillon and RSzhaEtAl is relatively constant, where RSdhillon is performing on a high level; we explain the flatness of the curves by the spectral methods used in the algorithms, which "denoise" the data well even for small p. The left-side recovery of both algorithms is clearly worse than those of the sofa-versions. Regarding the running times (Figure 1(c)), we see that all versions of sofa are about a factor 3 faster than static sofa; note that sofa with 100 centers is also significantly faster than the versions of sofa with 200 centers. RSdhillon and RSzhaEtAl are about factor 1.5–2 slower than sofa.

Varying Size of Right Clusters. Next, we varied the sizes $r \in \{15, 20, 30, 50\}$ of the planted right clusters V_i. We can see (Figures 1(d) and 1(e)) that most algorithms benefit from larger r and that once again static sofa is the best method, followed by sofa with 200 counters and 200 centers. When the right clusters are very small (sizes 15, 20), sofa is much worse than static sofa and RSdhillon. Indeed, for small values of r, the vertices become much harder to cluster for sofa, because the Hamming distances of the vertices get dominated by noise. However, for $r \geq 30$, the version of sofa with 200 counters and 200 centers only has a 0.1 gap in quality compared to static sofa. Furthermore, observe that the performance of sofa with only 100 counters in the heavy hitters data structures drops dramatically for $r = 50$; this is caused by the frequency estimations of the right-side vertices getting too inaccurate due to the too small number of counters in the heavy hitters data structures. RSdhillon's quality is again relatively constant at roughly the same level as before, while RSzhaEtAl clearly benefits from larger cluster sizes. The running times of the algorithms (Figure 1(f)) slightly rise as r increases since the datasets contain more non-zero entries.

Varying Size of Left Clusters. Finally, we varied the sizes $\ell \in \{100, 150, 200, 300, 400, 500, 600\}$ of the left clusters U_i, which corresponds to adding more left-side vertices to the bipartite graph (and, hence, also more edges). Figures 1(g) and 1(h) show that the recovery quality is relatively unaffected from this change in ℓ and that the ranking of the algorithms is as before. However, note that the running times of static sofa increase much more rapidly than those of the streaming algorithms. For example, for $\ell = 100$ the running times of sofa and static sofa differ by a factor of less than 2 but for $\ell = 600$ this is already approximately 7.

Conclusion. We conclude that sofa can achieve recovery qualities close to the static baseline even when its number of centers is only $4k$ and its number of counters is within factor 4 of the size of the right-side clusters. Furthermore, sofa's run-time scales much better than the static baseline's. While RSdhillon delivered good quality for right-side recovery, its left-side recovery was rather poor. RSzhaEtAl performs badly overall; we blame this on the data being too sparse, which does not allow the algorithm to find good cuts.

6.2 Real-World Datasets

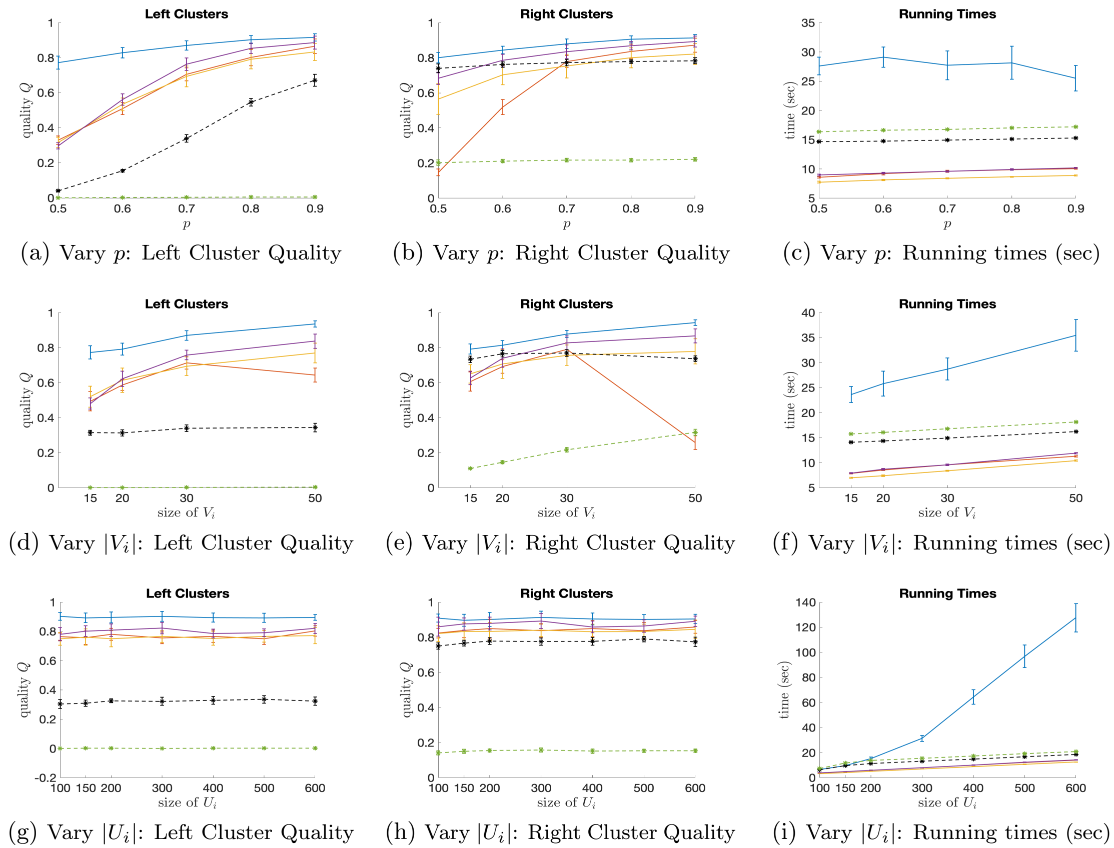
For the real-world experiments, it is more realistic to allow the left-side clusters U_i to overlap. Thus, for the real-world experiments, we use the version of sofa which solves the BMF problem from Section 5.3.

Methods and Measures. For these experiments, we use sofa and sofa-auto. For sofa, we set the threshold θ using a line search over values $\theta \in \{0.3, 0.4, 0.5, 0.6, 0.7\}$. The remaining parameters were set as follows: $c_{\max} = 20k$, where k is the desired number of clusters; $s = P_{99}$, the 99th quantile of the degrees on the left-side vertices (see Table 1 for the values for each dataset); and we set the number of counters in the heavy hitters data structures to $\max\{3s, 0.05n\}$.

As for the synthetic datasets, we compare sofa against RSdhillon and RSzhaEtAl. We used $\tilde{m} = \tilde{n} = 15000$ in the reduction. With these parameters, RSdhillon and RSzhaEtAl have running times comparable to sofa and already for $\tilde{m} = \tilde{n} = 20000$, our workstation would often run out of memory. Further, we compare against the static (i.e., non-streaming) algorithm basso[2], which is an efficient implementation of the asso algorithm [29]. basso has one hyperparameter, τ. We try values $\tau \in \{0.2, 0.4, 0.6, 0.8\}$ and report the results with the best value. For run-time and memory usage analysis, we report average values over different thresholds. The time complexity of basso is $O(k|U|^2|V|)$ and thus we flipped U and V in the input for basso when $|U| > |V|$.

For all datasets, we computed clusterings consisting of $k = 50, 100, 200$ clusters. Since for the real-world datasets no information about the ground-truth clusters is available, we use relative Hamming gain and recall as quality measures to evaluate the obtained clusterings. Formally, let B be the biadjacency matrix of the bipartite graph and let $\tilde{B}$ an approximation thereof. The *relative Hamming gain* is defined as $1 - |\{(i,j) : B_{ij} \neq \tilde{B}_{ij}\}|/|\{(i,j) : B_{ij} = 1\}|$, and it indicates how much better $\tilde{B}$ approximates B than a trivial (all-zeros) matrix would. The *recall* is defined as

[2] basso v0.5 from https://cs.uef.fi/~pauli/bmf/asso/



(a) Vary p: Left Cluster Quality (b) Vary p: Right Cluster Quality (c) Vary p: Running times (sec)

(d) Vary $|V_i|$: Left Cluster Quality (e) Vary $|V_i|$: Right Cluster Quality (f) Vary $|V_i|$: Running times (sec)

(g) Vary $|U_i|$: Left Cluster Quality (h) Vary $|U_i|$: Right Cluster Quality (i) Vary $|U_i|$: Running times (sec)

Figure 1: Results on synthetic data. Figures 1(a)–1(c) have varying p, Figures 1(d)–1(f) have varying sizes of the right clusters V_i, Figures 1(g)–1(i) have varying sizes of the left clusters U_i. Markers are mean values over 15 different datasets; error bars are one standard deviation over the 15 datasets.

$|\{(i,j) : B_{ij} = 1 \wedge \tilde{B}_{ij} = 1\}|/|\{(i,j) : B_{ij} = 1\}|$, and it indicates the fraction of edges (1s) in B which are "covered" correctly by the matrix $\tilde{B}$ returned by one of the algorithm.

Explanation of Datasets. In our experiments, we used six real-world datasets. Their basic properties are described in Table 1. Notice that all datasets are very sparse, and their left-side degrees (even in the 99th percentile) are small compared to the number of vertices on the right side of the graph. This empirically validates two of the three properties we discussed in the introduction.

Let us briefly discuss each of the datasets. 20News[3] contains newsgroup postings on the left side and words on the right side; edges indicate a word appearing in a posting. The datasets Reuters and Flickr were taken from the KONECT[4] [23] website. Reuters has articles from the news organization Reuters on the left side and words on the right. Flickr encodes the group memberships (right) of Flickr users (left). Wiki[5] is from the SuiteSparse Matrix Collection [13] and consists of Wikipedia pages on both sides of the graph; an edge (u, v) indicates that page u links to page v (note that this relationship is asymmetric). Book[6] [47] is a rating matrix consisting of users on the left side and books on the right side; an edge indicates that a user rated book. Movie[7] is a rating matrix between users and movies [18].

Experiments. Results for relative Hamming gain and recall are presented in Table 2. Note that basso did not finish on the Wiki dataset, because it ran out of memory.

The results for relative Hamming gain show that, when it is able to finish, basso is always the best method. This is to be expected as it can make unlimited passes over the data. On all datasets except Book and for all values of k, the results of sofa and basso are within factor at most 2.2. For $k = 200$, the results of sofa are at most 50% worse than those of basso on 20News, Reuters and Movie. With Book, on the other hand, sofa is significantly worse (up to factor 5.8) but still much better than RSdhillon and RSzhaEtAl. We believe this results from Book being too sparse; indeed, the 50% percentile of the degrees of the left vertices in book is 1 and thus sofa's

[3] http://qwone.com/~jason/20Newsgroups/
[4] http://konect.uni-koblenz.de
[5] https://www.cise.ufl.edu/research/sparse/matrices/Gleich/wikipedia-20051105
[6] http://www2.informatik.uni-freiburg.de/~cziegler/BX/
[7] https://grouplens.org/datasets/movielens/20m/

Table 1: Real-world dataset properties. Datasets are considered as bipartite graphs $G = (U \cup V, E)$ and density is $|E|/(|U| \cdot |V|)$. Average degree $\overline{\deg}$ and the 99th percentile degree P_{99} are calculated from U and rounded to the nearest integer.

Dataset	$\|U\|$	$\|V\|$	$\|E\|$	density	$\overline{\deg}$	P_{99}
20News	18 773	61 056	1 766 780	0.0015	94	548
Reuters	38 677	19 757	978 446	0.0013	25	498
Book	105 282	340 550	1 149 779	< 0.0001	11	174
Movie	138 493	26 744	20 000 263	0.0054	144	1113
Flickr	395 979	103 631	8 545 307	0.0002	22	268
Wiki	1 562 433	1 170 854	19 753 078	< 0.0001	17	177

Table 2: Relative Hamming gain and recall in different real-world datasets

k	Algorithm	Relative Hamming gain						Recall					
		20News	Reuters	Book	Movie	Flickr	Wiki	20News	Reuters	Book	Movie	Flickr	Wiki
50	sofa-auto	0.0298	0.0450	0.0198	0.0805	0.0380	0.0617	0.0446	0.0649	0.0201	0.1262	0.0480	0.0657
	sofa	0.0424	0.0454	0.0212	0.1188	0.0453	0.0695	0.0483	0.0652	0.0214	0.1779	0.0474	0.0700
	basso	0.0545	0.1005	0.1226	0.1394	0.0719	–	0.0683	0.1677	0.1226	0.2855	0.0760	–
	RSdhillon	0.0042	0.0273	0.0008	0.1056	0.0040	0.0001	0.0069	0.0316	0.0009	0.1999	0.0088	0.0001
	RSzhaEtAl	0.0001	0.0274	0.0008	0.0297	0.0000	0.0000	0.0004	0.0447	0.0014	0.0614	0.0001	0.0000
100	sofa-auto	0.0411	0.0792	0.0298	0.1028	0.0486	0.0730	0.0570	0.0991	0.0307	0.1597	0.0636	0.0777
	sofa	0.0574	0.0777	0.0333	0.1367	0.0668	0.0824	0.0649	0.0987	0.0341	0.2030	0.0721	0.0840
	basso	0.0793	0.1097	0.1783	0.1739	0.1068	–	0.0959	0.1907	0.1783	0.3143	0.1124	–
	RSdhillon	0.0059	0.0307	0.0028	0.1378	0.0137	0.0262	0.0103	0.0430	0.0060	0.2400	0.0246	0.0302
	RSzhaEtAl	0.0006	0.0342	0.0030	0.0696	0.0000	0.0000	0.0017	0.0500	0.0040	0.1182	0.0002	0.0000
200	sofa-auto	0.0624	0.1253	0.0427	0.1247	0.0663	0.0861	0.0788	0.1441	0.0435	0.1926	0.0837	0.0924
	sofa	0.0930	0.1254	0.0472	0.1598	0.0817	0.1061	0.0991	0.1442	0.0479	0.2353	0.0906	0.1087
	basso	0.1171	0.1334	0.2531	0.2376	0.1556	–	0.1321	0.2100	0.2532	0.3521	0.1603	–
	RSdhillon	0.0092	0.0402	0.0024	0.1771	0.0203	0.0270	0.0159	0.0619	0.0030	0.2812	0.0317	0.0299
	RSzhaEtAl	0.0014	0.0291	0.0017	0.1104	0.0007	0.0001	0.0022	0.0454	0.0027	0.1644	0.0021	0.0002

clustering seems to fails. Overall, the results of sofa and sofa-auto improve significantly as k increases, showing that it can be used for small and large values of k alike. RSdhillon and RSzhaEtAl perform well when $|V|$ is small (e.g., Movie and Reuters), but as soon as $|V|$ increases, their results decays dramatically (e.g., Book, Flickr and Wiki); this appears to be a limitation of the random sampling approach.

The results concerning the recall look very similar to relative Hamming gain: For all datasets except Book, sofa has approximately 50% of the recall of basso, and in Book it is again significantly worse. For Wiki, sofa has results that are comparable to other datasets, thus, the size of Wiki datasets does not seem to affect the quality. For RSdhillon and RSzhaEtAl we observe a similar behavior as above.

Using the heuristic in sofa-auto to set the threshold typically leads to slightly worse results than setting it using line search. Given that the heuristic is usually 3–4 times as fast, there seems to be a tradeoff which version one should pick.

The running times and memory usages of the algorithms are presented in Table 3. For sofa and sofa-auto, presented is the total running time (with full line search in sofa); for basso, the presented time is the *average time for a single value* of the threshold parameter τ. Still, basso is consistently the slowest method, often by orders of magnitude. The run-times of RSdhillon and RSzhaEtAl scale well in k, since the size of the sampled subgraph and, hence, the time spent on the static computation, is largely unaffected by the choice of k.

Regarding the memory usage, basso again needs significantly more resources. sofa and sofa-auto can compute clusterings of graphs with millions of vertices and edges, while never using more than 500 MB of RAM. RSdhillon and RSzhaEtAl have relatively large memory footprints (using gigabytes of memory) due to the spectral methods they use.

Overall, the real-world experiments show that sofa can achieve results that are not too far from a static baseline method, while using only a fraction of resources.

7. RELATED WORK

Random graph models for bipartite graphs as presented in Section 2 are usually studied under the name bipartite stochastic block models (SBMs) [1]. This problem has received attention in the past [25,42] and recently it was shown that in bipartite graphs even very small clusters can be recovered [32,37,46]. Furthermore, if all clusters have size $\Omega(n)$, algorithms achieving the information-theoretically optimal recovery thresholds were presented [2,3,45]. However, these algorithms do not work in the streaming setting and (on the hardware we used) none of them would be able to process the real-world datasets we considered in Section 6.

Yun et al. [43] studied SBMs in a streaming setting and provided algorithms using $O(n^{2/3})$ bits of space when the clustering does not have to be stored explicitly. However, their algorithm does not apply to bipartite graphs and it assumes that all clusters have size $\Omega(n)$ which is unrealistic in bipartite graphs as we discussed in the introduction.

Alistarh et al. [5] consider a biclustering problem in random graphs which is similar to the one studied in this paper. They provide guarantees for recovering the left-side clusters of the graph, but they do not provide recovery guarantees for the right-side clusters. Furthermore, their data generating model is more simplistic than the one used in this paper and their

Table 3: Algorithm run-time and memory usage

k	Algorithm	Run-time in CPU minutes						Memory in GB					
		20News	Reuters	Book	Movie	Flickr	Wiki	20News	Reuters	Book	Movie	Flickr	Wiki
50	sofa-auto	2.1	3.2	1.7	45.9	9.7	14.1	0.15	0.12	0.10	0.24	0.21	0.20
	sofa	6.2	10.3	5.5	120.0	24.0	42.9	0.16	0.13	0.10	0.24	0.20	0.22
	basso	22.7	13.2	2951.8	598.1	4667.8	–	0.40	0.66	10.81	1.80	11.48	–
	RSdhillon	28.1	23.1	16.4	27.8	21.0	49.7	8.95	8.70	6.12	8.99	7.16	5.61
	RSzhaEtAl	36.0	75.2	75.4	35.9	98.5	76.3	10.72	10.43	7.26	10.73	8.63	6.57
100	sofa-auto	5.2	8.3	4.7	102.2	19.9	25.8	0.19	0.14	0.11	0.33	0.27	0.30
	sofa	15.6	25.4	16.5	311.6	52.7	70.4	0.20	0.17	0.13	0.33	0.26	0.30
	basso	24.6	13.6	3003.8	932.3	5066.0	–	0.40	0.67	10.95	1.80	11.79	–
	RSdhillon	26.9	23.7	18.1	31.2	23.0	55.5	8.96	8.70	6.09	8.99	7.20	5.54
	RSzhaEtAl	41.6	81.2	80.7	39.7	172.3	63.7	10.71	10.40	7.26	10.73	8.58	6.63
200	sofa-auto	12.2	34.8	14.2	229.1	63.7	57.1	0.25	0.18	0.13	0.49	0.36	0.43
	sofa	43.5	142.8	60.4	959.0	161.4	157.5	0.26	0.22	0.17	0.50	0.36	0.42
	basso	26.7	14.3	3097.4	1441.2	5574.1	–	0.40	0.67	10.99	1.80	12.22	–
	RSdhillon	25.3	23.1	20.8	42.2	25.8	68.3	8.96	8.68	6.00	8.98	7.18	5.57
	RSzhaEtAl	39.4	90.0	68.6	51.5	350.8	100.9	10.72	10.46	7.30	10.73	8.54	6.63

algorithm can require up to $O(kn)$ space in practice.

The BMF problem was introduced in the data mining community by Miettinen et al. [29] and has been popular in this community ever since [19, 21, 26, 28, 30, 34]. Recently, the problem was also studied in the machine learning community [22, 24, 36, 38, 39] and the theory community [7, 16]. The only streaming algorithm for BMF is by Bhattacharya et al. [8], who provided a 4-pass streaming algorithm which computes a $(1+\varepsilon)$-approximate solution for BMF. However, their algorithm is of rather theoretical nature since it requires space $O(n \cdot (\lg m)^{2k} \cdot 2^{\tilde{O}(2^{2k}/\varepsilon^2)})$ and since it uses exhaustive enumeration steps which are slow in practice. Chandran et al. [11] showed that under a standard assumption in complexity theory, any approximation algorithm for BMF requires time $2^{2^{\Omega(k)}}$ or $(mn)^{\omega(1)}$; this essentially rules out practical algorithms for BMF with approximation guarantees.

We are not aware of any algorithm which (like sofa) performs a single pass over the left-side vertices of a bipartite graph and then returns a clustering of the right-side vertices.

8. CONCLUSION

We presented sofa, the first algorithm which after single pass over the left-side vertices of a bipartite graph returns the right-side clusters using sublinear memory. We showed that after a second pass over the stream, sofa solves biclustering and BMF problems. Our experiments showed that sofa is orders of magnitude faster and more memory-efficient than a static baseline algorithm while still providing high quality results. Furthermore, we proved that under a standard random graph model, a version of sofa can find the planted clusters under a natural separation condition. In future work it will be interesting to consider streaming settings in which the edges arrive one by one. Since the main building blocks of sofa (coresets and mergeable heavy hitters data structures) extend to distributed settings, it will be interesting to make sofa distributed.

Acknowledgments

We are deeply grateful to Vincent Cohen-Addad for helpful discussions during early stages of this project. Parts of this work were done while SN was visiting Brown University and University of Easter Finland. SN gratefully acknowledges the financial support from the Doctoral Programme "Vienna Graduate School on Computational Optimization" which is funded by the Austrian Science Fund (FWF, project no. W1260-N35) and from the European Research Council under the European Community's Seventh Framework Programme (FP7/2007-2013) / ERC grant agreement No. 340506.

APPENDIX

A. THEORETICAL GUARANTEES

Proof of Theorem 1. For all proofs we assume that the conditions from Theorem 1 hold. The concrete values of the constants K_j are set inside the proofs. We start by characterising the distances of vertices from the same cluster U_i and vertices from different clusters U_i and $U_{i'}$.

LEMMA 3. *Let $u, u' \in U_i$ and let $u'' \in U_{i'}$ for $i' \neq i$. Then with probability at least $1 - m^{-3}$,*

$$\begin{aligned} d(x_u, x_{u'}) &< 1.01\left[2|V_i|p(1-p) + 2(|V \setminus V_i|)q(1-q)\right], \\ d(x_u, x_{u''}) &> 0.99[|V_i \triangle V_{i'}|(p(1-q) + q(1-p)) \\ &\quad + 2|V_i \cap V_{i'}|p(1-p) + 2|V \setminus (V_i \cup V_{i'})|q(1-q)]. \end{aligned}$$

PROOF. First, recall that the neighbors of u, u' and u'' are random variables such that if $u \in U_i$ then $\mathbf{Pr}((u, v_j) \in E) = p$, if $v_j \in V_i$, and $\mathbf{Pr}((u, v_j) \in E) = q$, if $v_j \in V \setminus V_i$. Since u's neighbors are random this implies that the vector x_u is a random vector with $\mathbf{Pr}(x_u(j) = 1) = \mathbf{Pr}((u, v_j) \in E)$. Next, observe that we can rewrite the event $\{x_u(j) \neq x_{u'}(j)\}$ as $\{x_u(j) = 1 \text{ and } x_{u'}(j) = 0\} \cup \{x_u(j) = 0 \text{ and } x_{u'}(j) = 1\}$. Together, this implies for vertices from the same cluster,

$$\mathbf{Pr}(x_u(j) \neq x_{u'}(j)) = \begin{cases} 2p(1-p), & v_j \in V_i, \\ 2q(1-q), & v_j \in V \setminus V_i. \end{cases}$$

Similarly, we obtain for vertices from different clusters,

$$\mathbf{Pr}(x_u(j) \neq x_{u''}(j)) = \begin{cases} p(1-q) + q(1-p), & v_j \in V_i \triangle V_{i'}, \\ 2p(1-p), & v_j \in V_i \cap V_{i'}, \\ 2q(1-q), & v_j \notin V_i \cup V_{i'}. \end{cases}$$

Next, using linearity of expectation we get that

$$\begin{aligned}
\mathbf{E}\left[d(x_u, x_{u'})\right] =& \sum_{j=1}^{n} \mathbf{Pr}\left(x_u(j) \neq x_{u'}(j)\right) \\
=& 2|V_i|p(1-p) + 2(|V \setminus V_i|)q(1-q), \\
\mathbf{E}\left[d(x_u, x_{u''})\right] =& |V_i \triangle V_{i'}|(p(1-q) + q(1-p)) \\
&+ 2|V_i \cap V_{i'}|p(1-p) \\
&+ 2|V \setminus (V_i \cup V_{i'})|q(1-q).
\end{aligned}$$

Since $|V_i| \geq K_3 \lg n$ and $|V_i \triangle V_{i'}| \geq K_4 s \geq K_3 K_4 \lg n$,

$$\begin{aligned}
\mathbf{E}\left[d(x_u, x_{u'})\right] &\geq 2p(1-p)|V_i| \geq 2K_3 p(1-p) \lg n, \\
\mathbf{E}\left[d(x_u, x_{u''})\right] &\geq |V_i \triangle V_{i'}|(p(1-q) + q(1-p)) \\
&\geq K_4 p(1-q)s \geq K_3 K_4 p(1-q) \lg n.
\end{aligned}$$

A Chernoff bound and setting K_3 large enough implies the lemma (we will set K_4 later independently of K_3). □

Next, we show that when setting $\alpha = 0.49 K_4 s$ in Algorithm 1, the algorithm clusters all left-side vertices correctly.

Lemma 4. *The following events hold w.h.p.: (1) When Algorithm 1 finishes, $|C| = k$ and for all i, C contains exactly one center c with $c \in U_i$. (2) For all i, there exists a center $c_i \in C$ s.t. all points $u \in U_i$ were assigned to c_i.*

Proof. First, we condition on the event from Lemma 3 occurring for each pair of vertices from U for the rest of the proof. A union bound implies that this happens with probability at least $1 - m^{-1}$.

Second, consider $u, u' \in U_i$. Then

$$\begin{aligned}
d(x_u, x_{u'}) &< 1.01\left[2sp(1-p) + 2nq(1-q)\right] \\
&\leq 1.01\left[s/2 + 2n\frac{K_1 s}{n}\right] \leq 1.01(1/2 + 2K_1)s,
\end{aligned}$$

where we used $p(1-p) \leq 1/4$ and $q \leq K_1 ps/n \leq K_1 s/n$.

Third, for $u \in U_i$ and $u'' \in U_{i'}$ for $i \neq i'$,

$$\begin{aligned}
d(x_u, x_{u''}) > 0.99[&|V_i \triangle V_{i'}|(p(1-q) + q(1-p)) \\
&+ 2|V_i \cap V_{i'}|p(1-p) + 2|V \setminus (V_i \cup V_{i'})|q(1-q)] \\
\geq 0.99[&K_4 sp(1-q) + 0 + 0] \geq 0.98 K_4 s/2,
\end{aligned}$$

where we used that $|V_i \triangle V_{i'}| \geq K_4 s$ and further $p(1-q) \geq p - K_1 p^2 s/n \geq p - K_1 p^2 \geq \frac{0.98}{0.99} \cdot \frac{1}{2}$, since $p \geq 1/2$ and since we can pick K_1 small enough to satisfy the last inequality.

Pick K_1, K_4 with $K_4 \geq \frac{2.02}{0.98}(1/2 + 2K_1)$. Then we get that $d(x_u, x_{u''}) > 0.98 K_4 s/2 \geq 1.01(1/2 + 2K_1)s > d(x_u, x_{u'})$.

Next, we show that Algorithm 1 satisfies the properties of the lemma with $\alpha = 0.98 K_4 s/2$. To prove (1), suppose a vertex $u \in U_i$ is processed and for all $c \in C$, $d(x_u, x_c) > \alpha$. Then C cannot contain any point $u' \in U_i$ (if C contained such a point, then the previous computation and the event we conditioned on imply $d(x_u, x_{u'}) \leq \alpha$). Thus, opening u as a new center is the correct choice and C contains exactly one center from U_i. To prove (2), suppose that a vertex $u \in U_i$ is processed and $d(x_u, x_c) \leq \alpha$ for some $c \in C$. The previous computation and the event we conditioned on imply that $c \in U_i$. Thus, all $u \in U_i$ are assigned to the same $c \in C$. □

Lemma 5. *With probability at least $1 - n^{-2}$, each vertex $u \in U$ has degree $O(s)$.*

Proof. Let $u \in U_i$ and let $d(u)$ be the degree of u. Then $\mathbf{E}\left[d(u)\right] = p|V_i| + q|V \setminus V_i| \leq ps + (K_1 s/n)n = O(s)$. Since $\mathbf{E}\left[d(u)\right] \geq p|V_i| \geq K_3 p \lg n$, we apply a Chernoff bound to obtain that $d(u) \in [0.99\mathbf{E}\left[d(u)\right], 1.01\mathbf{E}\left[d(u)\right]]$ with probability at least $1 - n^{-2}$ for large enough K_3. □

Now we show that Algorithm 1 indeed returns the correct right-side clusters if we set $\theta = 0.75p$.

Lemma 6. *Algorithm 1 returns clusters $\tilde{V_1}, \dots, \tilde{V_k}$ such that $\{\tilde{V_1}, \dots, \tilde{V_k}\} = \{V_1, \dots, V_k\}$ w.h.p.*

Proof. Condition on the events from Lemma 4. Let $i \in [k]$ and suppose $c \in C$ satisfies $c \in U_i$. We show $\tilde{V_c} = V_i$.

Consider the heavy hitters data structure $\mathsf{MG}(c)$. Recall that when a vertex $u \in U$ is assigned to c, we added all $j \in [n]$ to $\mathsf{MG}(c)$ with $(u, v_j) \in E$. Hence, the stream X of numbers that were processed by $\mathsf{MG}(c)$ satisfies that the frequency f_j of j is exactly $f_j = |\{u \in U_i : (u, v_j) \in E\}|$.

From the random graph model we get that $\mathbf{E}\left[f_j\right] = p|U_i|$ if $v_j \in V_i$ and $\mathbf{E}\left[f_j\right] = q|U_i|$ if $v_j \notin V_i$. Using a Chernoff bound and $|U_i| \geq K_2 \lg n$, we get that when K_2 is large enough, $f_j > 0.99p|U_i|$ if $v_j \in V_i$ and $f_j < 1.01q|U_i| \leq 0.5p|U_i|$ if $v_j \notin V_i$ with probability at least $1 - n^{-2}$. Using a union bound, we get that the previous event holds for all $j \in [n]$ simultaneously with probability at least $1 - n^{-1}$. We condition on this event for the rest of the proof.

The total number of points inserted into $\mathsf{MG}(c)$ is $|X| = \sum_{u \in U_i} d(u)$ and using Lemma 5 and a union bound, $|X| = O(|U_i|s)$ with high probability. Thus, if we run $\mathsf{MG}(c)$ with $\varepsilon = Cp/(2s)$ for some suitable constant C, we get that $\mathsf{MG}(c)$ uses space $O(1/\varepsilon) = O(s)$ and provides an approximation $\hat{f}_j$ of each f_j within additive error $\varepsilon|X| \leq 0.1p|U_i|$.

Thus, if $v_j \in V_i$ then $\hat{f}_j \geq f_j - \varepsilon|X| \geq 0.89p|U_i|$ and if $v_j \notin V_i$ then $\hat{f}_j \leq f_j + \varepsilon|X| \leq 0.6p|U_i|$. Setting $\theta = 0.75p$ we get that the algorithm satisfies $\tilde{V_c} = V_i$. □

Lemma 7. *W.h.p. the space usage of Algorithm 1 is $O(ks)$ and its running time is $O(mks)$.*

Proof. Conditioning on Lemma 4, the algorithm only stores k centers. Storing a single center takes space $O(s)$ to store its neighbors by Lemma 5. Furthermore, for a single center we need to store its heavy hitters data structure. As we argued in the proof of Lemma 6 it suffices to use the heavy hitters data structure with $O(s)$ counters for each center. Thus, the total space usage is $O(ks)$.

Observe that for each $u \in U$ the running time is dominated by computing $d = \min_{c \in C} d(x_u, x_c)$. As there are only k centers $c \in C$ and since all $u \in U$ and $c \in C$ have only $O(s)$ neighbors, we can compute d in time $O(ks)$. Merging the heavy hitters data structures can be done in constant amortized time. Thus, the total running time for the pass over the stream is $O(mks)$ since $|U| = m$. In the postprocessing step, we only spend time $O(ks)$ because each of the heavy hitters data structures only contains $O(s)$ counters. □

Proof of Proposition 2. Any algorithm to solve the biclustering problem must be able to output the planted clusters $V_1, \dots, V_k$. Suppose that each V_i consists of s vertices and that all V_i are mutually disjoint. Then there are $\binom{n}{ks}$ possibilities to pick the V_i. Thus, any algorithm that is able to return the V_i exactly must use at least $\lg \binom{n}{ks} = \Omega(\lg n^{ks}) = \Omega(ks \lg n)$ bits. Since the standard word RAM model of computation is considering words of size $\Theta(\lg n)$, this yields a lower bound of $\Omega(ks)$ space.

B. REFERENCES

[1] E. Abbe. Community detection and stochastic block models: recent developments. *CoRR*, abs/1703.10146, 2017.

[2] E. Abbe and C. Sandon. Community detection in general stochastic block models: Fundamental limits and efficient algorithms for recovery. In *FOCS*, pages 670–688, 2015.

[3] E. Abbe and C. Sandon. Recovering communities in the general stochastic block model without knowing the parameters. In *NIPS*, pages 676–684, 2015.

[4] P. K. Agarwal, G. Cormode, Z. Huang, J. M. Phillips, Z. Wei, and K. Yi. Mergeable summaries. *ACM Trans. Database Syst.*, 38(4):26:1–26:28, 2013.

[5] D. Alistarh, J. Iglesias, and M. Vojnovic. Streaming min-max hypergraph partitioning. In *NeurIPS*, pages 1900–1908, 2015.

[6] V. Arya, N. Garg, R. Khandekar, A. Meyerson, K. Munagala, and V. Pandit. Local search heuristics for k-median and facility location problems. *SIAM J. Comput.*, 33(3):544–562, 2004.

[7] F. Ban, V. Bhattiprolu, K. Bringmann, P. Kolev, E. Lee, and D. P. Woodruff. A PTAS for lp-low rank approximation. In *SODA*, pages 747–766, 2019.

[8] A. Bhattacharya, D. Goyal, R. Jaiswal, and A. Kumar. Streaming PTAS for binary ℓ_0–low rank approximation. *CoRR*, abs/1909.11744, 2019.

[9] V. Braverman, A. Meyerson, R. Ostrovsky, A. Roytman, M. Shindler, and B. Tagiku. Streaming k-means on well-clusterable data. In *SODA*, pages 26–40, 2011.

[10] T. Calders, N. Dexters, J. J. M. Gillis, and B. Goethals. Mining frequent itemsets in a stream. *Inf. Syst.*, 39:233–255, 2014.

[11] L. S. Chandran, D. Issac, and A. Karrenbauer. On the parameterized complexity of biclique cover and partition. In *IPEC*, pages 11:1–11:13, 2016.

[12] G. Cormode and S. Muthukrishnan. An improved data stream summary: the count-min sketch and its applications. *J. Algorithms*, 55(1):58–75, 2005.

[13] T. A. Davis and Y. Hu. The university of florida sparse matrix collection. *ACM Trans. Math. Softw.*, 38(1):1:1–1:25, 2011.

[14] I. S. Dhillon. Co-clustering documents and words using bipartite spectral graph partitioning. In *SIGKDD*, pages 269–274, 2001.

[15] K. Eren, M. Deveci, O. Küçüktunç, and Ü. V. Çatalyürek. A comparative analysis of biclustering algorithms for gene expression data. *Briefings in Bioinformatics*, 14(3):279–292, 2013.

[16] F. V. Fomin, P. A. Golovach, D. Lokshtanov, F. Panolan, and S. Saurabh. Approximation schemes for low-rank binary matrix approximation problems. *ACM Trans. Algorithms*, 16(1):12:1–12:39, 2020.

[17] A. Goyal, H. D. III, and S. Venkatasubramanian. Streaming for large scale NLP: language modeling. In *HLT-NAACL*, pages 512–520, 2009.

[18] F. M. Harper and J. A. Konstan. The movielens datasets: History and context. *TiiS*, 5(4):19:1–19:19, 2016.

[19] S. Hess, N. Piatkowski, and K. Morik. The trustworthy pal: Controlling the false discovery rate in boolean matrix factorization. In *SDM*, pages 405–413, 2018.

[20] Q. Huang, T. Wang, D. Tao, and X. Li. Biclustering learning of trading rules. *IEEE Trans. Cybernetics*, 45(10):2287–2298, 2015.

[21] P. Krajca and M. Trnecka. Parallelization of the grecond algorithm for boolean matrix factorization. In *ICFCA*, pages 208–222, 2019.

[22] R. Kumar, R. Panigrahy, A. Rahimi, and D. P. Woodruff. Faster algorithms for binary matrix factorization. In *ICML*, pages 3551–3559, 2019.

[23] J. Kunegis. KONECT: the koblenz network collection. In *WWW*, pages 1343–1350, 2013.

[24] L. Liang and S. Lu. Noisy and incomplete boolean matrix factorization via expectation maximization. *CoRR*, abs/1905.12766, 2019.

[25] S. H. Lim, Y. Chen, and H. Xu. A convex optimization framework for bi-clustering. In *ICML*, pages 1679–1688, 2015.

[26] C. Lucchese, S. Orlando, and R. Perego. A unifying framework for mining approximate top-k binary patterns. *IEEE Trans. Knowl. Data Eng.*, 26(12):2900–2913, 2014.

[27] S. C. Madeira and A. L. Oliveira. Biclustering algorithms for biological data analysis: A survey. *IEEE/ACM Trans. Comput. Biology Bioinform.*, 1(1):24–45, 2004.

[28] P. Miettinen. Dynamic boolean matrix factorizations. In *ICDM*, pages 519–528, 2012.

[29] P. Miettinen, T. Mielikäinen, A. Gionis, G. Das, and H. Mannila. The discrete basis problem. *IEEE Trans. Knowl. Data Eng.*, 20(10):1348–1362, 2008.

[30] P. Miettinen and J. Vreeken. MDL4BMF: minimum description length for boolean matrix factorization. *TKDD*, 8(4):18:1–18:31, 2014.

[31] J. Misra and D. Gries. Finding repeated elements. *Sci. Comput. Program.*, 2(2):143–152, 1982.

[32] S. Neumann. Bipartite stochastic block models with tiny clusters. In *NeurIPS*, pages 3871–3881, 2018.

[33] J. Orlin. Contentment in graph theory: covering graphs with cliques. *Indagationes Mathematicae*, 80(5):406–424, 1977.

[34] P. Osicka and M. Trnecka. Boolean matrix decomposition by formal concept sampling. In *CIKM*, pages 2243–2246, 2017.

[35] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *JMLR*, 12:2825–2830, 2011.

[36] S. Ravanbakhsh, B. Póczos, and R. Greiner. Boolean matrix factorization and noisy completion via message passing. In *ICML*, pages 945–954, 2016.

[37] Z. S. Razaee, A. A. Amini, and J. J. Li. Matched bipartite block model with covariates. *J. Mach. Learn. Res.*, 20:34:1–34:44, 2019.

[38] T. Rukat, C. C. Holmes, M. K. Titsias, and C. Yau. Bayesian boolean matrix factorisation. In *ICML*, pages 2969–2978, 2017.

[39] T. Rukat, C. C. Holmes, and C. Yau. Probabilistic

boolean tensor decomposition. In *ICML*, pages 4410–4419, 2018.

[40] M. Shindler, A. Wong, and A. W. Meyerson. Fast and accurate k-means for large datasets. In *NeurIPS*, pages 2375–2383, 2011.

[41] J. Wang, S. Levy, R. Wang, A. Kulshrestha, and R. Rabbany. SGP: spotting groups polluting the online political discourse. *CoRR*, abs/1910.07130, 2019.

[42] J. Xu, R. Wu, K. Zhu, B. E. Hajek, R. Srikant, and L. Ying. Jointly clustering rows and columns of binary matrices: algorithms and trade-offs. In *SIGMETRICS*, pages 29–41, 2014.

[43] S. Yun, M. Lelarge, and A. Proutière. Streaming, memory limited algorithms for community detection. In *NeurIPS*, pages 3167–3175, 2014.

[44] H. Zha, X. He, C. H. Q. Ding, M. Gu, and H. D. Simon. Bipartite graph partitioning and data clustering. In *CIKM*, pages 25–32, 2001.

[45] Z. Zhou and A. A. Amini. Optimal bipartite network clustering. *CoRR*, abs/1803.06031, 2018.

[46] Z. Zhou and A. A. Amini. Analysis of spectral clustering algorithms for community detection: the general bipartite setting. *JMLR*, 20:47:1–47:47, 2019.

[47] C. Ziegler, S. M. McNee, J. A. Konstan, and G. Lausen. Improving recommendation lists through topic diversification. In *WWW*, pages 22–32, 2005.