Streaming Set Cover in Practice

Michael Barlow[*] Christian Konrad[*] Charana Nandasena[†]

Abstract

State-of-the-art practical algorithms for solving large Set Cover instances can all be regarded as variants of the Greedy Set Cover algorithm. These algorithms maintain the input sets in memory, which yields a substantial memory footprint. In particular, in the context of massive inputs, these sets may need to be maintained on the hard disk or on external memory, and, consequently, access to these sets is slow.

In this paper, we demonstrate that simple one-pass algorithms with small memory footprints are able to compete with the more involved Greedy-like algorithms for Set Cover in practice. Our experiments show that a recent Set Cover streaming algorithm by Emek and Rosén [ACM Trans. on Alg. 2016] produces covers whose sizes are on average within 8% of those produced by state-of-the-art algorithms, while using between 10 and 73 times less memory.

We also provide a theoretical analysis of an extension of the Emek-Rosén algorithm to multiple passes and demonstrate that multiple passes allow us to further reduce cover sizes in practice.

1 Introduction

Given a collection $\mathcal{S}$ of m sets containing elements from some universe $\mathcal{U}$ of size n, the Set Cover problem is to find a smallest subcollection $\mathcal{A} \subseteq \mathcal{S}$ that covers the entire universe, i.e., $\bigcup_{S \in \mathcal{A}} S = \mathcal{U}$. This fundamental problem arises at the heart of many practical applications, including document retrieval [1], test suite reduction [28] and protein interaction prediction [18] to name a few. The ubiquity of Set Cover stems from the abstract nature of its input. In the domain of connected data, Set Cover can be equivalently expressed in terms of hypergraphs, where vertices comprise the universe and hyperedges comprise the set collection. For graphs, a solution to Set Cover over the collection of edges results in a minimum Edge Cover, and a solution of Set Cover over the inclusive neighborhoods of the vertices results in a minimum Dominating Set.

The Set Cover problem is known to be NP-hard to solve exactly [20], and also NP-hard to approximate to within a factor of $(1 - o(1)) \ln n$ [8, 11]. The Greedy Set Cover algorithm, which produces a $(\ln n - \ln \ln n + \Theta(1))$-approximate solution [25], is therefore essentially optimal from a theoretical perspective. Intuitively, Greedy repeatedly adds to the solution the set with the most as-yet-uncovered elements, terminating when the universe is covered. This simple algorithm clearly runs in polynomial time, and has been shown to perform well in practice [15, 16]. However, the non-trivial task of efficiently finding the set with the most uncovered elements means Greedy (1) requires space linear to the size of input; (2) has an arbitrary memory access pattern; and (3) proves awkward to implement in practice, requiring multiple ancillary data structures. As arbitrary access to disk is typically slow,[1] Greedy does not scale well to data sets whose sizes exceed the capacity of the available Random Access Memory (RAM). For this reason, a focus of recent research has been to find memory efficient algorithms with close-to-optimal approximation guarantees.

Cormode et al. [6] propose Disk Friendly Greedy (DFG), a variant of Greedy which instead adds sets to the solution if they have an uncovered element count that is close to maximal. This relaxation does not have a significant impact on the approximation factor, and affords a mostly sequential memory access pattern which greatly improves the efficiency of interactions with disk. Cormode et al. also demonstrate that DFG performs well in practice, producing solutions of a similar size to Greedy and, for the larger data sets, in a fraction of the time. Lim et al. [23] explore the in-memory implementation options of Greedy in greater detail, and introduce Lazy Greedy, a variant of Greedy which has a smaller memory footprint at the cost of occasional set difference recomputations. An equivalent algorithm is also described in [26]. Though these algorithms have similar approximation factors to Greedy, they still require access to space linear to the size of input, be it RAM, disk or external memory. As accessing disk is, in most circumstances, orders of magnitude slower than accessing RAM, the performance of Greedy-like algorithms declines when applied to problem instances whose size necessitates disk residence, even if the access pattern is mostly sequential.

To quickly process such data sets, we can instead consider Set Cover under the *data streaming model*,

[*]Department of Computer Science, University of Bristol, UK, {michael.barlow,christian.konrad}@bristol.ac.uk

[†]Melbourne School of Engineering, University of Melbourne, Australia, anandasena@student.unimelb.edu.au

[1]We use *disk* as a general term to refer to the class of storage devices secondary to RAM.

where algorithms are permitted only sequential access to the problem instance set collection, and must produce a solution using working memory of size sublinear to the size of the input. In 2014, Emek and Rosén [9] (see also [10]) presented the first one-pass streaming algorithm for Set Cover, which gives an $O(\sqrt{n})$-approximate solution using $\widetilde{O}(n)$ space.[2] This work received considerable attention in the algorithms research community and led to a large number of follow-up works (e.g., [7, 3, 17, 4, 2, 19]), giving algorithms with different characteristics and space lower bounds.

Small-space streaming algorithms for Set Cover inherently have a worse approximation guarantee than Greedy and the aforementioned Greedy-like state-of-the-art algorithms. For example, it is known that p-pass algorithms (for any constant $p \geq 1$) with space complexity $\widetilde{O}(n)$ cannot have an approximation ratio smaller than $\Theta(n^{\frac{1}{p+1}})$ [4], which implies that the approximation factor of $O(\sqrt{n})$ of the ($\widetilde{O}(n)$-space) Emek-Rosén algorithm is optimal. Improving on the approximation factor of $O(\sqrt{n})$ in a single pass requires much more space: one-pass C-approximation algorithms, for $C = o(\sqrt{n})$, require $\Omega(nm/C)$ space [3].

In this paper, we ask whether the recent (theoretical) streaming algorithms for Set Cover are able to compete with the state-of-the-art algorithms in practice. The obvious advantage of the streaming approach is the sublinear memory usage. In addition, streaming algorithms are typically conceptually simple which gives reason to hope that such algorithms yield fast runtimes in practice. As mentioned previously, streaming algorithms for Set Cover have worse theoretical approximation guarantees than the state-of-the-art Greedy-like algorithms, though past empirical Set Cover evaluations observed that theoretical guarantees do not well predict practical solution sizes [15, 16]. Our aim is therefore to determine whether this theoretical solution size difference is significant in practice, and to learn the extent of working memory savings afforded by employing a streaming approach. Depending on the application, savings in RAM usage may well outweigh marginally larger solutions.

Our Contributions In this paper, we conduct an empirical evaluation of the Emek-Rosén algorithm [10] and compare its performance on practical instances to the DFG algorithm by Cormode et al. [6]. To the best of our knowledge, we are the first to evaluate one of the recent theoretical Set Cover streaming algorithms. We remark that, among the known streaming algorithms, the Emek-Rosén algorithm and the Progressive Greedy algorithm by Chakrabarti and Wirth [4] stand out as potentially competitive practical algorithms, since they are deterministic, simple, and use the least space. We also include Progressive Greedy in our experiments, showing that the produced covers are larger than those obtained by the Emek-Rosén algorithm.

[2] We write $\widetilde{O}(\cdot)$ to mean $O(\cdot)$ with poly log factors suppressed.

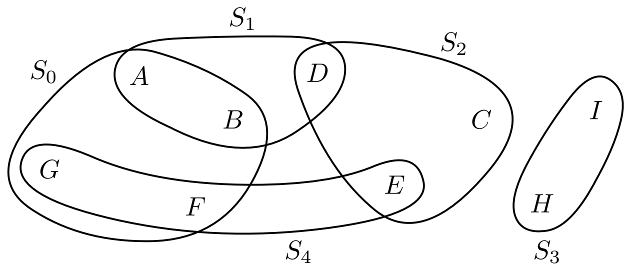


Figure 1: An example instance of Set Cover. Here, $\mathcal{U} = \{A, B, C, D, E, F, G, H, I\}$ and $\mathcal{S} = \{S_0, S_1, S_2, S_3, S_4\}$, where $S_0 = \{A, B, F, G\}$, $S_1 = \{A, B, D\}$, $S_2 = \{C, D, E\}$, $S_3 = \{H, I\}$ and $S_4 = \{E, F, G\}$.

Our experiments show that the Emek-Rosén algorithm produces solution sizes which are on average 8% larger than those produced by DFG, while using only 1.4% to 9.9% of the memory required by DFG on our input data sets. In passing, we also show that the approximation ratio of the Emek-Rosén algorithm can be expressed in terms of Δ, the size of the largest set, showing that the algorithm has an approximation ratio of at most $4\sqrt{2}\sqrt{\Delta}$. We note that in practical instances Δ is typically much smaller than n and is therefore the more relevant parameter. Last, we extend the Emek-Rosén algorithm to multiple passes. We show that p passes, for constant p, yield an approximation factor of $O(\Delta^{\frac{1}{p+1}})$, which is known to be optimal up to constant factors [4]. We demonstrate that multiple passes allow us to further narrow the gap in solution sizes between the Emek-Rosén algorithm and DFG in practice.

2 Preliminaries

2.1 Problem Definition An instance of the Set Cover problem is defined over a universe $\mathcal{U}$, where $|\mathcal{U}| = n$. Given a collection $\mathcal{S}$ of m sets $\{S_0, S_1, ..., S_{m-1}\}$, where $S_i \subseteq \mathcal{U}$ for all $0 \leq i < m$ and $\bigcup_{S \in \mathcal{S}} = \mathcal{U}$, the aim is to find a smallest set $\mathcal{A} \subseteq \mathcal{S}$ such that $\bigcup_{S \in \mathcal{A}} S = \mathcal{U}$. The size of the largest set in the instance is denoted by Δ, and the combined size of all sets is denoted by $M = \sum_{S \in \mathcal{S}} |S|$. We hereafter assume, without loss of generality, that each set $S \in \mathcal{S}$ is associated with an identifier $\text{id}(S)$ of size $O(\log m)$ bits. A solution to Set Cover is therefore given as a collection $\mathcal{I}$ of identifiers, where $\{S \in \mathcal{S} \mid \mathrm{id}(S) \in \mathcal{I}\} = \mathcal{A}$. We refer to the optimal solution to any given Set Cover instance as Opt.

Figure 1 shows an example instance of the Set Cover problem with $n = 9$, $m = 5$, $\Delta = 4$ and $M = 15$. In this instance, $\{S_1, S_2, S_3, S_4\}$ is a cover of $\mathcal{U}$, however the smallest cover, and hence the optimal solution to the Set Cover problem in this context, is $\{S_0, S_2, S_3\}$.

2.2 Streaming A *stream* is a sequence of data points which arrive as input to an algorithm one at a time. Since the size of this stream is often unknown (size is typically only discovered upon arrival of the final data point) or very large, a complete history of the stream cannot necessarily be stored in memory. For this reason, *streaming algorithms* maintain some internal summary of the visited portion of the input in space sublinear—and ideally polylogarithmic—to the size of the stream. As the whole input does not reside in memory, a second pass over the stream would be required to access data points which have already been visited; additional passes such as this are permitted in the context of *multi-pass* algorithms. In contrast, a *one-pass* algorithm is constrained to a single traversal of the stream.

Considering the Set Cover problem under the streaming model, the set collection $\mathcal{S}$ forms the input stream, with the set S_t arriving at time t. It is not always possible to approximate Set Cover in polylogarithmic space; we realise this by observing that $O(n \log m)$ bits of space is required simply to store a solution. We therefore consider an algorithm to be space efficient if it uses only $O(n \operatorname{polylog}(n, m)) = \widetilde{O}(n)$ bits of space, and refer to such an algorithm as a *semi-streaming algorithm* [12, 10].

Algorithm 1: Greedy Set Cover

```
  input  : A collection of sets $\mathcal{S}$
  output: A covering collection of identifiers $\mathcal{I}$
1 $\mathcal{I}$, $C \leftarrow \emptyset$
2 while $C \neq \mathcal{U}$ do
3     $i \leftarrow i'$ which maximises $|S_{i'} \setminus C|$
4     $\mathcal{I} \leftarrow \mathcal{I} \cup \{\mathrm{id}(S_i)\}$
5     $C \leftarrow C \cup S_i$
6 return $\mathcal{I}$
```

3 Greedy and Greedy-Like Algorithms

The Greedy algorithm for Set Cover is formally given in Algorithm 1. The algorithm maintains a collection C of covered elements, initialised to the empty set. Until every universe element is present in C, the algorithm finds the set S_i containing the most uncovered elements, then adds the identifier of S_i to $\mathcal{I}$, and updates C to include all elements in S_i. This simple algorithm leaves implementation details unspecified, including how to find the set containing the most uncovered elements. This task is non-trivial, as the uncovered element count of each set may change after the collection of covered elements is updated.

A simple approach to this task would be to repeatedly iterate through the problem instance set collection, noting the identifier of the set with the most uncovered elements for each pass. This method accesses memory sequentially, however, in the worst case, m traversals of the problem instance would be required. Under the reasonable assumption that membership of C can be checked in constant time, this method would require $O(Mm)$ time—this is clearly inefficient.

An alternative solution would be to maintain a priority queue storing the uncovered element counts of each set. At each execution step, the set at the head of the queue would be added to the solution, and, after updating C, all other sets which contain those elements freshly included in C would need their priority queue entries updating. This can be achieved with an inverted index: a precomputed data structure which stores for each element the identifiers of the sets in which it is included. This inverted index requires $O(M \log m)$ additional bits of space, effectively doubling the space usage of the algorithm. Assuming that each change to the priority queue occurs in $O(\log m)$ time, this approach has a time complexity of $O(M \log m)$, which can be further improved to $O(M)$ with a more involved implementation [23].

Lazy Greedy, a variant of Greedy first described by Lim et al. [23] and later reinvented by Stergiou and Tsioutsiouliklis [26], forgoes the use of an inverted index by updating the priority queue only when necessary. At each execution step, the uncovered element count of the set at the head of the priority queue is checked; if the value stored in the priority queue is accurate, then this set is added to the solution and C is updated. If not, then the set is reinserted into the priority queue with the updated uncovered element count. This process repeats until all elements are covered. Though the space required by this approach is roughly half that of the inverted index implementation, the number of main loop iterations is no longer bounded by m, and the total number of times that sets are compared with C is no longer bounded by M. Lim et al. show that there exists an adverserial Set Cover instance of size M on which Lazy Greedy expends $\Theta(M^{4/3})$ time, but observe that such instances rarely occur in practice. They also demonstrate empirically that Lazy Greedy runs faster than the inverted index approach on typical large problem instances.

Both the inverted index and lazy implementations of Greedy have a predominantly arbitrary memory access pattern. While this is fine for data sets small enough to reside fully in RAM, the overheads associated with arbitrary access to disk mean these algorithms run painfully slowly on larger-than-RAM data sets. Cormode et al. [6] address this problem, introducing Disk Friendly Greedy (DFG), a Greedy-like algorithm which accesses memory in large contiguous chunks. DFG employs a bucketing approach with granularity controlled by an input parameter p. In a preprocessing sweep of the input, each set is placed in a bucket, such that the sets in the kth bucket have size between p^k (inclusive) and p^{k+1} (exclusive). Depending on the size of the problem instance, these buckets can be stored either in RAM or in files on disk. DFG then processes each bucket in sequence, starting with the bucket holding the largest set. For each set in the bucket, the uncovered element count is first computed, after which the set is added to the solution if this value is at least p^k. If not, then the uncovered elements are removed from the set, and the set is then demoted to the bucket associated with its new cardinality. This repeats until only the bucket with $k = 0$ remains; sets in this bucket are added to the solution if they contain a non-zero number of uncovered elements. Though DFG is no longer guaranteed to choose the set with the most uncovered elements at each execution step, it has an approximation factor of $1 + p \ln n$, which is only marginally worse than the approximation guarantee of Greedy. In practice, the solutions produced by DFG are very similar in size to those given by Greedy [6].

Importantly, the largely sequential memory access pattern of DFG means interactions with disk are quite efficient, so DFG scales to data sets whose sizes exceed the capacity of RAM better than the other approaches described. Despite this, DFG still maintains an ancillary data structure of size linear to the size of input, and when this structure resides on disk, accessing it is inherently slower than accessing RAM (sometimes by orders of magnitude), even when accessing sequentially.

4 Emek-Rosén Algorithm

We now turn our attention to streaming algorithms, which are able to solve larger-than-RAM Set Cover instances without the use of disk. Specifically, we consider the recent theoretical one-pass semi-streaming algorithm introduced by Emek and Rosén [10]. This algorithm is designed to solve the *weighted* Set Cover problem, a generalisation of Set Cover which admits element costs and set benefits. As we address the unweighted Set Cover problem here, we describe a version of the algorithm that is simplified for our context. The algorithm is conceptually simple and easy to implement, and is well suited to problem instances which naturally take the form of a stream, such as real-time data or data received over a network.

$x \in S_t$	$\underline{\text{A}}$	B	$\underline{\text{C}}$	$\underline{\text{D}}$	$\underline{\text{E}}$	F	G	$\underline{\text{H}}$
$\text{eff}_t(x)$	0	4	-1	2	2	4	3	1

Figure 2: Effective subset example.

The Emek-Rosén algorithm uses a simple heuristic to build a small in-memory summary of the sets as they arrive in a stream. For each element $x \in \mathcal{U}$, the algorithm maintains an integer *effectiveness* and an *effectiveness identifier*, denoted $\text{eff}(x)$ and $\text{eid}(x)$ respectively. Intuitively, $\text{eid}(x)$ stores the identifier of the set which is currently covering x, and $\text{eff}(x)$ aims to quantify the quality of this set. As the values of $\text{eid}(x)$ and $\text{eff}(x)$ change throughout execution, we define $\text{eid}_t(x)$ and $\text{eff}_t(x)$ to be the values of $\text{eid}(x)$ and $\text{eff}(x)$ respectively at time t, i.e., just before the set $S_t \in \mathcal{S}$ is processed. The following two definitions are central to the algorithm.

Definition 4.1. (Level) *The level of some set S is defined as*

$$\text{lev}(S) = \lceil \log_2 |S| \rceil.$$

Definition 4.2. (Effective subset) *A subset $T \subseteq S_t$ is said to be effective at time t if and only if all elements in T have an effectiveness strictly less than the level of T. Formally,*

$$T \subseteq_{\text{eff}_t} S_t \quad \text{iff.} \quad \forall x \in T,\ \text{eff}_t(x) < \text{lev}(T).$$

Figure 2 shows an example of an effective subset. Here, $S_t = \{A, B, C, D, E, F, G, H\}$, and the effectiveness values corresponding to these elements are given. The underlined elements together form the effective subset $\{A, C, D, E, H\}$. Indeed, this is the largest effective subset in this example: if any other elements were added to this subset, the level would not increase, and thus the subset would cease to be effective.

Rather than returning a solution as a list of identifiers, the algorithm returns a *cover certificate* χ, which is a mapping from universe elements to set identifiers. More formally, χ is a total function with domain $\mathcal{U}$ and codomain $\{\text{id}(S) \mid S \in \mathcal{S}\}$ where if $\chi(x) = \text{id}(S)$, then $x \in S$. The image of χ, which is given by

$$\text{Im}(\chi) = \{\text{id}(S) \mid \exists x \in \mathcal{U} \text{ such that } \chi(x) = \text{id}(S)\},$$

is equivalent to $\mathcal{I}$, the set of identifiers which together cover $\mathcal{S}$.

The Emek-Rosén algorithm is formally given in Algorithm 2. Execution starts by initialising the effectiveness and effectiveness identifier values; this step is

Algorithm 2: Emek-Rosén Set Cover

```
input : A set stream $\mathcal{S}$
output: A cover certificate and an effectiveness map
1 $\forall x \in \mathcal{U}$: $\mathrm{eid}(x) \leftarrow$ NULL; $\mathrm{eff}(x) \leftarrow -1$
2 for $t \leftarrow 0$ to $m-1$ do
3     Read the set $S_t$ from $\mathcal{S}$
4     $T \leftarrow T' \subseteq_{\mathrm{eff}_t} S_t$ which maximises $|T'|$
5     foreach $x \in T$ do
6         $\mathrm{eid}(x) \leftarrow \mathrm{id}(S_t)$
7         $\mathrm{eff}(x) \leftarrow \mathrm{lev}(T)$
8 return $\mathrm{eid}(\cdot)$ and $\mathrm{eff}(\cdot)$
```

done implicitly in practice. Then, sets are read from the stream one by one, with the set S_t being read at time t. When considering the set S_t, an effective subset $T \subseteq_{\mathrm{eff}_t} S_t$ is first found with maximum cardinality, after which for each $x \in T$ the effectiveness of x is set to the level of T, and the effectiveness identifier is set to that of S_t. Intuitively, this means the set S_t is only added to the cover certificate if it contains a subset $T \subseteq S_t$ where the size of T is at least a factor of 2 greater than the size of the subsets previously designated to covering the elements in T. The algorithm proceeds in this way until the stream terminates, at which point it returns the resulting cover certificate (alongside the final effectiveness values if desired).

To find the largest effective subset $T \subseteq S_t$, Emek and Rosén present Observation 4.1 and suggest sorting S_t by descending effectiveness.

Observation 4.1. *If $T \subseteq_{\mathrm{eff}_t} S_t$ and $x \in T$, then $T \cup \{y\}$ is effective at time t for every $y \in S_t$ such that $\mathrm{eff}_t(y) \leq \mathrm{eff}_t(x)$.*

When subsequently iterating through the sorted list, the first encountered effective element defines the starting point from which the list tail equals T. Due to the initial sort, this method processes the set S_t in $O(|S_t| \log |S_t|)$ time. Simplifying the Emek-Rosén algorithm to our unweighted context allows us to improve upon this asymptotic bound. To this end, we establish two further observations, the first of which does not apply in the context of weighted Set Cover.

Observation 4.2. *For all $0 \leq t < m$ and $x \in \mathcal{U}$, we have $-1 \leq \mathrm{eff}_t(x) \leq \lceil \log_2 \Delta \rceil$. Furthermore, $\mathrm{eff}_t(x)$ is integral.*

Observation 4.3. *If $x \in S_t$ and $\mathrm{eff}_t(x) \geq \mathrm{lev}(S_t)$, there exists no subset $T \subseteq_{\mathrm{eff}_t} S_t$ where $x \in T$.*

By Observation 4.2, effectiveness values are integral and bounded from above and below, so we can find the maximal effective subset $T \subseteq S_t$ in linear time by iterating over the frequency distribution of the effectiveness values in S_t. Specifically, a counter array of size $\mathrm{lev}(S_t)+1$ can be populated with an initial pass over S_t, such that the $(i+1)^{\text{th}}$ value of the array corresponds to the number of elements in S_t with effectiveness i (the offset accounts for effectiveness values of -1). Per Observation 4.3, any effectiveness which is beyond the bounds of this counter array can safely be omitted. Then, to obtain a mapping of effectiveness values to would-be subset sizes, a cumulative sum of this counter array can be computed. With this mapping, the largest effectiveness value which is exceeded by the level of its corresponding sum becomes the critical effectiveness; in a final pass over S_t, all elements whose effectiveness is at most this critical value are added to the resulting subset. Finding the maximal effective subset with this asymptotically favourable method allows the Emek-Rosén method to run in $O(M)$ time. We also note that the additional space required by this approach is negligible: to process a set, a number of bits polylogarithmic to the size of the set are required.

4.1 Approximation Factor

Applying the Emek-Rosén algorithm to the simplified context of unweighted Set Cover also allows for a slightly improved approximation factor. To see this, we reuse two key lemmas given by Emek and Rosén in their analysis of the algorithm [10]. We start by introducing relevant definitions, in which $\mathrm{eff}_\infty(x)$ denotes the value of $\mathrm{eff}(x)$ for some $x \in \mathcal{U}$ upon termination of the input stream.

Definition 4.3. ($I(r)$) *For some $r \in \mathbb{Z}$, let*

$$I(r) = \{x \in \mathcal{U} \mid \mathrm{eff}_\infty(x) = r\}.$$

Definition 4.4. ($S(r)$) *For some $r \in \mathbb{Z}$, let*

$$S(r) = \{S \in \mathcal{S} \mid \exists x \in I(r) \text{ s.t. } \mathrm{eid}(x) = \mathrm{id}(S)\}.$$

Definition 4.3 and Definition 4.4 are extended to the real interval $[a, b]$ as follows:

$$I([a,b]) = \{x \in \mathcal{U} \mid a \leq \mathrm{eff}_\infty(x) \leq b\}, \text{ and}$$
$$S([a,b]) = \{S \in \mathcal{S} \mid \exists x \in I([a,b]) \text{ s.t. } \mathrm{eid}(x) = \mathrm{id}(S)\}.$$

The real intervals $(-\infty, r]$ and (r, ∞) are denoted by $\leq r$ and $> r$ respectively. We now give the two key lemmas, which have been modified to be defined over the real numbers. It is easy to see that the original proof of the first lemma by Emek and Rosén also applies to real numbers without modifications. Besides allowing for real numbers, the second lemma has also been simplified by observing that $|\mathsf{Opt}| \geq n/\Delta$ and $|I(>\lceil \log_2 \Delta \rceil)| = 0$. For completeness, we provide a proof of the second lemma in Appendix A.

LEMMA 4.1. *For some $r \in \mathbb{R}$,*

$$|I(\leq r)| < 2^{r+1} \cdot |\mathsf{Opt}|.$$

Proof. See [10]. ◻

LEMMA 4.2. *For some $r \in \mathbb{R}$,*

$$|S(>r)| < \left(\frac{\Delta}{2^{r-2}} - 1\right) \cdot |\mathsf{Opt}|.$$

Proof. See Appendix A. ◻

The approximation factor can now be established.

THEOREM 4.1. *The cardinality of the solution produced by the* EMEK-ROSÉN *algorithm is within a factor $4\sqrt{2}\sqrt{\Delta}$ of $|\mathsf{Opt}|$.*

Proof. Consider some $r \in \mathbb{R}$. By combining the bounds of the solution subsets addressed in Lemma 4.1 and Lemma 4.2, and noting that $|S(r)| \leq |I(r)|$ for all r, the cover certificate χ returned by the EMEK-ROSÉN algorithm satisfies

$$|\mathrm{Im}(\chi)| < \left(2^{r+1} + \frac{\Delta}{2^{r-2}} - 1\right) \cdot |\mathsf{Opt}|.$$

This approximation factor is minimised, i.e.,

$$\frac{\partial}{\partial r}\left(2^{r+1} + \frac{\Delta}{2^{r-2}} - 1\right) = \left(2^{r+1} - \frac{\Delta}{2^{r-2}}\right)\ln 2 = 0,$$

when $r = \log_2 \sqrt{2\Delta}$. With this r, it follows that

$$|\mathrm{Im}(\chi)| < \left(4\sqrt{2}\sqrt{\Delta} - 1\right) \cdot |\mathsf{Opt}|.$$ ◻

5 Evaluation

We now demonstrate the practical performance of the EMEK-ROSÉN algorithm on typical Set Cover instances.

5.1 Data Sets For our empirical analysis, we use six benchmark data sets, detailed in Table 1. Of the six files, accidents.dat, kosarak.dat and webdocs.dat come from the Frequent Itemset Mining Dataset Repository and have previously been used by Cormode et al. [6] for the evaluation of the DFG algorithm.[3] The remaining three come from the Stanford Large Network Dataset Collection.[4] All files have been modified to fit the same format: a text file with one set to a line (terminated with a line feed), where the elements are space delimited integers. Additionally, the elements of each data set were mapped in order of appearance to the contiguous interval $[1, n]$ for consistency. The twitter.dat and friendster.dat files, which originally encoded (directed and undirected, respectively) graphs as edge lists, have both been reformatted to Set Cover instances whose solutions are a Dominating Set of the respective networks. We define for some set S the value $\mathrm{id}(S)$ to be its position in the problem instance file, indexed from 0; a solution can therefore be given in a text file as a newline-separated list of set indices.

[3]`http://fimi.uantwerpen.be/data/`
[4]`https://snap.stanford.edu/data/`

5.2 Experimental Setup C++ implementations of RAM-based DFG, disk-based DFG and the EMEK-ROSÉN algorithm were prepared; these were compiled with G++ using the `-O2` optimisation flag. All experiments were performed on a 2.4GHz Intel Core i5 machine running macOS Mojave 10.14.6, with 4 cores, 256KB L2 cache (per core), 6MB L3 cache, 16GB RAM and 512GB SSD.

To ensure only the required resources are used during execution, the implementations assume prior knowledge of the necessary problem instance parameters. The universe elements and set indices are stored as 4-byte integers and, for the EMEK-ROSÉN implementation, effectiveness values are stored as 1-byte integers. The EMEK-ROSÉN implementation computes a maximal effective subset using the linear time approach we describe in Section 4. A bit vector is used to maintain the set of covered elements for the DFG algorithms. For each algorithm, we measured the output solution size, total execution time, and peak RAM usage. The execution time encompasses all necessary algorithm steps, including reading the problem instance from disk, the DFG bucketing preprocess, and the time taken to extract a list of identifiers from the cover certificate given by the EMEK-ROSÉN algorithm. The non-essential task of writing the solution to disk is excluded from the total execution time for all algorithms.

5.3 Results We present the results of a comparison between RAM-based DFG (with $p = 1.001$) and the EMEK-ROSÉN algorithm in Table 2. For the largest file (friendster.dat), RAM-based DFG had not found a solution after 6 hours. This is as a result of the problem instance exceeding the capacity of RAM: the abstraction to virtual memory means the implementation effectively uses portions of disk as an extension to RAM, at the cost of significantly slower arbitrary access.

From these results, some clear patterns emerge. Firstly, solutions produced by the EMEK-ROSÉN algorithm are of a slightly lower quality than those given by DFG: EMEK-ROSÉN covers are 18% larger in the worst case (accidents.dat), 2% larger in the best case (webdocs.dat), and 8% larger in the (geometric) average case. However, the differences in resource usage are

File name	Credit	File size (MB)	n	m	M	Δ
accidents.dat	[14, 13]	35.17	468	340 183	11 500 870	51
kosarak.dat	[14]	32.05	41 270	990 002	8 018 988	2497
orkut-cmty.dat	[22, 24, 27]	805.44	2 322 299	15 301 901	107 080 530	9120
webdocs.dat	[14, 5]	1481.89	5 267 656	1 692 082	299 887 139	71 472
twitter.dat	[22, 21]	12 576.85	41 652 230	40 103 281	1 508 468 165	2 997 470
friendster.dat	[22, 27]	32 353.29	65 608 366	65 608 366	3 677 742 636	5215

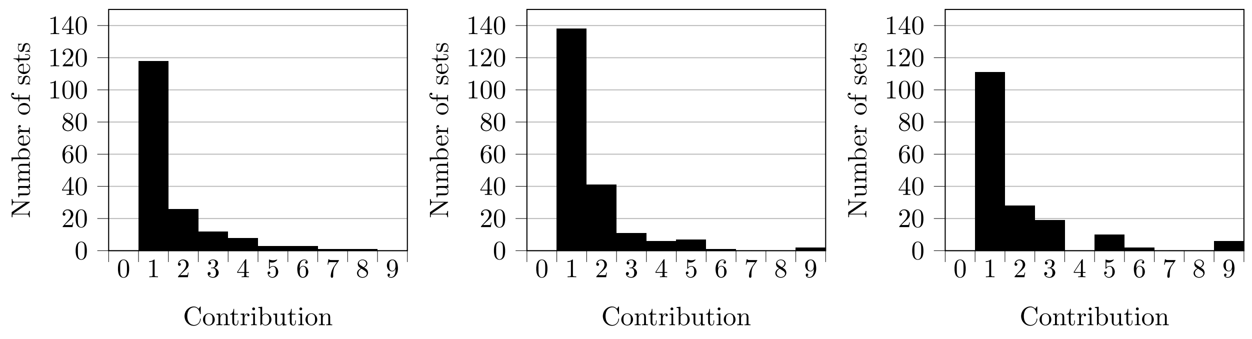
Table 1: Benchmark data set statistics.

File name	Cover size		Time (s)		Peak RAM (MB)	
	DFG	EMEK-ROSÉN	DFG	EMEK-ROSÉN	DFG	EMEK-ROSÉN
accidents.dat	181	213	1.43	0.72	66.65	0.91
kosarak.dat	17 741	18 618	1.03	0.79	75.45	2.37
orkut-cmty.dat	149 244	158 439	15.44	12.35	1094.59	21.67
webdocs.dat	406 338	413 819	18.39	15.51	1401.55	56.04
twitter.dat	9 246 029	9 955 112	213.48	158.35	8044.29	797.70
friendster.dat	-	13 310 036	-	367.52	-	1183.56

Table 2: A comparison between RAM-based DFG ($p = 1.001$) and the EMEK-ROSÉN algorithm.

File name	Cover size	Time (s)	Peak RAM (MB)	Peak disk (MB)	Disk I/O (MB)
accidents.dat	183	2.55	1.22	51.62	173.65
kosarak.dat	17 746	1.57	2.16	40.00	86.41
orkut-cmty.dat	149 239	21.66	4.69	551.03	1404.53
webdocs.dat	406 375	27.69	8.81	1213.09	2698.33
twitter.dat	9 246 096	238.25	146.69	6356.30	17 753.60
friendster.dat	10 616 833	670.17	120.15	15 373.80	48 667.50

Table 3: Disk-based DFG results ($p = 1.065$).



(a) DFG ($p = 1.001$)
(b) EMEK-ROSÉN
(c) MULTI-PASS EMEK-ROSÉN

Figure 3: Cover contribution distributions, truncated at 10, for the file accidents.dat.

more pronounced, strongly favouring the Emek-Rosén algorithm. Emek-Rosén used 73 times less RAM than DFG in the best case (accidents.dat), and 10 times less RAM in the worst case (twitter.dat), which is still an improvement of an order of magnitude. Emek-Rosén was also successfully able to process friendster.dat—which was intractable for RAM-based DFG on our machine—using only 1.2GB of RAM. In terms of speed, Emek-Rosén is consistently faster than DFG, but only by a small margin. Specifically, the Emek-Rosén algorithm finds a solution in 50% of the time taken by DFG in the best case (accidents.dat), 84% of the time taken by DFG in the worst case (webdocs.dat), and 72% of the time taken by DFG in the average case.

We also applied disk-based DFG—which stores the set buckets in files, with one file per bucket—to each of the problem instances, and present the results in Table 3. To guarantee that the limit of 253 open files (imposed by the operating system) is not exceeded during execution, we select $p = 1.065$ which satisfies $p > \Delta^{1/253}$ for all values of Δ in the benchmark data set collection. In addition to the usual metrics, we also measure peak disk usage (discounting the storage of the problem instance and solution) and *disk I/O*: the number of bytes read from and written to disk (not including the initial reading of the problem instance, or the writing of the solution). In the context of disk-based algorithms, disk I/O serves as a better indicator of performance than time, as it is invariant to hardware. At the expense of using 13 times more memory than Emek-Rosén and exchanging 49GB of data with the disk during execution, disk-based DFG was able to process friendster.dat and produce a solution whose size is 20% smaller than that given by Emek-Rosén.

6 Multi-Pass Emek-Rosén

In comparison to DFG, the Emek-Rosén algorithm typically includes in the solution a higher number of sets which cover only a few elements, leading to larger cover sizes. To demonstrate this, it is useful to first define the *contribution* of a set to be the number of elements for which this set is the designated coverer. In the context of Emek-Rosén, the contribution of a set S can be found by evaluating $|\{x \in \mathcal{U} \mid \mathrm{eid}(x) = \mathrm{id}(S)\}|$ after the algorithm has terminated. For DFG, the contribution of a set is given by the number of as-yet-uncovered elements it contains at the point which it is added to the solution. By direct comparison of Figure 3a and Figure 3b, which show the distribution of contributions for solutions given by DFG and the Emek-Rosén algorithm respectively, we observe that the Emek-Rosén cover contains more sets which contribute only one or two elements.

Algorithm 3: Multi-Pass Emek-Rosén

```
input  : A set stream $\mathcal{S}$, a pass count $p$, and a
         threshold value $r_j$ for each $1 \le j \le p$
output: A cover certificate and an
         effectiveness map
1  $\forall x \in \mathcal{U}$: $\mathrm{eid}(x) \leftarrow$ NULL; $\mathrm{eff}(x) \leftarrow -1$
2  for $j \leftarrow 1$ to $p$ do
       /* Do an Emek-Rosén pass */
3      for $t \leftarrow 0$ to $m-1$ do
4          Read the set $S_t$ from $\mathcal{S}$
5          $S'_t \leftarrow S_t \cap \mathcal{U}$
6          $T \leftarrow T' \subseteq_{\mathrm{eff}_t} S'_t$ which maximises $|T'|$
7          foreach $x \in T$ do
8              $\mathrm{eid}(x) \leftarrow \mathrm{id}(S_t)$
9              $\mathrm{eff}(x) \leftarrow \mathrm{lev}(T)$
       /* Restrict the universe */
10     foreach $x \in \mathcal{U}$ do
11         if $j = p$ or $\mathrm{eff}(x) > r_j$ then
12             $\mathcal{U} \leftarrow \mathcal{U} \setminus \{x\}$
13         else
14             $\mathrm{eff}(x) \leftarrow -1$
15 return $\mathrm{eid}(\cdot)$ and $\mathrm{eff}(\cdot)$
```

To better cover these elements at the lower end of the contribution distribution, we present in Algorithm 3 Multi-Pass Emek-Rosén, a generalisation of the original algorithm which makes multiple passes over the input stream. On the first pass, normal Emek-Rosén is applied to the set stream. Then, the universe is restricted to a subset of the original, such that all elements in this restricted universe have an effectiveness of at most some threshold value r_1. Only those sets covering the elements not in this restricted universe remain in the cover certificate. The Emek-Rosén algorithm is then applied on the second pass, this time omitting all elements not in the restricted universe, after which the universe is again restricted with the slightly lower threshold r_2, and so this recursive approach continues for a total of p passes.

We note that a solution set can be obtained during execution without post-processing the cover certificate: if we aggregate the identifiers of those sets responsible for covering the elements excluded from the universe on line 12, this is equivalent to the resulting cover.

6.1 Approximation Factor

We now establish the approximation factor of Multi-Pass Emek-Rosén.

Theorem 6.1. *The cardinality of the solution produced by p-pass* Emek-Rosén *is within a factor* $8(p+1)\Delta^{1/(p+1)}$ *of* $|\mathsf{Opt}|$.

Proof. We prove this by induction.

Induction hypothesis. For some $p \geq 1$, p-pass Emek-Rosén produces a solution within a factor α_p of $|\mathsf{Opt}|$, where $\alpha_p = (p+1)8^{p/(p+1)}\Delta^{1/(p+1)} - p$.

Base case. Let us first consider the base case $p = 1$. The predicted approximation factor α_1 is given by

$$\alpha_1 = (1+1)8^{1/(1+1)}\Delta^{1/(1+1)} - 1 = 4\sqrt{2}\sqrt{\Delta} - 1,$$

which matches the approximation factor given in Theorem 4.1.

Induction step. Assume the induction hypothesis is true for $p = k$, i.e.,

$$\alpha_k = (k+1)8^{k/(k+1)}\Delta^{1/(k+1)} - k.$$

We now show that the induction hypothesis holds for $p = k+1$.

Consider some threshold $r \in \mathbb{R}$. In the first of our $k+1$ passes, we keep only those sets in $S(>r)$ in our final solution $\mathrm{Im}(\chi)$, leaving the subproblem of finding a cover for the remaining subuniverse $I(\leq r)$. As we have k passes remaining, we now apply k-pass Emek-Rosén to this subproblem, which we know under our assumption produces a solution of size less than

$$\left((k+1)8^{k/(k+1)}{\Delta_2}^{1/(k+1)} - k\right) \cdot |\mathsf{Opt}_2|,$$

where Δ_2 corresponds to the maximum number of elements in $I(\leq r)$ contained in a single set and Opt_2 is the optimal covering of $I(\leq r)$ which satisfies $|\mathsf{Opt}_2| \leq |\mathsf{Opt}|$. We know that every set contains fewer than 2^{r+1} elements of $I(\leq r)$, therefore $\Delta_2 < 2^{r+1}$. Considering this alongside Lemma 4.2, it follows that

$$|\mathrm{Im}(\chi)| < \left((k+1)8^{k/(k+1)}\left(2^{r+1}\right)^{1/(k+1)} - k + \frac{\Delta}{2^{r-2}} - 1\right) \cdot |\mathsf{Opt}|.$$

This approximation factor is minimised, i.e.,

$$\begin{aligned}\frac{\partial}{\partial r}&\left((k+1)8^{k/(k+1)}\left(2^{r+1}\right)^{1/(k+1)} - k + \frac{\Delta}{2^{r-2}} - 1\right)\\ &= \left(8^{k/(k+1)}\left(2^{r+1}\right)^{1/(k+1)} - \frac{\Delta}{2^{r-2}}\right)\ln 2\\ &= 0,\end{aligned}$$

when the threshold r is set to

$$r = \frac{(k+1)\log_2\Delta + 1 - k}{k+2}.$$

With this threshold, and after some simplification,

$$\begin{aligned}|\mathrm{Im}(\chi)| &< \left((k+2)8^{(k+1)/(k+2)}\Delta^{1/(k+2)} - k - 1\right) \cdot |\mathsf{Opt}|\\ &= \alpha_{k+1} \cdot |\mathsf{Opt}|,\end{aligned}$$

thus showing that α_{k+1} indeed holds.

Conclusion. We have shown that p-pass Emek-Rosén produces a solution that satisfies

$$\begin{aligned}|\mathrm{Im}(\chi)| &< \left((p+1)8^{p/(p+1)}\Delta^{1/(p+1)} - p\right) \cdot |\mathsf{Opt}|\\ &< \left(8(p+1)\Delta^{1/(p+1)}\right) \cdot |\mathsf{Opt}|,\end{aligned}$$

which completes the proof. ∎

Naturally, this approximation factor relies on selecting the appropriate threshold value r_j for the j^{th} pass. Though we omit the proof here for the sake of brevity, we find by induction that the approximation factor given in Theorem 6.1 is achieved when

$$r_j = ((p-j+1)\log_2\Delta + 3j - p - 1)/(p+1).$$

Chakrabarti and Wirth [4] show that a semi-streaming algorithm cannot achieve an approximation factor better than $0.99n^{1/(p+1)}/(p+1)^2$, thus Multi-Pass Emek-Rosén is essentially tight up to a factor of $8(p+1)^3$. They also introduce Progressive Greedy, a multi-pass semi-streaming Greedy-like algorithm for Set Cover. During the j^{th} of p passes, Progressive Greedy adds to the solution all sets whose uncovered element count is at least some threshold τ_j. The algorithm also *folds* the final two passes into one by noting sets with a non-zero contribution during the would-be penultimate pass and merging these into the solution as a post-processing step. With $\tau_j = n^{1-j/(p+1)}$, they show that Progressive Greedy is a $(p+1)n^{1/(p+1)}$-approximation algorithm for Set Cover; we note that this bound also holds for Δ when $\tau_j = \Delta^{1-j/(p+1)}$, i.e., solutions given by Progressive Greedy are at most a factor $(p+1)\Delta^{1/(p+1)}$ larger than $|\mathsf{Opt}|$ with this threshold. Though this is favourable to the result of our analysis of Multi-Pass Emek-Rosén by a constant factor, the performance of the two algorithms asymptotically match.

6.2 Multi-Pass Emek-Rosén in Practice

To show that multiple passes improve resulting covers, and for comparison with a Greedy-like algorithm which operates under the same computational model, we prepared C++ implementations of both Multi-Pass Emek-Rosén and Progressive Greedy. We obtained better results when using $r_j = (p-j)\lceil\log_2\Delta\rceil/p$ (rather than the theoretical threshold given in Section 6.1) for Multi-Pass Emek-Rosén and $\tau_j = \Delta^{1-j/(p+1)}$ (rather than $\tau_j = n^{1-j/(p+1)}$) for Progressive Greedy, so we assume these thresholds hereafter.

We ran both algorithms on each of the data sets given in Table 1 for each pass count from 1 to 16 using the same system described in Section 5.2. The

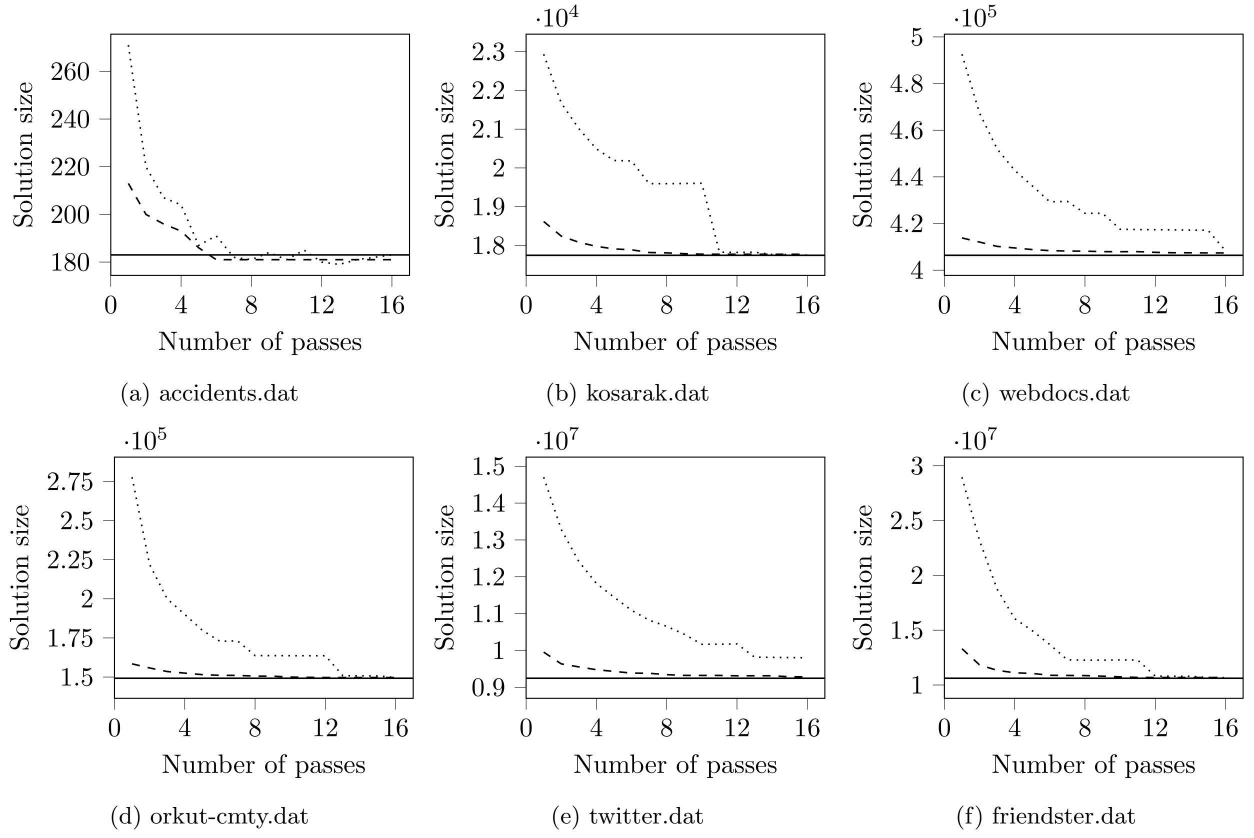


Figure 4: Solution quality comparison between Multi-Pass Emek-Rosén (dashed) and Progressive Greedy (dotted), with DFG ($p = 1.065$, solid) for reference.

memory usage and (I/O-dominated) execution times were very similar between the two algorithms for each run; the main difference was solution quality. In Figure 4, we compare for each data set the cover sizes given by Multi-Pass Emek-Rosén and Progressive Greedy, and show the solution size obtained by DFG for reference. In each case, the solution sizes all tend towards the result given by DFG, with the Multi-Pass Emek-Rosén solutions typically converging in fewer passes. We note also an inherent weakness of Multi-Pass Emek-Rosén: for a given data set, there are at most $\lceil \log_2 \Delta \rceil + 2$ possible effectiveness values (levels), therefore if the universe is partitioned by each of these levels, the performance of the algorithm cannot improve any further. With the effectiveness threshold as formulated in our implementation, Multi-Pass Emek-Rosén reaches a steady final solution size after $\lceil \log_2 \Delta \rceil + 1$ passes, and any further passes are superfluous. Figure 3c shows the contribution distribution of a cover given by Multi-Pass Emek-Rosén after $\lceil \log_2(51) \rceil + 1 = 7$ passes. By comparison to Figure 3b, we see that there are fewer sets covering only one or two elements; the distribution better matches that given by DFG in Figure 3a.

7 Conclusions and Future Work

In this work, we have demonstrated that a streaming approach to the Set Cover problem works well in practice. We empirically compared the semi-streaming Emek-Rosén algorithm to the state-of-the-art Disk Friendly Greedy algorithm. We found that, at the cost of slightly larger cover sizes, the Emek-Rosén algorithm was able to approximate instances of the Set Cover problem faster and using significantly less space than Disk Friendly Greedy, giving a strongly positive answer to Emek and Rosén's remark in their paper [10] that the algorithm may be useful in practice.

In the last decade, a tremendous number of new data streaming algorithms with provable guarantees have been designed for large scale problems. Most of these algorithms have never been implemented, which we believe is a missed opportunity. We therefore strongly advocate further research into the applicability of recent data streaming algorithms, thereby bridging the gap between theory and practice.

References

[1] Aris Anagnostopoulos, Luca Becchetti, Ilaria Bordino, Stefano Leonardi, Ida Mele, and Piotr Sankowski. Stochastic query covering for fast approximate document retrieval. *ACM Transactions on Information Systems (TOIS)*, 33(3):1–35, 2015.

[2] Sepehr Assadi. Tight space-approximation tradeoff for the multi-pass streaming set cover problem. In Emanuel Sallinger, Jan Van den Bussche, and Floris Geerts, editors, *Proceedings of the 36th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, PODS 2017, Chicago, IL, USA, May 14-19, 2017*, pages 321–335. ACM, 2017.

[3] Sepehr Assadi, Sanjeev Khanna, and Yang Li. Tight bounds for single-pass streaming complexity of the set cover problem. In Daniel Wichs and Yishay Mansour, editors, *Proceedings of the 48th Annual ACM SIGACT Symposium on Theory of Computing, STOC 2016, Cambridge, MA, USA, June 18-21, 2016*, pages 698–711. ACM, 2016.

[4] Amit Chakrabarti and Anthony Wirth. Incidence geometries and the pass complexity of semi-streaming set cover. In *Proceedings of the twenty-seventh annual ACM-SIAM symposium on Discrete algorithms*, pages 1365–1373. SIAM, 2016.

[5] Raffaele Perego Claudio Lucchese, Salvatore Orlando and Fabrizio Silvestri. Webdocs: a real-life huge transactional dataset.

[6] Graham Cormode, Howard Karloff, and Anthony Wirth. Set cover algorithms for very large datasets. In *Proceedings of the 19th ACM international conference on Information and knowledge management*, pages 479–488, 2010.

[7] Erik D. Demaine, Piotr Indyk, Sepideh Mahabadi, and Ali Vakilian. On streaming and communication complexity of the set cover problem. In Fabian Kuhn, editor, *Distributed Computing - 28th International Symposium, DISC 2014, Austin, TX, USA, October 12-15, 2014. Proceedings*, volume 8784 of *Lecture Notes in Computer Science*, pages 484–498. Springer, 2014.

[8] Irit Dinur and David Steurer. Analytical approach to parallel repetition. In *Proceedings of the forty-sixth annual ACM symposium on Theory of computing*, pages 624–633, 2014.

[9] Yuval Emek and Adi Rosén. Semi-streaming set cover - (extended abstract). In Javier Esparza, Pierre Fraigniaud, Thore Husfeldt, and Elias Koutsoupias, editors, *Automata, Languages, and Programming - 41st International Colloquium, ICALP 2014, Copenhagen, Denmark, July 8-11, 2014, Proceedings, Part I*, volume 8572 of *Lecture Notes in Computer Science*, pages 453–464. Springer, 2014.

[10] Yuval Emek and Adi Rosén. Semi-streaming set cover. *ACM Trans. Algorithms*, 13(1), November 2016.

[11] Uriel Feige. A threshold of ln n for approximating set cover. *Journal of the ACM (JACM)*, 45(4):634–652, 1998.

[12] Joan Feigenbaum, Sampath Kannan, Andrew McGregor, Siddharth Suri, and Jian Zhang. On graph problems in a semi-streaming model. *Departmental Papers (CIS)*, page 236, 2005.

[13] Karolien Geurts, Geert Wets, Tom Brijs, and Koen Vanhoof. Profiling high frequency accident locations using association rules. In *Proceedings of the 82nd Annual Transportation Research Board, Washington DC. (USA), January 12-16*, page 18pp, 2003.

[14] Bart Goethals. Frequent itemset mining dataset repository.

[15] Fernando C Gomes, Cláudio N Meneses, Panos M Pardalos, and Gerardo Valdisio R Viana. Experimental analysis of approximation algorithms for the vertex cover and set covering problems. *Computers & Operations Research*, 33(12):3520–3534, 2006.

[16] Tal Grossman and Avishai Wool. Computational experience with approximation algorithms for the set covering problem. *European Journal of Operational Research*, 101(1):81–92, 1997.

[17] Sariel Har-Peled, Piotr Indyk, Sepideh Mahabadi, and Ali Vakilian. Towards tight bounds for the streaming set cover problem. In Tova Milo and Wang-Chiew Tan, editors, *Proceedings of the 35th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, PODS 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, pages 371–383. ACM, 2016.

[18] Chengbang Huang, Faruck Morcos, Simon P Kanaan, Stefan Wuchty, Danny Z Chen, and Jesus A Izaguirre. Predicting protein-protein interactions from protein domains using a set cover approach. *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, 4(1):78–87, 2007.

[19] Piotr Indyk, Sepideh Mahabadi, Ronitt Rubinfeld, Jonathan Ullman, Ali Vakilian, and Anak Yodpinyanee. Fractional set cover in the streaming model. In Klaus Jansen, José D. P. Rolim, David Williamson, and Santosh S. Vempala, editors, *Approximation, Randomization, and Combinatorial Optimization. Algorithms and Techniques, APPROX/RANDOM 2017, August 16-18, 2017, Berkeley, CA, USA*, volume 81 of *LIPIcs*, pages 12:1–12:20. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2017.

[20] Richard M Karp. Reducibility among combinatorial problems. In *Complexity of computer computations*, pages 85–103. Springer, 1972.

[21] Haewoon Kwak, Changhyun Lee, Hosung Park, and Sue Moon. What is twitter, a social network or a news media? In *Proceedings of the 19th international conference on World wide web*, pages 591–600, 2010.

[22] Jure Leskovec and Andrej Krevl. SNAP Datasets: Stanford large network dataset collection. `http://snap.stanford.edu/data`, June 2014.

[23] Ching Lih Lim, Alistair Moffat, and Anthony Wirth. Lazy and eager approaches for the set cover problem. In *Proceedings of the Thirty-Seventh Australasian Computer Science Conference-Volume 147*, pages 19–27, 2014.

[24] Alan Mislove, Massimiliano Marcon, Krishna P. Gummadi, Peter Druschel, and Bobby Bhattacharjee. Measurement and Analysis of Online Social Networks. In *IMC*, San Diego, CA, October 2007.

[25] Petr Slavík. A tight analysis of the greedy algorithm for set cover. In *Proceedings of the twenty-eighth annual ACM symposium on Theory of computing*, pages 435–441, 1996.

[26] Stergios Stergiou and Kostas Tsioutsiouliklis. Set cover at web scale. In *Proceedings of the 21th acm sigkdd international conference on knowledge discovery and data mining*, pages 1125–1133, 2015.

[27] Jaewon Yang and Jure Leskovec. Defining and evaluating network communities based on ground-truth. *CoRR*, abs/1205.6233, 2012.

[28] Shin Yoo. A novel mask-coding representation for set cover problems with applications in test suite minimisation. In *2nd International Symposium on Search Based Software Engineering*, pages 19–28. IEEE, 2010.

A Proof of Lemma 4.2

LEMMA A.1. *For some $r \in \mathbb{Z}$, $|S(r)| < n/2^{r-1}$.*

Proof. See [10]. □

LEMMA 4.2. *For some $r \in \mathbb{R}$,*

$$|S(>r)| < \left(\frac{\Delta}{2^{r-2}} - 1\right) \cdot |\mathsf{Opt}|.$$

Proof. Consider Lemma A.1, which is given by Emek and Rosén in their original analysis of the algorithm. Summing this bound over the integer interval $[\lfloor r \rfloor + 1, \lceil \log_2 \Delta \rceil]$ results in the converging series

$$\begin{aligned} \sum_{r'=\lfloor r \rfloor+1}^{\lceil \log_2 \Delta \rceil} \frac{n}{2^{r'-1}} &= n\left(\frac{1}{2^{\lfloor r \rfloor -1}} - \frac{1}{2^{\lceil \log_2 \Delta \rceil -1}}\right) \\ &< n\left(\frac{1}{2^{r-2}} - \frac{1}{\Delta}\right) \\ &\leq \left(\frac{\Delta}{2^{r-2}} - 1\right) \cdot |\mathsf{Opt}|, \end{aligned}$$

where the final inequality follows from the observation that $|\mathsf{Opt}| \geq n/\Delta$. □