Set Coverage Problems in a One-Pass Data Stream

Huiwen Yu* Dayu Yuan†

Abstract

Finding a maximum coverage by k sets from a given collection (Max-k-Cover), finding a minimum number of sets with a required coverage (Partial-Cover) are both important combinatorial optimization problems. Various problems from data mining, machine learning, social network analysis, operational research, etc. can be generalized as a set coverage problem. The standard greedy algorithm is efficient as an in-memory algorithm. However, when we are facing very large-scale dataset or in an online environment, we seek a new algorithm which makes only one pass through the entire dataset.

Previous one-pass algorithms for the Max-k-Cover problem cannot be extended to the Partial-Cover problem and do not enjoy the prefix-optimal property. In this paper, we propose a novel one-pass streaming algorithm which produces a prefix-optimal ordering of sets, which can easily be used to solve the Max-k-Cover and the Partial-Cover problems. Our algorithm consumes space linear to the size of the universe of elements. The processing time for a set is linear to the size of this set. We also show with the aid of computer simulation that the approximation ratio of the Max-k-Cover problem is around 0.3. We conduct experiments on extensive datasets to compare our algorithm with existing one-pass algorithms on the Max-k-Cover problem, and with the standard greedy algorithm on the Partial-Cover problem. We demonstrate the efficiency and quality of our algorithm.

Keyword: max-k-cover problem; one-pass stream; partial-cover problem

1 Introduction

Given a universe of elements U, a collection of subsets $\mathcal{S}$ of U, the *Max-k-Cover problem* asks for selecting at most k sets from $\mathcal{S}$, such that the weight of the total coverage by those k sets is maximized. Many applications can be reduced to the Max-k-Cover problem. In the influence maximization problem [18] used in marketing strategy design, we want to target a bounded number of nodes in a social network which have a large influence on the rest of the network. In the sensor placement problem [20] for water pollution monitoring, sensors are expensive and therefore only a limited number is installed in a way that the area under detection is maximized. In the blog monitoring problem [24], we want to know how to choose a small number of interesting blogs which cover wide topics. A closely related problem is the Set-Cover problem, which asks for finding the minimum number of sets which cover the whole universe. However, Set-Cover is sometimes not a practical model, as it can be costly to cover the whole universe. In most cases, we only require a sufficiently large fraction of the universe to be covered. Therefore, we consider a variant of the Set-Cover problem, *Partial-Cover*, which for a given c, finds a minimum number of sets which have a coverage of at least c. Dictionary learning [7, 15], feature selection in frequent graph mining [25], and feature selection in text mining [11] are examples where we need to select a small number of informative features which can approximately represent the whole dataset, and they can be modeled as the Partial-Cover problem.

The Max-k-Cover and the Partial-Cover problems are NP-hard. They can be solved approximately by a simple greedy approach which iteratively picks a set maximizing the increase of coverage. For datasets of small or medium scale, this greedy approach can easily be implemented using an inverted index. However, there are two scenarios where this implementation fails to perform well. First, if data is too large to reside on main memory. For example, selecting representative subgraph features from exponentially-large number of candidates to build index [27]. In this case, random access is not practical, and it can be expensive to setup an inverted index. Therefore, we need to read through the whole dataset on disk sequentially in each iteration. The running time partly depends on the frequency an algorithm accesses the external storage. Cormode et al. [9] give a more efficient algorithm dealing with disk-reside data. The algorithm improves the number of passes through external storage, but is still not scalable when a dataset is very large. Secondly, if data is given in an online fashion and we are required to provide a high-quality solution efficiently once new data is available [5]. In this situation, we cannot retrieve all data in previous time. Instead, we have to keep a solution and make a fast update whenever there is new data. Blog monitoring [24] is an example of the online Max-k-Cover problem. To overcome the difficulties raised in these two scenarios, in this paper, we study algorithms which consume small memory and make only one-pass through the entire dataset.

The one-pass space-constraint Max-k-Cover problem has been studied in [1, 24]. In both papers, the algorithm keeps k sets in main memory as the current k-cover solution. For each new set, the algorithm decides to keep it by replacing a set in the cover or discard it. We call it *swapping-based* one-pass streaming algorithm (SOPS). However, there

*Department of Computer Science and Engineering, The Pennsylvania State University, USA. hwyu@cse.psu.edu. Research supported in part by NSF Grant CCF-0964655.

†Department of Computer Science and Engineering, The Pennsylvania State University, USA. duy113@cse.psu.edu.

are two major drawbacks of swapping-based algorithms. First, the algorithms produce a solution for a pre-determined k and does not provide a ranking for sets in the cover. This can fail to solve some applications. For example, query diversity can be modeled as a coverage problem [26]. A search engine returns an ordered list of ranking results. It only loads more contents if a user clicks to read results in the next page. In this example, the algorithm should be able to give cover by an arbitrary number of sets which are ordered. Secondly, the swapping-based algorithm cannot be extended to the Partial-Cover problem, as we will show in Section 3.2.

Our main contribution of this paper is a novel one-pass streaming algorithm. We call our algorithm *greedy-based* one-pass streaming algorithm (GOPS). GOPS aims to output a prefix-optimal list of sets, such that for any k, the union of the first k sets forms a large k-cover. The algorithm can be adapted to the Partial-Cover problem in a natural way. GOPS needs $O(|U|)$ space, which is the minimum requirement for solving the Partial-Cover problem with $c = \Theta(|U|)$. It takes time $O(|S|)$ to process a new set S. We analyze the worst case of GOPS with the aid of computer simulation and find that it has an approximation ratio around 0.3. We compare our algorithm with swapping-based algorithms [1, 24] via extensive real-world and synthetic data. Experimental results demonstrate that our algorithm is much faster and more accurate than swapping-based algorithms in the Max-k-Cover problem. It has a performance close to the standard greedy algorithm on the Partial-Cover problem.

The outline of the paper is as follows. We review the standard greedy approach and related works in Section 2. We describe the swapping-based one-pass algorithms and discuss the difficulty of extending them to other set coverage problems in Section 3. We then present and analyze our greedy-based one-pass algorithm in Section 4. In Section 5, we show experimental results. Finally, we conclude in Section 6.

2 Preliminaries and Related Works

In this paper, we consider the Max-k-Cover problem and the Partial-Cover problem for the uniform-weighted case. Namely, the weight of a set is the size of the set. The discussion can be extended to the weighted case, where we have a weight function $w : U \to \mathbb{R}^+$ and the weight of a set S is the total weight of the elements in S. We always denote the weight of a set S by $w(S)$.

The greedy approach to both problems selects a set S which covers the maximum weight of uncovered elements in each iteration, until k sets are selected for the Max-k-Cover problem or the total weight of coverage exceeds c for the Partial-Cover problem. This greedy algorithm has an approximation ratio $1 - \frac{1}{e} \approx 0.632$ for the Max-k-Cover problem and $1 + \ln c$ for the Partial-Cover problem [17].

For the Max-k-Cover problem used in many applications, we desire not only a large coverage, but also a ranking of sets, which indicates the contribution of each set to the cover. We introduce the concept of prefix-optimality to define such an ordering.

DEFINITION 2.1. (α-PREFIX-OPTIMAL) *We say an ordered collection of k sets $S_1, S_2, ..., S_k$ α-prefix-optimal for the Max-k-Cover problem, if for any $1 \leq k' \leq k$, $S_1, ..., S_{k'}$ is an α-approximation to the Max-k'-Cover problem.*

As an example, the greedy algorithm outputs an ordered list of sets which is $(1 - \frac{1}{e})$-prefix-optimal for the Max-k-Cover problem. The prefix-optimal ordering offers the flexibility to choose an arbitrary number of sets with a large coverage. Once we have an α-prefix-optimal ordering, we can approximate the Partial-Cover problem by picking sets in order until the weight of their union exceeds the pre-determined threshold c. This simple algorithm has an approximation ratio $1 + \log_{\frac{1}{1-\alpha}} c$ (proof similar to the one in [17]).

Notations. In the following, small letters represent elements, capital letters represent sets of elements, and calligraphic letters represent collections of sets. Specifically, we always use $\mathcal{O}$ to represent a collection of optimal sets, and OPT for the value of the optimal solution. We denote the number of sets in a collection $\mathcal{S}$ by $n_{\mathcal{S}}$.

2.1 Related works

We can solve the Max-k-Cover problem by a greedy algorithm [17]. On the other hand, a hardness result [10] indicates that we cannot do better than the greedy approach in polynomial time assuming $P \neq NP$. Nemhauser et al. [23] present the first swapping-based algorithm for the Max-k-Cover problem. The algorithm makes multiple passes over the entire dataset and has a $1/2$ approximation ratio. Gomes and Krause [15] apply this idea and show empirically that this method converges in not too many rounds under reasonable assumptions. Saha and Getoor [24] give the first one-pass swapping-based algorithm. Their algorithm has a $1/4$ approximation ratio. Ausiello et al. [1] later improve the ratio for small k. They also show that $1/2$ is a lower bound for any deterministic or randomized algorithm with the constraint that only k sets can be stored. Another line of works on finding large-scale maximum k-covers focus on designing efficient parallel algorithms [3, 8].

The Set-Cover problem can be solved by the same greedy approach. It has an approximation ratio $O(\ln n)$, which is optimal assuming $NP \nsubseteq DTIME(n^{O(\log \log n)})$ [10], where n is the size of the universe. Efficient solutions for the Set-Cover problem have been studied for a long time. [14, 16] demonstrate that the greedy approach works very well on real instances. Cormode et al. [9] modify the greedy algorithm by applying the bucketing idea for disk-resident dataset. [2–4] propose efficient greedy parallel algorithms for the Set-Cover problem.

The Partial-Cover problem is a generalization of the Set-Cover problem which can also be solved by the same greedy approach. Gandhi et al. [12] present a primal-dual approach to the Partial-Cover problem where every element has a bounded frequency. Könemann et al. [19] further generalize the Partial-Cover problem by assigning different costs to sets.

3 Swapping-based One-pass Algorithm

In this section, we first review the structure of swapping-based one-pass algorithms [1, 24], and give a time and space complexity analysis (which is missing in the original papers). We then give a simple example to illustrate the difficulty of extending the algorithm to provide a prefix-optimal ordering, or to the Partial-Cover problem. In the following context, we denote SOPS in [24] by SOPS1 and in [1] by SOPS2.

3.1 Swapping-based algorithms The swapping-based algorithm always keeps k sets $S_1, ..., S_k$ as the current solution to the Max-k-Cover problem. The algorithm starts by picking the first k sets. To process a new set S, it computes a specific score function for S and the k sets in the current cover. The score function depends on C and S. The $score$ of a set S with respect to cover C is usually a function of the weight of the elements exclusively covered by S. In SOPS1, $score(S) = w(S \setminus C)$, $score(S_i) = w(S_i \setminus (C \bigcup S \setminus S_i))$. In SOPS2, $score(S) = w(S \setminus C)$, $score(S_i) = w(S_i \setminus C)$. The algorithm replaces an S_j of the smallest score with S if $score(S) > 2 * score(S_j)$ (SOPS1), or $score(S) + w(C \setminus S_j) > (1 + \frac{1}{k})w(C)$ (SOPS2). Both algorithms achieve an approximation ratio at least $1/4$, with SOPS2 slightly better for small k. However, as we observe in experiments, the solution quality of SOPS2 is much worse than SOPS1.

Complexity. The space complexity for both algorithms is $O(\sum_{i=1}^{k} |S_i|)$. The time complexity is quite different. In SOPS1, $score(S_i)$ depends on both C and S, which has to be recomputed each time the algorithm processes a new set. The time for processing S is $O(|S| + \sum_{i=1}^{k} |S_i|)$. In SOPS2, $score(S_i)$ depends only on C. The algorithm recomputes scores of $S_1, ..., S_k$ only after S replaces some S_i. Otherwise, the time for processing S is $O(|S|)$. Hence, SOPS2 is usually much faster than SOPS1, which is demonstrated in experiments.

3.2 Difficulty of Extensions We give a simple example to show that the swapping-based one-pass algorithm fails to produce a non-trivial prefix-optimal ordering, or solution to the Partial-Cover problem. Suppose we have a stream of sets $S_1, S_2, ..., S_k, S_{k+1}$, where $S_1, ..., S_k$ are pair-wise disjoint and of size 1, $S_{k+1} = \bigcup_{i=1}^{k/2} S_i$. Both swapping-based algorithms pick $S_1, ..., S_{k'}$ as solution to the Max-k'-Cover problem, for $k/2 \leq k' \leq k$. However, we need to put S_{k+1} in the first place to form a nontrivial prefix-optimal ordering. Using the same example for the Partial-Cover problem for $c = 1 + k/2$, if the swapping-based algorithm decides to keep c sets, then it outputs c as a solution while the optimal solution is 2.

Motivated by this example, we observe that the swapping-based algorithms consider only the marginal increase of a set to the current cover, while the quality of the cover depends also on the weight of the set. We utilize this information to design a new one-pass streaming algorithm.

4 Greedy-based One-pass Algorithm

In this section, we introduce a novel one-pass streaming algorithm, which is based on the standard greedy approach, but uses a restricted memory and makes one pass through the entire data. Our algorithm outputs a prefix-optimal ordering of sets, and thus can easily solve the Max-k-Cover and the Partial-Cover problems. We first give the details of our algorithm with complexity analysis. We then compare our algorithm to the swapping-based algorithms. Finally, we present an analysis of the performance guarantee.

4.1 Algorithm Contrary to previous approaches [1, 23, 24], we do not restrict the sets we can keep to be exactly k. Rather, we assume that we can hold $O(|U|)$ elements in memory. This assumption basically says our memory can keep the universe of elements, and thus is large enough for solving the Max-k-Cover or the Partial-Cover problems with any k, c. We denote the collection of sets in memory by $\mathcal{C}$. During the execution of the algorithm, we might modify a set by deleting some elements it contains. In the following context, when we talk about the weight of a set in $\mathcal{C}$, we refer to its weight after possible element deletions.

We want to mimic the standard greedy process, where we always choose the set which maximizes the increase of cover in each iteration, but in an online fashion. To adapt to this change, we compare every new set S to the sets in $\mathcal{C}$ which have weight at least a $\frac{1}{\beta}$ fraction of $w(S)$, for some parameter $\beta > 1$. We denote the collection of these sets by $\mathcal{U}_S$ and the rest by $\mathcal{L}_S$. We delete elements in S which are also contained in $\mathcal{U}_S$. In other words, we count elements covered *uniquely* by $S \setminus \mathcal{U}_S$, rather than $S \setminus \mathcal{C}$ as in swapping-based algorithms. If S remains at least $w(S)/\beta$ elements, we insert S to $\mathcal{C}$. Otherwise, as the weight of S decreases, we update $\mathcal{U}_S$ and keeps on comparing S with the new $\mathcal{U}_S$, until either S is inserted, or S is empty and discarded. If S is inserted, we say S is *ordered* before sets in $\mathcal{L}_S$, and we also update $\mathcal{L}_S$ by deleting all elements in $\mathcal{L}_S$ which belong to S. We call this whole procedure $Sift(S, \mathcal{C}, \beta)$. To efficiently check if an element in S belongs to $\mathcal{U}_S$ and to fast locate sets in $\mathcal{L}_S$ which intersect S, we can use a map to keep the set that every element in $\mathcal{C}$ belongs to. In this way, procedure

$Sift$ runs in time linear to the size of S.

We give an example on how *Sift* is executed. Suppose we have $S_1 = \{a, b, c, d, e, f\}$, $S_2 = \{g, h, i, j, k\}$, $S_3 = \{l, m, n, o\}$, $S_4 = \{p, q, r\}$, $S_5 = \{s, t\}$, $S_6 = \{u\}$ in $\mathcal{C}$ and set $\beta = 1.1$. Now we have a new set $S = \{a, l, s, u, v\}$. In this case, $\mathcal{U}_S = \{S_1, S_2\}$. S is updated by $S \setminus \mathcal{U}_S = \{l, s, u, v\}$. Then S_3 is included into $\mathcal{U}_S$ and S is further updated to $\{s, u, v\}$. S_4 is then included into $\mathcal{U}_S$ but this time S stays unchanged and we insert S into $\mathcal{C}$. $\mathcal{L}_S = \{S_5, S_6\}$. S_5 is updated by $S_5 \setminus S = \{t\}$. S_6 becomes empty and is deleted from $\mathcal{C}$.

The whole algorithm executes by calling the *Sift* procedure to every set in a stream. We show later that the output of this algorithm is an α_β-prefix optimal ordering of sets in $\mathcal{C}$, for a constant α_β depending on the parameter β. To obtain a solution to the Max-k-Cover problem, we pick the largest k sets from $\mathcal{C}$. For the Partial-Cover problem, we pick the largest p sets, for the smallest p such that their total weight exceeds c. We give a description of this algorithm GOPS in Algorithm 1.

Algorithm 1 Greedy-based one-pass streaming algorithm

```
1:  C ⟵ ∅.
2:  for each new set S do
3:      Sift(S, C, β).
4:  end for
5:  return C.
6:  procedure SIFT(S, C, β)
7:      ▷ Insert S.
8:      while S is not inserted to C and S is not empty do
9:          S' ⟵ S.
10:         U_S ⟵ {T ∈ C | w(T) ≥ w(S)/β}.
11:         S ⟵ S \ U_S.
12:         if w(S) ≥ w(S')/β then
13:             Insert S to C.
14:         end if
15:     end while
16:     ▷ Update L_S.
17:     if S is not empty then
18:         L_S ⟵ C \ U_S.
19:         for T ∈ L_S and T ⋂ S ≠ ∅ do
20:             T ⟵ T \ S.
21:             Delete T from C if it becomes empty.
22:         end for
23:     end if
24: end procedure
```

Complexity. The space complexity of Algorithm 1 is $O(|U|)$, as sets in $\mathcal{C}$ are disjoint. We compute the time the algorithm spends on processing a set S by summing up the time for sifting S and possibly later updating $\mathcal{L}_S$. Notice that after calling $Sift$ on S, the size of S shrinks by a factor of at least $1 - \frac{1}{\beta}$. Moreover, the number of sets being updated in $\mathcal{L}_S$ is at most the size of S when S is inserted to $\mathcal{C}$. Therefore, the total time is at most $c_o \cdot (1 + \frac{1}{\beta} + \frac{1}{\beta^2} + \cdots) \cdot |S|$, which is $c_o \cdot (1 + \frac{1}{\beta - 1}) \cdot |S|$, for some constant c_o, i.e. $O(|S|)$.

4.2 Comparison to swapping-based algorithms We compare the two types of one-pass streaming algorithms, SOPSs [1, 24] with GOPS from an algorithm design point of view. We give more comparisons based on experimental performance in Section 5. SOPS has the advantages of being nonparametric, and it keeps the original data during execution. However, unlike GOPS, it is less flexible as there is no easy way to extend SOPS to produce a prefix-optimal ordering or to solve the Partial-Cover problem.

A typical swapping-based algorithm keeps k sets in memory. Hence, the memory consumption is $O(\sum_{i=1}^{k} |S_i|)$. If k is large, this quantity can be much larger than the space for storing the universe, and may not even be held in memory. The time complexity for processing a set of SOPS1 is $O(|T| + \sum_{i=1}^{k} |S_i|)$. It is much larger than GOPS especially for medium or large k. As for SOPS2, the time for computing $score(S)$ is less than the time GOPS takes to process S. However, if S is kept, SOPS2 will recompute $score(S_i)$, $i = 1, ..., k$, then the single set processing time is higher than GOPS. As we observe in experiments, the average single set processing time of SOPS2 is higher than GOPS.

4.3 Analysis In this section, we show that GOPS outputs an α_β-prefix optimal set ordering by analyzing its approximation ratio to the Max-k-Cover problem for arbitrary k. This implies the approximation ratio of the Partial-Cover problem is $1 + \log_{\frac{1}{1-\alpha_\beta}} c$. Since we cannot access the whole data in advance, the performance guarantee of GOPS can be worse than that of the standard greedy algorithm, $1 - \frac{1}{e}$. We explain the main reason of this performance decay by the following example.

EXAMPLE 4.1. *Suppose in a data stream, we first have a collection $\mathcal{S} = \{S_1, ..., S_n\}$ of disjoint sets of weight $\{w_1, ..., w_n\}$, where $w_1 \geq w_2 \geq \cdots \geq w_n$. The algorithm keeps every set in $\mathcal{S}$. Then we have a collection $\mathcal{T}$, where every set T in $\mathcal{T}$ has weight at least β times the maximum set-weight in $\mathcal{S}$ when the algorithm starts processing T. Moreover, T has an intersection of weight $w_1 - w_2$ with S_1, an intersection of weight $w_2 - w_3$ with S_1, S_2, and generally an intersection of weight $w_i - w_{i+1}$ with $S_1, ..., S_i$, for i satisfying that $\sum_{j=1}^{i} j(w_j - w_{j+1}) \leq \mu$ and $\sum_{j=1}^{i+1} j(w_j - w_{j+1}) > \mu$, and finally an intersection of weight $(\mu - \sum_{j=1}^{i} j(w_j - w_{j+1}))/(i+1)$ with $S_1, .., S_{i+1}$. By the Sift process, after inserting T, S_j decreases by $w_j - w_i + (\mu - \sum_{j=1}^{i} j(w_j - w_{j+1}))/(i+1)$, for $j = 1, ..., i, i+1$.*

In general, we say a set S splits a collection of sets $\mathcal{C}$ if S is ordered before the sets in $\mathcal{C}$ and S has an intersection with

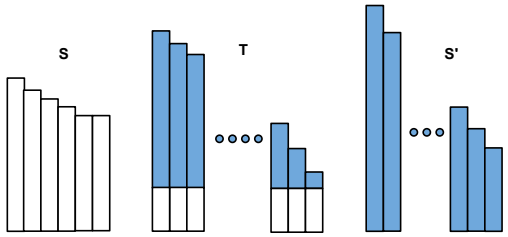

Figure 1: Illustrative figure for Example 4.1. A rectangle represents a set. The area with the same color represents a same set of elements. The stream of sets comes in a left-to-right order. At the end of the event, only $\mathcal{S}'$ and the white part of $\mathcal{T}$ are kept in memory.

every set in $\mathcal{C}$. To simplify description, we sometimes say a collection $\mathcal{C}_1$ splits a collection $\mathcal{C}_2$ when we only require that every set in $\mathcal{C}_1$ splits a subcollection of $\mathcal{C}_2$, not necessarily the whole collection.

Back to the example, the splitting event could repeat many times until the set S_i in $\mathcal{S}$ remains of weight $\nu_i \geq 0$, $i \geq 1$. Assume in this stream, the collection $\mathcal{S}$ becomes empty at some point and we have a collection $\mathcal{T}$. If the stream stops, we can extract a better k-cover from $\mathcal{T}$ than $\mathcal{S}$. However, the splitting events can again happen. Suppose we have a collection $\mathcal{S}'$, where $\mathcal{S}' \bigcap \mathcal{T} = \mathcal{T} \setminus \mathcal{S}$. Every set S' in $\mathcal{S}'$ has weight at least β times the maximum set-weight in $\mathcal{T}$ when the algorithm starts processing S'. The way S' intersects $\mathcal{T}$ is similar to the way T intersects $\mathcal{S}$. The difference from the $\mathcal{T}$ splitting $\mathcal{S}$ scenario is that, S' intersects of weight $w(S')$ with $\mathcal{T}$. Moreover, after we process all sets in $\mathcal{S}'$, every set in $\mathcal{T}$ leaves weight μ, which is the common part of $\mathcal{T}$ and $\mathcal{S}$. Now, the stream ends.

We illustrate the whole event in Figure 1. Observe that $\mathcal{S}$ and $\mathcal{S}'$ are disjoint. After being split by $\mathcal{S}'$, $\mathcal{T}$ covers the same set of elements as $\mathcal{S}$ but using more sets. The optimal solution contains the k largest sets in $\mathcal{S}'$ and $\mathcal{S}$, while GOPS chooses the k largest sets from $\mathcal{S}'$ and $\mathcal{T}$ after being split by $\mathcal{S}'$ (the white part of $\mathcal{T}$). If we consider $\mathcal{S}$ and $\mathcal{S}'$ as collections of sets chosen by a greedy algorithm, GOPS performs strictly worse due to splitting events.

We are now ready to compute the approximation ratio of GOPS. We first summarize how we compute the approximation ratio. Let $\mathcal{A}$ be k sets chosen by GOPS and ALG be the value of this solution. We first construct a collection of auxiliary sets $\mathcal{G}$ from $\mathcal{A}$, such that $\mathcal{G}$ is a greedy solution. Let GR be the weight of coverage by sets in $\mathcal{G}$. For parameter β, we will show that $\frac{GR}{OPT} \geq 1 - e^{-\frac{1}{\beta}} = g_\beta$. We then find a lower bound c_β of $\frac{ALG}{GR}$ with the aid of computer simulation. Hence, the approximation ratio of GOPS, $\alpha_\beta = \frac{ALG}{OPT} = \frac{ALG}{GR} \cdot \frac{GR}{OPT}$ can be lower bounded by $(1 - e^{-\frac{1}{\beta}}) \cdot c_\beta$. For practical reason, we consider β ranging from 1.1 to 1.9. We list the approximation ratio α_β, and g_β, c_β in Table 1. In the following analysis, some proofs and deductions are omitted due to space constraint.

Table 1: Approximation ratio of GOPS with respect to β and parameter settings for searching c_β. $\nu = 0$ for all cases.

β	1.1	1.3	1.5	1.7	1.9
α_β	0.1240	0.2455	0.2965	0.3035	0.3000
g_β	0.5971	0.5366	0.4866	0.4447	0.4092
c_β	0.2076	0.4575	0.4093	0.6824	0.7340
l	7	6	4	3	3
w_1	5.5	3.6	3.5	4.6	8.1
μ	1	1.4	1.8	2.4	4.7

Let $\mathcal{A} = \{S_1, ..., S_k\}$. We say a set S^* is the original set of S if S^* is a set with no element deletion and S is a subset of S^* after S^* being modified. We construct an auxiliary collection $\mathcal{G}$ as follows.

• Construction of $\mathcal{G}$.

1. For any set $S_i \in \mathcal{A}$, let S_i^* be the corresponding original set of S_i. Consider a set of elements E_S which belong to $\{S_i^*\}$ but not $\{S_i\}$. For any element $e \in E_S$, suppose e belongs to every set in the subcollection $\{S_{i_j}^*\}$, add e to the set in $\{S_{i_j}\}$ with the largest weight. Denote the new collection by $\mathcal{A}'$.

2. For any set S in the data stream and $S \notin \mathcal{A}'$, consider all those sets such that $S \setminus \mathcal{A}'$ has at least the minimum set-weight in $\mathcal{A}'$. Add the qualified sets to $\mathcal{A}'$ greedily. (Note that we need to update $\mathcal{A}'$ every time we insert a new set.) Denote the new collection by $\mathcal{G}'$.

3. Pick the first k largest sets from $\mathcal{G}'$ greedily, denote the collection of these k sets by $\mathcal{G}$.

CLAIM 4.1. $\frac{GR}{OPT} \geq 1 - e^{-\frac{1}{\beta}}$.

• Compute lower bound of $\frac{ALG}{GR}$.

We first consider the splitting event in Example 4.1. A collection $\mathcal{S}$ is split by $\mathcal{T}$ and $\mathcal{T}$ is split by $\mathcal{S}'$. This event can happen many times when we process a data stream by GOPS. Namely, we can have a chain of collection, $\mathcal{S}_1, \mathcal{T}_1, ..., \mathcal{S}_{l-1}, \mathcal{T}_{l-1}, \mathcal{S}_l$, where $\mathcal{T}_i$ splits $\mathcal{S}_i$ and is split by $\mathcal{S}_{i+1}$, for $i = 1, 2, ..., l$. The splitting events can be more complicated than a chain of collections $\mathcal{S}_1, \mathcal{T}_1 ..., \mathcal{S}_{l-1}, \mathcal{T}_{l-1}$, one splits another. For example, a collection $\mathcal{S}_1$ might be first split by $\mathcal{T}_1$ into sets of weight $\nu' > \nu$, $\mathcal{T}_1$ is then split by $\mathcal{S}_2$, finally, $\mathcal{S}_1$ and $\mathcal{S}_2$ can be both split by $\mathcal{T}_2$. Namely, a collection can be split by more than one collection of sets. With the aid of computer simulation, we rule out this kind of scenario as a worst case instance.

Hence, we focus on the event for collections $\mathcal{S}_1, \mathcal{T}_1, ..., \mathcal{S}_{l-1}, \mathcal{T}_{l-1}, \mathcal{S}_l$, $\mathcal{T}_i$ splits $\mathcal{S}_i$ and is split by $\mathcal{S}_{i+1}$, for $i = 1, 2, ..., l$. We deduct performance lower bounds for this event, and then search for the minimum value of those bounds by numerical computation.

For simplicity, we assume that every set in $\mathcal{S}_i$ remains of weight $\nu \geq 0$ after being split by $\mathcal{T}_i$, and every set in $\mathcal{T}_i$ remains of weight $\mu \geq 0$ after being split by $\mathcal{S}_{i+1}$, for $i = 1, ..., l-1$. The original collections $\mathcal{S}_1, ..., \mathcal{S}_{l-1}, \mathcal{S}_l$ are disjoint. Collections $\mathcal{S}_1, \mathcal{T}_1 ..., \mathcal{S}_{l-1}, \mathcal{T}_{l-1}$ after being split are also disjoint. $\mathcal{A}$ contains the largest k sets from $\mathcal{S}_l$ and collections $\mathcal{S}_1, \mathcal{T}_1 ..., \mathcal{S}_{l-1}, \mathcal{T}_{l-1}$ after being split. $\mathcal{G}$ does not contain sets in $\mathcal{T}_i$. $\mathcal{G}$ contains the largest k sets in the original sets from $\mathcal{S}_1, ..., \mathcal{S}_{l-1}, \mathcal{S}_l$. To compute the approximation ratio of GOPS, we first have the following observation.

CLAIM 4.2. *Given a collection $\mathcal{S}$ which is split by a collection $\mathcal{T}$ into sets of weight ν. Assume set T_i in $\mathcal{T}$ has an intersection of weight μ with $\mathcal{S}$, then we can determine the minimum-weighted collection $\mathcal{T}$ which satisfies those conditions by the procedure for defining $\mathcal{T}$ in Example 4.1.*

Denote the collection of sets included in $\mathcal{G}$ from $\mathcal{S}_i$ by $\mathcal{Q}_i$, $i = 1, ..., l$. Then $\mathcal{G} = \bigcup_{i=1}^{l} \mathcal{Q}_i$. From Claim 4.2, since the weight of sets in $\mathcal{T}_i$ only serve as a lower bound of weight of sets in $\mathcal{S}_{i+1}$, we can assume that $\mathcal{T}_i$ consists of sets generated by the procedure in Example 4.1. Notice that in this way, the size of $\mathcal{T}_i$ is larger than the size of $\mathcal{S}_i$. Assume for simplicity that $\mu \geq \nu$, $\mathcal{A}$ contains the first $\min\{k, n_{\mathcal{S}_l}\}$ largest sets $\mathcal{S}_l'$ from $\mathcal{S}_l$, and $k - n_{\mathcal{S}_l}$ sets of weight μ if $n_{\mathcal{S}_l} < k$, or $\mathcal{A}$ is $\mathcal{S}_l'$ if $n_{\mathcal{S}_l} \geq k$. Assume for simplicity that every original set in $\mathcal{S}_1$ has weight w_1. Fix w_1 and $\mathcal{S}_i$, let γ_i be the ratio of $w(\mathcal{Q}_{i+1})$ and $w(\mathcal{Q}_i)$ when $\mathcal{S}_{i+1}$ is defined by the procedure in Example 4.1, for $i \geq 1$. Let δ be the ratio of $w(\mathcal{S}_l')$ and $w(\mathcal{Q}_{l-1})$ when $\mathcal{S}_l$ is defined by the procedure in Example 4.1. In general cases, $w(\mathcal{Q}_{i+1}) \geq \gamma_i w(\mathcal{Q}_i)$, for $i = 1, ..., l-1$, and $w(\mathcal{S}_l') \geq \delta w(\mathcal{Q}_{l-1})$. We first consider the $n_{\mathcal{S}_l} < k$ case. In this case, $\mathcal{S}_l'$ equals to $\mathcal{S}_l$. We can derive l lower bounds of $\frac{ALG}{GR}$ which are,

$$\frac{\gamma_{l-2} \cdots \gamma_1 \delta + (k - n_{\mathcal{S}_l})\mu / w(\mathcal{Q}_1)}{\gamma_{l-2} \cdots \gamma_1 \delta + \gamma_{l-3} \cdots \gamma_1 + \cdots + \gamma_1 + 1}. \tag{4.1}$$

$$\frac{\gamma_{l-2} \cdots \gamma_i \delta}{\gamma_{l-2} \cdots \gamma_i \delta + \gamma_{l-3} \cdots \gamma_i + \cdots + \gamma_i + 1}, \text{ for } i = l-1, ..., 1. \tag{4.2}$$

The case $n_{\mathcal{S}_l} \geq k$ implies the same lower bounds as (4.2). Therefore, the performance ratio of this instance is the minimum value of (4.1) and (4.2). However, it is still not clear under what combination of parameters ν, μ, w_1, l and $n_{\mathcal{S}_1}$, the ratio $\frac{ALG}{GR}$ obtains the minimum value. We search for such a combination of parameters via numerical computation. We observe that for fixed w_1, ν, μ, l, the minimum performance ratio converges as $n_{\mathcal{S}_1}$ becomes large. The result is presented in Table 1.

As a remark, we observe that c_β increases monotonically with respect to β. Intuitively, when splitting happens, the maximum set-weight in $\mathcal{Q}_{i+1}$ should be at least β^2 times that of $\mathcal{Q}_i$. Hence, the maximum set-weight in $\mathcal{Q}_l$ should be at least $\beta^{2(l-1)}$ times that of $\mathcal{Q}_1$. This $\beta^{2(l-1)}$ factor shows up in both denominator and numerator of (4.1), (4.2). When β, l increase, it becomes the dominating factor of lower bounds of (4.1), (4.2). Therefore, c_β increases monotonically with respect to β.

5 Experiments

5.1 Experiments setup We conduct experiments to test performance of GOPS and two SOPSs [1, 24] on the Max-k-Cover problem, GOPS on the Partial-Cover problem, and GOPS with respect to β. Experiments are conducted in in-memory and external environments. As finding exact solution to either coverage problem is very time-consuming, we use standard greedy algorithm to provide comparative quality results for both problems.

Real-world Data. Datasets from the Frequent Itemset Mining Dataset Repository[1]. We in particular appreciate the effort of authors who make the dataset retail.dat [6], accidents.dat [13], and webdocs.dat [21] available. We also generate another six datasets from network datasets[2], ca-AstroPh(astro), email-EuAll(email), amazon0601(amazon), web-Google(google), wiki-Talk(wiki), and soc-LiveJounal1(liveJ). For a network G with vertex set V and edge set E, let S_v be the union of v and its neighbors. We let $U = V$, $\mathcal{S} = \{S_v\}_{v \in V}$. An interpretation of a k-cover can be found in [22]. Finally, data from feature selection in subgraph search problem [27], we call it graph.

The statistics of real-world data are summarized in Table 2. We perform external tests on real-world data, for $k = 20, 50, 100, 500$.

Implementation and System. The swapping-based algorithms SOPS1 and SOPS2 are implemented following what suggested in [24]. GOPS is implemented following Section 4.1. We slightly modify the algorithm for the Set Cover problem proposed in [9] to implement greedy algorithm. For parameter p in [9], we set $p = 1.1$. As shown in [9], their version produces covers of almost the same quality as standard greedy implementation. The modification to the Max-k-Cover and the Partial-Cover problems follows Section 2.

The experiments are conducted on a server machine with a 2.67GHz Intel Xeon 4-core processer, running Linux 2.6.18 and gcc 4.1.2. The system has 256KB L2 cache per

[1] http://fimi.ua.ac.be/data/

[2] http://snap.stanford.edu/data/

Table 2: Statistics of real-world datasets. $|U|$- size of universe. $|\mathcal{S}|$-number of sets. S_m-maximum set-size. $\overline{S}$-average set-size. File size measured by Mb

Data	$\|U\|$	$\|\mathcal{S}\|$	S_m	$\overline{S}$	File
pumsb	2113	49047	74	74	16.7
retail	16470	88163	76	10.3	4.16
accidents	468	340184	51	33.8	35.5
kosarak	41270	990004	2498	8.10	33.0
webdocs	5267656	1692082	71472	177	1414
astro	18772	18772	504	21.1	2.01
email	74660	225409	930	1.86	2.29
amazon	403312	402439	10	8.42	22.2
google	714545	739454	456	6.90	33.6
wiki	2369181	147602	100022	34.02	34.1
liveJ	4489240	4308450	20293	16.01	483
graph	$4 \cdot 10^6$	15591	1223310	41229	4635

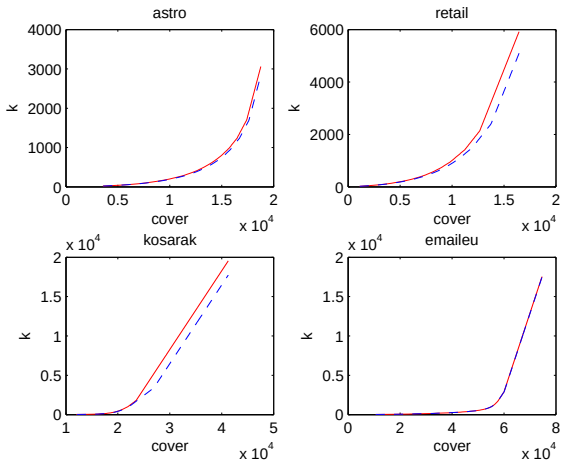

Figure 2: Partial cover results. Solid red curve for GOPS ($\beta = 1.1$), dashed blue curve for greedy algorithm [9] ($p = 1.1$).

core, 8192KB L3 cache, and 24G main memory. We implement all algorithms in C++, and use the STL implementation for all data structures.

5.2 Results 1. Comparison of GOPS and SOPSs as external algorithms on the Max-k-Cover problem (Table 4). *Space complexity.* The memory consumption for two SOPS algorithms are very similar, which increases when k increases. GOPS has the same memory usage for any k. For medium-sized sets, GOPS consumes more space than SOPSs as it has a space complexity linear to the size of the universe. While for large datasets, GOPS outperforms SOPSs (webdocs, graph etc.). *Time complexity.* GOPS has a clear win on time efficiency. It is of several orders of magnitude faster than SOPS1, and $10^2 \sim 10^3$ faster than SOPS2 on large datasets and large k. There are several tests SOPSs fail to finish within 24 hours. The time complexity of GOPS does not depend on k. It has more significant advantage on larger k ($k \geq 100$). *Solution quality.* GOPS and SOPS1 generally produce higher quality cover than SOPS2 (up to 25%, pumsb, astro, amazon). GOPS is often better than SOPS1 ($5\% \sim 10\%$). GOPS performs a little worse but almost similar to the greedy solution, e.g. 4% less in retail.

In conclusion, GOPS has a better time and space efficiency for large datasets and similar solution quality as greedy algorithm. SOPS1 is not efficient even for medium-scale datasets (e.g. retail). SOPS2 is more efficient than SOPS1 but does not produce comparatively good solution. On the other hand, SOPSs have better space complexity for medium scale datasets.

2. Comparison of GOPS and standard greedy algorithm for the Partial-Cover problem (Figure 2). To test performance on the Partial-Cover problem, we plot curve of coverage with respect to number of sets k on four datasets. The curve has an increasing derivative. GOPS performs similarly for the first half of coverage. However, as coverage approaching to full coverage, it uses more sets than greedy algorithm (about 10% more in retail, kosarak.). In practice, we often choose coverage when the corresponding derivative is small, e.g. 1 (otherwise we need much more sets to cover the same number of elements). In this case, GOPS selects similar number of sets as greedy algorithm (up to 5% more in retail).

3. Impact of β to GOPS (Tabel 3). We test the influence of β to the performance of GOPS on four real-world datasets in-memory for $k = 20, 50, 100$. As we observe from results that the running time and memory consumption do not differ too much for different β, we include only the solution quality in Table 3. For $\beta = 1.9$, the solution is often 5% worse than the best one. $\beta < 1.5$ it generally performs better. $\beta = 1.5, 1.7$ is a little worse ($1\% \sim 2\%$) than the best one. Notice that in Table 1, the worst case ratio for $\beta = 1.1$ is less than $\beta = 1.9$ because of splitting. However, worst case ratio may not capture performance of an algorithm on real-world data. Based on experimental results, we would prefer to choose $\beta < 1.5$ for GOPS.

6 Conclusions

In this paper, we propose a novel greedy-based one-pass streaming algorithm for the Max-k-Cover and the Partial-Cover problems by producing a prefix-optimal set ordering. By computer-aided simulation, we find that the algorithm has an approximation ratio around 0.3 on the Max-k-Cover problem. We show both from analysis and experiments that

Table 3: Solution quality of GOPS with respect to β. For each case, the best result is highlighted.

Data	k	1.1	1.3	1.5	1.7	1.9
pumsb	20	436	429	**438**	435	424
	50	683	**687**	681	682	664
	100	931	**940**	936	936	916
retail	20	**1117**	1086	1052	996	950
	50	2157	**2169**	2122	2079	2030
	100	**3431**	3425	3414	3366	3331
accidents	20	228	**229**	224	224	225
	50	301	**310**	303	296	301
	100	363	**377**	370	359	367
kosarak	20	**11999**	11979	11923	11974	11676
	50	**14777**	14750	14740	14770	14608
	100	**16672**	16657	16659	16663	16545

our one-pass algorithm has better efficiency and accuracy than swapping-based algorithms [1,24] for the Max-k-Cover problem. We also demonstrate by experiments that it has similar performance as standard greedy algorithm for the Partial-Cover problem.

There are several questions left for future work. First, finding a swapping-based one-pass algorithm which has a $1/2$ approximation ratio or improving the $1/2$ lower bound. Second, seeking a rigorous proof of GOPS. Third, it is of practical interest to design algorithms for set coverage problems with very large universe that cannot reside in memory.

References

[1] G. AUSIELLO, N. BORIA, A. GIANNAKOS, G. LUCARELLI, AND V. T. PASCHOS, *Online maximum k-coverage*, in FCT, 2011.

[2] B. BERGER, J. ROMPEL, AND P. W. SHOR, *Efficient nc algorithms for set cover with applications to learning and geometry*, J. Comput. Syst. Sci., 49 (1994).

[3] G. E. BLELLOCH, R. PENG, AND K. TANGWONGSAN, *Linear-work greedy parallel approximate set cover and variants*, SPAA, 2011.

[4] G. E. BLELLOCH, H. V. SIMHADRI, AND K. TANGWONGSAN, *Parallel and i/o efficient set covering algorithms*, SPAA, 2012.

[5] A. BORODIN AND R. EL-YANIV, *Online computation and competitive analysis*, Cambridge University Press, New York, NY, USA, 1998.

[6] T. BRIJS, G. SWINNEN, K. VANHOOF, AND G. WETS, *Using association rules for product assortment decisions: A case study*, in Knowledge Discovery and Data Mining, 1999.

[7] V. CEVHER AND A. KRAUSE, *Greedy dictionary selection for sparse representation*, Applied Optics, 50 (2011).

[8] F. CHIERICHETTI, R. KUMAR, AND A. TOMKINS, *Max-cover in map-reduce*, WWW, 2010.

[9] G. CORMODE, H. KARLOFF, AND A. WIRTH, *Set cover algorithms for very large datasets*, CIKM, 2010.

[10] U. FEIGE, *A threshold of ln n for approximating set cover*, J. ACM, 45 (1998).

[11] G. FORMAN, *An extensive empirical study of feature selection metrics for text classification*, J. Mach. Learn. Res., 3 (2003).

[12] R. GANDHI, S. KHULLER, AND A. SRINIVASAN, *Approximation algorithms for partial covering problems*, J. Algorithms, 53 (2004).

[13] K. GEURTS, G. WETS, T. BRIJS, AND K. VANHOOF, *Profiling high frequency accident locations using association rules*, in Proceedings of the 82nd Annual Transportation Research Board, 2003.

[14] F. C. GOMES, C. N. MENESES, P. M. PARDALOS, AND G. V. R. VIANA, *Experimental analysis of approximation algorithms for the vertex cover and set covering problems*, Comput. Oper. Res., 33 (2006).

[15] R. GOMES AND A. KRAUSE, *Budgeted nonparametric learning from data streams*, ICML, 2010.

[16] T. GROSSMAN AND A. WOOL, *Computational experience with approximation algorithms for the set covering problem*, European Journal of Operational Research, 101 (1997).

[17] D. S. JOHNSON, *Approximation algorithms for combinatorial problems*, in Proceedings of the fifth annual ACM symposium on Theory of computing, STOC, 1973.

[18] D. KEMPE, J. KLEINBERG, AND E. TARDOS, *Maximizing the spread of influence through a social network*, KDD, 2003.

[19] J. KÖNEMANN, O. PAREKH, AND D. SEGEV, *A unified approach to approximating partial covering problems*, ESA, 2006.

[20] A. KRAUSE AND C. GUESTRIN, *Near-optimal observation selection using submodular functions*, AAAI, 2007.

[21] C. LUCCHESE, S. ORLANDO, R. PEREGO, AND F. SILVESTRI, *Webdocs: a real-life huge transactional dataset*.

[22] A. S. MAIYA AND T. Y. BERGER-WOLF, *Sampling community structure*, WWW, 2010.

[23] G. L. NEMHAUSER, L. A. WOLSEY, AND M. L. FISHER, *An analysis of approximations for maximizing submodular set functions*, Mathematical Programming, 14 (1978).

[24] B. SAHA AND L. GETOOR, *On maximum coverage in the streaming model & application to multi-topic blog-watch*, SDM, 2009.

[25] M. THOMA, H. CHENG, A. GRETTON, J. HAN, H.-P. KRIEGEL, A. SMOLA, L. SONG, P. YU, X. YAN, AND K. BORGWARDT, *Near-optimal supervised feature selection among frequent subgraphs*, SDM, 2009.

[26] C. YU, L. LAKSHMANAN, AND S. AMER-YAHIA, *It takes variety to make a world: diversification in recommender systems*, EDBT, 2009.

[27] D. YUAN, P. MITRA, H. YU, AND C. L. GILES, *Iterative graph feature mining for graph indexing*, ICDE, 2012.

Table 4: Comparison of performance on real-world datasets in external environment. $p = 1.1$ for Greedy Algorithm [9]. $\beta = 1.1$ for GOPS. An "/" entry means the running time exceeds pre-set wall-time (24 hours). The best results in solution (excludes Greedy), time and RAM are highlighted.

		Solution				Time(s)			RAM(Mb)		
Data	k	GOPS	SOPS1	SOPS2	Greedy	GOPS	SOPS1	SOPS2	GOPS	SOPS1	SOPS2
pumsb	20	**440**	405	330	445	**1.90**	41.3	3.36	61.2	**26.5**	**26.5**
	50	**693**	643	535	718	**1.88**	150.6	4.43	61.2	**26.5**	26.6
	100	**955**	912	746	1008	**1.90**	399.6	4.77	61.2	**26.6**	26.8
	500	1710	**1783**	1476	1861	**1.89**	3104	7.18	61.2	28.2	**28.1**
retail	20	**1103**	922	831	1115	**0.606**	143.5	6.24	25.3	**10.8**	**10.8**
	50	**2160**	1966	1618	2204	**0.594**	688.9	10.6	25.3	11.1	**10.9**
	100	**3433**	3191	2686	3516	**0.604**	2010	17.5	25.3	11.4	**11.2**
	500	**7727**	7540	6507	7986	**0.593**	19933	38.1	25.3	12.4	**12.3**
accidents	20	**233**	227	179	238	**6.76**	175.2	11.9	163.3	16.5	**16.4**
	50	**307**	303	243	319	**6.73**	485.6	12.6	163.3	16.6	**16.5**
	100	361	**370**	302	386	**6.77**	959.4	13.9	163.3	16.8	**16.6**
	500	**468**	**468**	**468**	486	**6.75**	5135	16.7	163.3	**17.3**	**17.3**
kosarak	20	**11999**	11890	9858	12115	**12.96**	16234	692.3	**227.4**	240.2	239.9
	50	**14777**	14714	11948	14902	**12.97**	47651	829.6	**227.4**	242.4	240.6
	100	**16672**	/	14237	16787	**12.95**	/	974.0	**227.4**	/	242.0
	500	**20241**	/	17319	20380	**12.94**	/	1277	**227.4**	/	247.7
webdoc	20	**550894**	/	489226	551955	**585.1**	/	49370	**4336**	/	8069
	50	**886041**	/	/	886681	**578.6**	/	/	**4336**	/	/
	100	**1195202**	/	/	1195715	**567.8**	/	/	**4336**	/	/
	500	**1935057**	/	/	1935616	**564.6**	/	/	**4336**	/	/
astro	20	**3556**	3370	2768	3594	**0.24**	366.4	4.45	**14.9**	61.8	61.9
	50	**5627**	5376	4444	5676	**0.24**	406.2	6.80	**14.9**	62.3	62.3
	100	**7643**	7473	6125	7729	**0.24**	900.7	8.74	**14.9**	62.9	62.9
	500	13274	**13338**	10737	13517	**0.24**	7773	15.5	**14.9**	66.7	66.6
email	20	**10641**	9825	8868	10625	**0.83**	4249	185.7	**47.9**	115.2	115.5
	50	**18139**	17504	15930	18159	**0.84**	17898	348.8	**47.9**	116.6	117.1
	100	**26076**	25147	23750	26085	**0.83**	44397	496.2	**47.9**	118.2	118.9
	500	**49163**	/	41320	49166	**0.83**	/	901.7	**47.9**	/	127.5
amazon	20	**200**	185	143	200	**2.51**	127.9	7.12	108.3	5.01	**5.00**
	50	**500**	454	390	500	**2.53**	786.5	15.8	108.3	5.05	**5.04**
	100	**1000**	900	730	1000	**2.49**	3445	27.2	108.3	5.12	**5.11**
	500	**5000**	/	3812	5000	**2.49**	/	129.8	108.3	/	**5.84**
google	20	**5131**	4646	4234	5118	**4.25**	5064	240.2	207	**56.3**	56.4
	50	**9934**	9519	8117	9943	**4.27**	22536	467.3	207	56.9	**57.1**
	100	**15409**	14874	13129	15456	**4.18**	65474	689.4	207	**57.5**	57.8
	500	**40320**	/	33163	40356	**4.21**	/	1869	207	/	**62.4**
wiki	20	**350701**	/	323117	349984	**30.9**	/	3401	**265**	/	7961
	50	**537415**	/	477148	537454	**31.0**	/	5088	**265**	/	7989
	100	**725158**	/	648897	724932	**30.1**	/	7003	**265**	/	8045
	500	**1352828**	/	1171912	1352487	**30.2**	/	11790	**265**	/	8268
liveJ	20	**83084**	/	71422	83036	**226.7**	/	21412	**1714**	/	1963
	50	**108086**	/	95340	107972	**226.9**	/	37260	**1714**	/	1966
	100	**141635**	/	/	142137	**226.2**	/	/	**1714**	/	/
	500	**328233**	/	/	329768	**226.2**	/	/	**1714**	/	/
graph	20	**3537812**	/	2929189	3565074	**782.2**	/	3171	**1037**	/	1157
	50	**3734064**	/	3253627	3771996	**787.5**	/	5157	**1037**	/	3907
	100	**3832753**	/	3522265	3860914	**774.3**	/	5257	**1037**	/	4000
	500	**3939107**	/	3783826	3953505	**776.3**	/	12119	**1037**	/	4149