Set Cover Algorithms For Very Large Datasets

Graham Cormode
AT&T Labs–Research
graham@research.att.com

Howard Karloff
AT&T Labs–Research
howard@research.att.com

Anthony Wirth*
Department of Computer Science and Software Engineering
The University of Melbourne
awirth@unimelb.edu.au

ABSTRACT

The problem of Set Cover—to find the smallest subcollection of sets that covers some universe—is at the heart of many data and analysis tasks. It arises in a wide range of settings, including operations research, machine learning, planning, data quality and data mining. Although finding an optimal solution is NP-hard, the greedy algorithm is widely used, and typically finds solutions that are close to optimal.

However, a direct implementation of the greedy approach, which picks the set with the largest number of uncovered items at each step, does not behave well when the input is very large and disk resident. The greedy algorithm must make many random accesses to disk, which are unpredictable and costly in comparison to linear scans. In order to scale Set Cover to large datasets, we provide a new algorithm which finds a solution that is provably close to that of greedy, but which is much more efficient to implement using modern disk technology. Our experiments show a ten-fold improvement in speed on moderately-sized datasets, and an even greater improvement on larger datasets.

Categories and Subject Descriptors

H.2.8 [**Database Management**]: Database Applications—*Data mining*; F.2.2 [**Analysis of Algorithms and Problem Complexity**]: Nonnumerical Algorithms and Problems

General Terms

Algorithms, Experimentation

Keywords

set cover, greedy heuristic, disk friendly

*Work partially done at AT&T Labs–Research.

CIKM '10, October 26–30, 2010, Toronto, Ontario, Canada.

1. INTRODUCTION

The problem of Set Cover arises in a surprisingly broad number of places. Although presented as a somewhat abstract problem, it captures many different scenarios that arise in the context of data management and knowledge mining. The basic problem is that we are given a collection of sets, each of which is drawn from a common universe of possible items. The goal is to find a subcollection of sets so that their union includes every item from the universe. Places where Set Cover occurs include:

- In operations research, the problem of choosing where to locate a number of facilities so that all sites are within a certain distance of the closest facility can be modeled as an instance of Set Cover. Here, each set corresponds to the sites that are covered by each possible facility location.
- In machine learning, a classifier may be based on picking examples to label. Each item is classified based on the example(s) that cover it; the goal is to ensure that all items are covered by some example, leading to an instance of Set Cover.
- In planning, it is necessary to choose how to allocate resources well even when demands are predicted to vary over time. Variations of Set Cover have been used to capture these requirements [13].
- In data mining, it is often necessary to find a "minimal explanation" for patterns in data. Given data that correspond to a number of positive examples, each with a number of binary features (such as patients with various genetic sequences), the goal is to find a set of features so that every example has a positive example of one of these features. This implies an instance of Set Cover in which each set corresponds to a feature, and contains the examples which have that feature.
- In data quality, a set of simple rules which describes the observed data helps users understand the structure in their data. Given a set of rules that are consistent with the observed data, the tableau generation problem is to find a subset of rules which explains the data without redundancy. This is captured by applying Set Cover to the collection of rules [8].
- In information retrieval, each document covers a set of topics. In response to a query, we wish to retrieve the smallest set of documents that covers the topics in the query, i.e., to find a set cover [15].

In these situations, and others too, the goal is to find a *minimum* set cover: a cover which contains the smallest number of sets. This corresponds to the simplest explanation in mining, the cheapest solution to facility location, etc. Unsurprisingly, the SET COVER problem, finding the smallest set cover, is NP-hard, and so we must find efficient approximate techniques which can find a good solution. Fortunately, there is a simple algorithm that provides a guaranteed solution: the natural "greedy" algorithm, which repeatedly picks the set with the most uncovered items. The greedy algorithm is guaranteed to find a cover which is at most a logarithmic factor (in the number of items in the universe) larger than the optimal solution. Moreover, no algorithm can guarantee to improve this approximation by much [5].

For even moderately-sized instances, one might complain that this logarithmic factor is too large. However, it has been observed across a large number of instances that this method is surprisingly good in practice, especially when compared with other approximation algorithms [10, 9]. The greedy approach is often found to choose only a small percentage ($< 10\%$) more sets than the optimal solution, and is dramatically cheaper to compute. Note that the optimal solution is usually determined by exhaustive exploration of exponentially many options. Therefore, the greedy method for SET COVER is widely used to solve instances across the broad set of applications outlined.

Having said this, a direct implementation of this method scales surprisingly poorly when the data size grows. As our ability to record data via sensors, instrumentation and logging increases, the size of the instances of SET COVER to solve can rapidly become very large: (many) millions of sets, drawn over universes of (many) millions of items. The growth of such data is rapidly outstripping that of main memory (which has traditionally increased at a much slower rate than computing power or data volumes). It is therefore increasingly important to deal with instances that do not fit conveniently into main memory, but which are disk resident.

In these cases, the seemingly simple greedy method becomes surprisingly challenging. The basic idea, to repeatedly pick the set with the maximum number of uncovered elements, becomes very costly, due to the need to update all other sets every time the current "largest" set is selected, to reflect the items which have just been covered. Our experiments on several natural implementation choices for data that are disk resident (detailed later) took many hours to execute on even moderately-sized instances of only megabytes to gigabytes. This presents a fundamental problem: how to scale this important computation to modern data sizes?

Our Contributions. We consider the problem of solving large instances of SET COVER. In doing so, the contributions in this paper are:

- We formalize the problem of finding set covers on very large data sets, and provide detailed discussion of how to implement the traditional greedy algorithm.
- We introduce, as an alternative to greedy, a new algorithm which is much more appropriate for large datasets, in particular on datasets which are resident on disk.
- We show that the new algorithm gives a guarantee similar in nature to that of the original greedy heuristic, but has well-bounded running time. We argue that the new algorithm is very "disk-friendly."
- We implement two versions of our method and variations on the greedy algorithm, to study a selection of implementation choices. Through experiments on real datasets spanning several orders of magnitude, we show that the new method scales very gracefully to large datasets. With the problem instance stored on disk, our algorithm is over ten times faster than the standard greedy algorithm, even for instances with hundreds of thousands of sets. On the largest instance on which we tested our algorithm, with over a million sets and over five million items, our algorithm is over 400 times faster than the standard greedy approach.

  Despite this, the quality of the results, measured in the number of sets needed to find a cover, is as good as or better than that of the original greedy heuristic. Intriguingly, the running time of our disk-based algorithm is close to that of its memory-resident version, suggesting that the new algorithm is not I/O bound; indeed, the disk-based implementation is appreciably faster than the memory-resident version of the greedy heuristic!

Outline of the paper. Section 2 lays down some technical background, and suggests some natural approaches to implementing the greedy algorithm. Section 3 describes our new algorithm and analyzes its worst case behavior, while Section 4 shows that it is highly efficient in practice and provides excellent results. We outline some possible extensions in Section 5, and conclude in Section 6.

1.1 Prior work

Despite the importance of SET COVER, there has been relatively little study of how to find covers efficiently until quite recently. The oldest study of this question is due to Berger *et al.* [1] (15 years after the heuristic was first analyzed). That work is concerned with parallelizing the heuristic, by allowing multiple computing entities with shared memory to choose sets at the same time. There, randomization is necessary to ensure that multiple processors do not pick sets which cover the same elements redundantly. The algorithm assumes full random access to memory for all processors, and so does not seem to apply to the case of disk-based data.

More recently, the ideas of Berger *et al.* have been applied to the distributed case, to find (partial) set covers under the MapReduce paradigm [4]. This approach requires randomization, and requires a number of passes over the data cubic in the logarithm of the size of the data in the worst case. Their algorithm used hundreds of invocations of MapReduce on a dataset with about 5M sets to match the quality of the greedy solution.

Lastly, Saha and Getoor develop an efficient algorithm for SET COVER in the streaming model [15]. Their approach requires multiple passes, however, and is built on an algorithm for MAX k-COVERAGE. Their analysis shows that it requires $O(\log^2 n)$ passes (at least logarithmically many even in the best case) to find an approximate set cover. For the large datasets we consider, this equates to potentially hundreds of passes. For a gigabyte-sized dataset, a single pass takes time on the order of a minute, so the total cost can be many hours. In experiments in [15], the method achieves solutions

S_1	ABCDE	S_2	ABDFG
S_3	AFG	S_4	BCG
S_5	GH	S_6	EH
S_7	CI	S_8	A
S_9	E	S_{10}	I

Figure 1: Example input of $m = 10$ sets over the universe $\{\mathbf{A}, \mathbf{B}, \mathbf{C}, \mathbf{D}, \mathbf{E}, \mathbf{F}, \mathbf{G}, \mathbf{H}, \mathbf{I}\}$ of size 9.

that are somewhat worse than the greedy (offline) solution, by around 10%. The absolute time cost of the method is not described.

2. TECHNICAL BACKGROUND

2.1 The Set Cover problem

We consider the standard (unweighted) Set Cover problem. There are a universe X of n items and a collection $\mathcal{S}$ of m subsets of X: $\mathcal{S} = \{S_1, S_2, \ldots, S_m\}$. We assume that the union of all of the sets in $\mathcal{S}$ is X, with $|X| = n$. The aim is to find a subcollection of sets in $\mathcal{S}$, of minimum size, that covers all of X. An example input is shown in Figure 1.

Set Cover was one of the early problems to have been identified as NP-hard; Vertex Cover is an important special case.

Greedy heuristic. The best-known algorithm for Set Cover is based on a greedy heuristic [11]. Let Σ be the set of indices of sets in the solution so far and let C be the elements covered so far. Initially (see Figure 1) there are no sets in the solution and every element is uncovered, so that $\Sigma = \emptyset$ and $C = \emptyset$. We repeat the following steps until all elements are covered, that is, $C = X$:

> Choose (one of) the set(s) with the maximum value of $|S_i \setminus C|$; let the index of this set be i^*.
> Add i^* to Σ and update C to $C \cup S_{i^*}$.

Figure 2 shows the effect of this algorithm on the sample input shown in Figure 1. Initially the largest uncovered set is of size 5 (there are two such sets, so the algorithm arbitrarily picks $S_1 = ABCDE$). After this step, many items are covered—these are shown as lower case in Figure 2. Now the largest "uncovered set" is of size 2: there are two sets of this size, but they both contain the same uncovered items (FG), and from these the greedy algorithm arbitrarily picks S_2. The next step picks a set containing H, and the final step covers the last remaining uncovered item, I, and terminates. Note that the optimal solution picks only three sets: S_2, S_6, S_7 are sufficient to cover all items. In this case, it is easy to argue that this is indeed optimal: consider elements D, H and I. Since these never appear in any set together, the optimal solution must contain at least three sets, one for each of these elements.

It is an oft-repeated observation that this method happens to perform well in practice (see, for example, [8]). It clearly runs in polynomial time, and is also an approximation algorithm for Set Cover (indeed, it is often one of the early examples shown in textbooks on Approximation Algorithms).

Approximating Set Cover.

After step 1:

S_2	abdFG	S_3	aFG	S_4	bcG
S_5	GH	S_6	eH	S_7	cI
S_8	a	S_9	e	S_{10}	I

After step 2:

S_3	afg	S_4	bcg	S_5	gH	S_6	eH
S_7	cI	S_8	a	S_9	e	S_{10}	I

After step 3:

S_3	afg	S_4	bcg	S_6	eh	S_7	cI
S_8	a	S_9	e	S_{10}	I		

Figure 2: Execution of the greedy algorithm on example input

Lemma 1. *The greedy algorithm in the previous paragraph produces a solution within a factor $1 + \ln n$ of the optimum for* Set Cover.

It is worth repeating here a short proof of this lemma.

Proof. Let the number of sets in the optimal solution be σ^*. Let C_t be the set of covered elements after t iterations of the greedy algorithm. We know at each iteration that there is some set that covers at least $|X \setminus C_t|/\sigma^*$ previously-uncovered elements, otherwise there would not be an optimum solution of size σ^*. Since the greedy algorithm chooses the set with the largest number of uncovered elements, it covers at least $|X \setminus C_t|/\sigma^*$ new elements. Therefore

$$|X \setminus C_{t+1}| \leq |X \setminus C_t|(1 - \frac{1}{\sigma^*}),$$

which means that after t iterations

$$|X \setminus C_t| \leq n\left(1 - \frac{1}{\sigma^*}\right)^t < ne^{-t/\sigma^*}.$$

Consequently, if t is at least $\sigma^* \ln n$, the number of uncovered elements is strictly less than 1, and thus a solution has been found. We note that $\sigma^* \ln n$ might not be an integer, and thus we can only guarantee a solution of size $\lceil \sigma^* \ln n \rceil$. This is strictly bounded by $1 + \sigma^* \ln n \leq \sigma^*(1 + \ln n)$. □

Interestingly, in an approximation sense, this is essentially the best that can be achieved for Set Cover. Feige showed that no (efficient) algorithm can guarantee an approximation of the problem within a factor of $(1 - o(1)) \ln n$ unless there are efficient algorithms for the class NP [5].

A more naive algorithm. Several previous studies of the quality of the greedy algorithm have compared results to a so-called *naive* heuristic [4]. It proceeds as follows:

- Sort the (indices i of the) sets S_i into descending order according to $|S_i|$.
- For each S_i in this order, until $C = X$:
  - If $|S_i \setminus C| > 0$: add i to Σ and update C.

In the worst case this heuristic cannot provide an approximation with approximation ratio better than $n/6$. Consider an instance in which $S_i = \{2i - 1, 2i, \ldots, 2k + i\}$ for some k and all $i \leq k$. Now, $|S_i| = 2k + 2 - i$, so the naive algorithm will process the sets in the order $S_1, S_2, \ldots, S_k$, each time adding the set S_i to the solution, because it contains one uncovered element, $2i + 1$. The optimal solution, however,

A	1, 2, 3, 8	B	1, 2, 4	C	1, 4, 7
D	1, 2	E	1, 6, 9	F	2, 3
G	2, 3, 4, 6	H	4, 6	I	7, 10

Figure 3: Inverted index for example data

comprises just S_1 and S_k. Therefore the ratio of the sizes of the naive and optimal solutions is $k/2 = n/6$.

Nevertheless, we include this heuristic in our experimental study for comparison with the greedy method and our new algorithm, and for consistency and comparison with prior work which has also used this heuristic.

2.2 Memory and disk considerations

The chief problem with efficiently implementing the greedy algorithm is that it demands picking the set with the largest number of uncovered items. As each new set is chosen, because it covers items that might be present in other sets, it is necessary to find all sets which contain the covered items, and adjust the count of uncovered items in each set accordingly. Thus, algorithms for Set Cover must retrieve information from disk or main memory (depending on the implementation) in order to calculate the sizes of sets and to determine which items are in which set and vice versa. On large problem instances, these frequent calls for data have a significant effect on the running time of the algorithm. Although modern computers have large amounts of main memory, there is a memory hierarchy, and effective use of the cache often relies on locality of reference. This effect is even more pronounced on disk, since the cost of processing a single block once in main memory is, in most circumstances, orders of magnitude less than that of retrieving the block from disk. Hence, random access to data on disk can be highly expensive. Indeed models for the running time of disk-based algorithms often essentially ignore the internal computation costs, since these can be dominated by the I/O cost. In this paper, we analyze the performance of algorithms based on this *external memory* model of the running time.

2.3 Greedy "cooked" two ways

We assume that the problem instance specifies for each i a succinct description of the elements that are in set S_i. This consumes $O(\sum_{i=1}^{m} |S_i|)$ words of memory or disk, assuming an element can be represented in one word.

There are two canonical approaches to implementing the greedy heuristic. At each step of the greedy algorithm we need to find a set that has maximum $|S_i \setminus C|$. The first approach involves the use of an inverted index (or file), the second involves multiple passes over the original data. In this section, we provide a basic description of each approach. Further details and optimizations are described in Sections 4 and 5.

Inverted index. We can find the maximum $|S_i \setminus C|$ by maintaining these values in a large priority queue. In order to have up-to-date $|S_i \setminus C|$ values, as a set S_{i^*} is added to the solution, we need to determine which other sets contain the items in S_{i^*}, that is, those items freshly included in C. This can be done by use of an inverted index, in which for each item j we have a succinct representation of

$$T_j = \{i : j \in S_i\}.$$

As a preprocessing step, the algorithm creates the T_j's, and then looks them up as the greedy iterations proceed. Figure 3 shows the inverted index for the sample dataset shown in Figure 1. For each element, the index lists the indices of the sets in which it belongs.

If we assume a cost model in which random accesses to locations in the memory hierarchy take a constant amount of time, the use of an inverted index seems to make the maintenance of the priorities in the priority queue efficient In this cost model, the running time of this approach is $O((\log m) \sum_{j=1}^{n} |T_j|)$. Note this is somewhat of an overestimate as the updating of the $|S_i \setminus C|$ in the priority queue can be done once for each i at each iteration of the algorithm. Generating the inverted index requires examining the full description of all the sets, which is subsumed by the expression above.

However, the (pre-)process that generates the inverted index is very unlikely to observe locality of reference in its construction of the T_j sets. Moreover, consider retrieving the T_j's as the greedy algorithm proceeds: it seems very hard to predict which T_j will be needed at which stage in the algorithm, and the memory accesses are likely to be arbitrary. Hence, in practice, the cost of this algorithm can be painfully high due to the random accesses to many locations on disk.

Multiple passes. An alternative approach avoids the priority queue and the inverted index completely. Instead, we simply maintain the set C of elements covered so far. At each iteration, we loop through all of the (previously-unadded) sets and note the value of $|S_i \setminus C|$, by a (simple) comparison of the elements in the two sets. The running time depends on $O(\sigma \sum_i |S_i|)$, or equivalently (for comparison with the inverted index approach) $O(\sigma \sum_j |T_j|)$, where σ is the size of the solution obtained.

The algorithm makes a linear number of passes over the data, which seems efficient when data are resident on disk, since the number of random accesses (seeks of a disk head) is minimized. However, it will be very slow when σ is large, since it essentially has to read through the entire dataset to add a single set to the solution. An optimization can be applied when the number of items remaining to be covered becomes small. Once $|S_i^* \setminus C|$ drops below a certain threshold τ, we might take a different approach. For each value $T \leq \tau$ we loop through the (remaining) sets and then add a set if $|S_i \setminus C|$ equals T. Determining the best value of τ requires some experimentation, but the running time is now $O((\sigma_\tau + \tau) \sum_j |T_j|)$, where σ_τ is the number of sets in the solution whose size is at least τ.

This multiple pass approach has the advantage that it sweeps through memory sequentially, without needing to generate, nor access, an inverted index. We later study empirically the tradeoff between the large number of passes in this algorithm and the cost of the random accesses in the inverted file approach.

3. OUR NEW APPROACH

3.1 Greed is not good

Reflecting on the two approaches to the greedy algorithm, there is considerable effort expended to find the set S_{i^*} with the maximum value of $|S_i \setminus C|$. This requires either a priority queue and an inverted index, or many passes through

At start of step 1:

4–7	ABCDE, ABDFG
2–3	AFG, BCG, GH, EH, CI, AFG
1	A, E, I

At start of step 2:

2–3	aFG, bcG, GH, eH, cI, (abd)FG
1	a, e, I

After the third set has been selected:

2–3	bcg, gH, eH, cI, (abd)fg
1	a, e, I

At start of step 3:

1	a, e, I, (g)H, (e)H, (c)I

Figure 4: The new algorithm executed on the sample input

the instance description. What if we did not insist on finding the set with the absolute maximum $|S_i \setminus C|$, but just a set whose uncovered element count were close to maximal? Could we do this in a way that not only interacted with disk (or memory) in a friendly manner, with few passes through the data, but also had a reasonably good approximation factor and performed well in practice?

More formally, suppose instead of picking the exact maximum, we instead chose a set which is within a very small constant factor of the largest. We first consider the impact on the approximation factor:

LEMMA 2. *If an iterative algorithm always chooses a set S_c to add to the solution with*

$$|S_c \setminus C| \geq \alpha \max_i |S_i \setminus C|, \tag{1}$$

for $\alpha \leq 1$, then it has approximation factor which is at most $1 + (\ln n)/\alpha$ for SET COVER.

PROOF. Consider the proof of Lemma 1. The greedy algorithm guarantees that it chooses a set with at least $|X \setminus C_t|/\sigma^*$ uncovered elements. In the algorithm at hand, we can guarantee that the set chosen has at least $\alpha|X \setminus C_t|/\sigma^*$ uncovered elements. This α factor carries through the calculations of that proof, so that when t is at least $\sigma^*(\ln n)/\alpha$ there is fewer than one uncovered element. Again this quantity might not be an integer, so the solution size is strictly less than

$$1 + \frac{\sigma^* \ln n}{\alpha} \leq \sigma^* \left(1 + \frac{\ln n}{\alpha}\right).$$

□

Munagala *et al.* [14] suggest a similar heuristic for the PIPELINED SET COVER problem, but our result in the Lemma above is novel.

3.2 The algorithm

The approach we use is to partition the sets into subcollections based on the sizes of the sets. To that end, we select a real-valued parameter $p > 1$, which will govern both the approximation factor and running time of our algorithm. Initially, we assign set S_i to subcollection $\mathcal{S}^{(k)}$ if $p^k \leq |S_i| < p^{k+1}$; let K be the largest k with non-empty $\mathcal{S}^{(k)}$. The algorithm then proceeds in two loops:

- For $k \to K$ down to 1:
  - For each set S_i in $\mathcal{S}^{(k)}$:
    - If $|S_i \setminus C| \geq p^k$: add i to Σ and update C.
    - Else: let set $S_i \leftarrow S_i \setminus C$ and add the updated set to subcollection $\mathcal{S}^{(k')}$, where the new set size satisfies $p^{k'} \leq |S_i| < p^{k'+1}$ (and therefore $k' < k$).
- For each set S_i in $\mathcal{S}^{(0)}$:
  - If $|S_i \setminus C| = 1$: add i to Σ and update C.

Note that for values of p close to 1, the algorithm will separately track subcollections of sets whose sizes are $1, 2, 3, \ldots, 1/p$. Related notions are outlined in Berger *et al.* [1] for designing an efficient parallel version of the greedy algorithm for SET COVER. However, their phases concern not the sizes of the sets, but the number of sets that the elements are in, that is, the $|T_j|$ values.

Example. Figure 4 shows an example execution of the algorithm on the sample input of Figure 2, with parameter $p = 2$. Thus, we break the sets initially into those of size 1, 2–3 and 4–7. The algorithm first considers the sets of size 4–7, and selects the first set found, ABCDE (indicated via underlining). The next set now has only 2 uncovered items (F and G), so the new set is appended to list 2, and the first step finishes. At the start of the next step, many items are now covered (indicated in lower case); however, the "uncovered sizes" of the sets are not known to the algorithm until these sets are inspected. The set FG has been written to the end of list 2: the original items ABD have been removed, and are shown in parentheses. In step 2, AFG has two uncovered elements and so is added to the solution. Consequently, when the algorithm passes through the list of sets supposed to be of size 2–3, it finds that there are no sets of "uncovered size" remaining in this range, due to items' being covered. In step 3, each of the sets I and GH has one uncovered item when inspected, and so is selected (and underlined). The remaining sets in list 1 have no uncovered items. Thus, the chosen cover is ABCDE, AFG, GH, I. Observe that this is different from the cover chosen by greedy, but has the same number of sets (four).

3.3 Analyzing the algorithm

The approximation factor. The key fact is that a set added to the solution has at least p^k uncovered elements. From the design of the algorithm the following is clear.

PROPOSITION 1. *At the time in which subcollection $\mathcal{S}^{(k)}$ is being processed, there is no set that has at least p^{k+1} uncovered elements.*

Consequently, since $p^k/p^{k+1} = 1/p$, we know that the set chosen satisfies (1) with $\alpha = 1/p$ and therefore Lemma 2 implies the following.

LEMMA 3. *The algorithm described in this section is a $1 + p \ln n$ approximation for* SET COVER.

Running time. Consider what happens to some set S_i. Whenever it is processed, each of its elements is checked for presence in C. It is then either added to the solution and

never accessed again, or it is added to a subcollection of *smaller* sets. In the worst case, the set S_i is moved to the next subcollection in each round. Moreover, it is guaranteed to shrink by at least factor of p every second time it is moved. Hence, we can bound the worst case total number of items in all of the manifestations of S_i based on the following geometric series

$$|S_i| + \frac{|S_i|}{p} + \frac{|S_i|}{p^2} + \cdots = \frac{|S_i|}{1 - 1/p} = \left(1 + \frac{1}{p-1}\right)|S_i|\,.$$

Consequently, we can bound the worst case running time of the algorithm by $O([1 + 1/(p-1)]\sum_i |S_i|)$, which is at most a factor of $1 + 1/(p-1)$ as great as the time to scan the data. In practice, we do not expect to see such worst case examples: it is more likely that a set, if not picked, will move down multiple levels, rather than just one. More importantly, if the subcollections are written to disk, each in a separate file, then the file accesses are sequential.

Memory/disk considerations. Our algorithm's main appeal is that it behaves extremely well on external memory, and reasonably well on main memory. The initial partitioning of $\mathcal{S}$ can be done with one sweep through $\mathcal{S}$. Subsequently, within each subcollection $\mathcal{S}^{(k)}$, the sets can be processed in two passes. In the first pass, we simply add each set sequentially, either in the initial phase of partitioning $\mathcal{S}$ or when processing some $\mathcal{S}^{(k')}$ with $k' > k$. The second pass is the step in the *for* loop in which each set is examined in turn for its $|S_i \setminus C|$ value. The other benefit of our algorithm is that even when a set is pushed down to a lower subcollection, it has become smaller.

Referring to the analysis in the paragraph on running time, if information is read from disk in blocks of size B, then reading the input requires approximately $D = \lceil \sum_i |S_i|/B \rceil$ disk reads. Each subcollection $\mathcal{S}^{(k)}$ requires at most

$$\left\lceil \frac{2|S_i|}{Bp^{K-k}} \right\rceil$$

disk reads. Summing this over all subcollections, we have an upper bound of

$$2D\left(1 + \frac{1}{p-1}\right) + 2K$$

disk reads, where $K \leq \log_p \max |S_i|$.

4. EXPERIMENTS

4.1 Datasets and Experimental Environment

The datasets used in our experiments come from the Frequent Itemset Mining Dataset Repository, based on workshops in 2003 and 2004 [7]. Each line in one of these files describes a set in a natural way, as a white space-separated sequence of integers, where each integer represents an item.

Table 1 describes the properties of the problem instances. We acknowledge in particular the authors who made available the accidents.dat [6], retail.dat [2] and webdocs.dat datasets [12].

System. The system used for conducting the experiments is a 2.8GHz Intel Core i7 running the Mac OS X operating system version 10.6.3 under a light load. This system has 4 cores, 256KB L2 cache per core, 8MB L3 cache, 8GB of main memory and 2TB of hard disk.

4.2 Implementation details

The item and set indices, j and i, were stored as long integers, requiring eight bytes. For each problem instance, the values of $m, n, \max |T_j|$, and $\max |S_i|$ were calculated in advance, so that only the required resources were used by the SET COVER algorithms. The sorting of the $|S_i|$ in the naive algorithm was done using the built-in C `qsort` function. Maintenance of the $|S_i \setminus C|$ values and selection of a minimum such value were achieved with an array-based heap implementation of a priority queue.

The external memory greedy algorithms write out a copy of the problem instance (in S_i form) to disk in a program-friendly format before doing further processing. Constructing an inverted index on external memory required calculating the $|T_j|$ values; this was done by a first pass through the program-friendly disk-stored S_i data.

For each algorithm, we recorded the maximum amount of main memory allocated on the heap at any one time during the program execution. We also recorded the total number of bytes written to and read from disk, not including the initial reading of the problem instance, nor of writing the solution out to a file. Note that the solution description is a file of integers, one to a line, in ascending order corresponding to the input file, of the indices i in Σ. The recorded running time, however, included all steps in the program execution.

The output of each SET COVER program was verified by a separate checking program to be a cover.

The threshold τ used in the multiple pass greedy algorithm is 40, and was determined by trial and error.

We found that in practice our algorithm performed better if a threshold of p^{k-1}, rather than p^k, was used to determine whether a set in $\mathcal{S}^{(k)}$ should be added to the solution. The experimental results that are reported below use the p^{k-1} threshold. Note then that approximation bound for the algorithm, as tested, is $1 + p^2 \ln n$, rather than $1 + p \ln n$.

The maximum number of files for the $\mathcal{S}^{(k)}$, as used in our algorithm, allowed by the operating system was 250. When implemented in main memory, we allowed for up to 20,000 such subcollections. This limit on k imposes a constraint on the size of p: a very small p would require a very large number of files. Specifically, if κ is the upper bound on k, i.e., the maximum number of subcollections, then p must be at least $(\max |S_i|)^{1/\kappa}$ in order to ensure that $(\max |S_i|) = p^\kappa$.

4.3 Results

The results of our experiments are presented in Table 3 and Table 2. The algorithms used are as follows:

greedy The greedy algorithm using an inverted index, as described in Section 2.3.

multipass The multiple pass greedy algorithm, without an inverted index, as described in Section 2.3.

naive The naive algorithm mentioned in Section 2.1.

Disk-Friendly Greedy, aka DFG The algorithm presented in Section 3.2, the main contribution of this paper.

We first note that we do not show results for the external memory versions of the *greedy* and *multipass* algorithms for the largest problem instance. They required more than ten hours of computing time and seemed to place a very heavy

Table 1: Characteristics of the test problem instances.

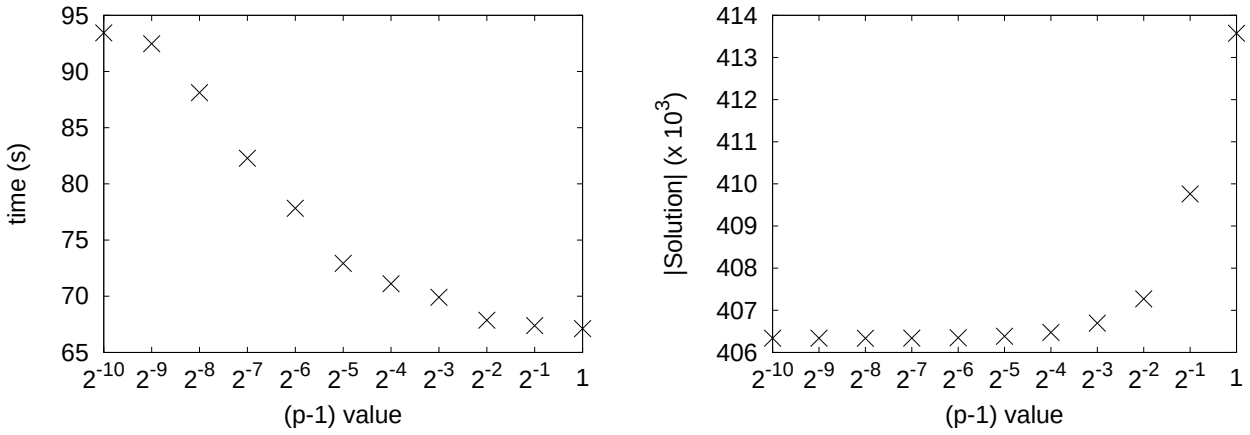
Name	$\|File\|$ *(Kb)*	*Items*	*Sets*	$\max\|T_j\|$	$\max\|S_i\|$
chess.dat	334	75	3196	3195	37
mushroom.dat	557	119	8124	8124	23
pumsbStar.dat	11027	7116	49046	38749	63
pumsb.dat	16298	7116	49046	48944	74
retail.dat	4069	16469	88162	50675	76
accidents.dat	34677	468	340183	340151	51
kosarak.dat	31278	41270	990002	601374	2498
webdocs.dat	1447158	5267656	1692082	1429525	71472

Table 2: Performance of the disk-based algorithms. The DFG algorithm was run with $p = 1.05$. For each dataset, the lowest running time is shown in *italics*, as is the smallest solution.

File	*Algorithm*	*RAM (Mb)*	*Disk (Mb)*	*Time (s)*	|Solution|
chess.dat	naive	0	1	*0.05*	26
	multipass	0	37	0.45	9
	greedy	0	3	0.37	*8*
	DFG	0	2	1.63	*8*
mushroom.dat	naive	0	2	*0.09*	44
	multipass	0	59	1.09	*22*
	greedy	0	5	0.64	25
	DFG	0	6	0.37	23
pumsbStar.dat	naive	1	37	*0.86*	1242
	multipass	1	830	7.30	752
	greedy	2	75	11.52	747
	DFG	0	66	1.08	*746*
pumsb.dat	naive	1	55	*1.86*	1317
	multipass	1	1217	7.66	*749*
	greedy	2	111	19.24	757
	DFG	0	98	2.07	751
retail.dat	naive	2	13	*0.82*	7153
	multipass	2	461	18.39	5103
	greedy	4	28	3.66	*5102*
	DFG	1	24	1.89	5111
accidents.dat	naive	8	175	4.64	245
	multipass	5	3772	49.24	*181*
	greedy	15	351	44.11	*181*
	DFG	5	329	*4.08*	182
kosarak.dat	naive	24	122	8.51	20664
	multipass	17	6571	331.66	*17746*
	greedy	44	247	98.66	17750
	DFG	17	164	*2.61*	17748
webdocs.dat	naive	206	4575	91.21	433412
	multipass				
	greedy				
	DFG	153	5075	*86.28*	*406440*

Table 3: Performance of the RAM-based algorithms. The DFG algorithm was run with $p = 1.001$. For each dataset, the lowest running time is shown in *italics* , as is the smallest solution.

File	*Algorithm*	*RAM (Mb)*	*Time (s)*	|Solution|
chess.dat	naive	1	0.05	26
	multipass	0	0.05	9
	greedy	1	0.07	*8*
	DFG	1	*0.04*	*8*
mushroom.dat	naive	3	0.07	44
	multipass	1	*0.06*	*22*
	greedy	3	0.07	25
	DFG	3	0.07	*22*
pumsbStar.dat	naive	39	*0.58*	1242
	multipass	19	0.64	752
	greedy	39	0.76	*747*
	DFG	27	0.59	753
pumsb.dat	naive	56	*0.83*	1317
	multipass	28	0.91	749
	greedy	57	1.03	757
	DFG	53	0.85	*735*
retail.dat	naive	16	*0.25*	7153
	multipass	8	0.37	5103
	greedy	17	0.32	*5102*
	DFG	11	0.26	5129
accidents.dat	naive	183	*2.27*	245
	multipass	93	2.74	*181*
	greedy	188	2.76	*181*
	DFG	112	2.37	*181*
kosarak.dat	naive	146	2.20	20664
	multipass	78	4.21	17746
	greedy	161	2.99	17750
	DFG	107	*1.97*	*17741*
webdocs.dat	naive	4722	100.98	433412
	multipass	2440	8049.08	406381
	greedy	4727	199.02	406351
	DFG	2827	*93.38*	*406338*



(a) Time cost as p varies

(b) Solution size as p varies

Figure 5: Effect of varying the p parameter in the RAM-based implementation of the DFG algorithm.

burden on the hard disk. This underlines our main point in this paper: even on modern hardware, what is currently considered a moderate- to large-size instance (about a million sets, totaling about 1.5GB) is simply beyond the reach of reasonable implementations of the traditional greedy algorithm.

Disk-based results. Our algorithm was primarily designed to handle external datasets, and here it shines. Table 2 shows the memory usage, maximum amount of disk used, running time of each algorithm, along with the size of the solution produced ($|\text{Solution}| = \sigma$). Except on the smallest instances, our algorithm is dramatically faster than previous greedy methods. The speedup increases as the instance size increases: ranging from about 10 times faster on medium instances (accidents.dat, pumsb.dat, pumsbStar.dat) to over 37 times faster on kosarak.dat. As noted above, webdocs.dat took a matter of minutes for our algorithm to process, whereas greedy had not completed after several hours of run time. Even on kosarak.dat, multipass was over a hundred times slower than DFG.

In-memory results. When there is sufficient memory to run the algorithms in memory, our proposed method compares very favorably to existing methods. Table 3 shows the results. We observe that in terms of time, there is relatively little to choose between the methods until we reach the larger instances. Arguably on the medium size instances (accidents.dat, kosarak.dat), the greedy algorithm is slower, while our algorithm is about as fast as naive. On the largest instance (webdocs.dat), the speed advantage is clearer: our algorithm is over twice as fast as greedy, and even faster than naive, while multipass is just impractical. In terms of quality, our algorithm is essentially equivalent to the greedy algorithm, improving in some cases, and never finding a solution with more than 0.5% more sets than greedy nor multipass. In contrast, naive is always worse, over 80% worse in one instance.

Impact of parameter p. To understand better the behavior of the algorithm as p is varied, we show further experiments in Figure 5 for the in-memory case (disk-based was quite similar). This shows that while the time cost increases as p is decreased, this is not so significant: increasing the accuracy guarantee by a factor of 1000 only adds 30s to the initial cost of around 60s. This is much better than worst case analysis, which predicts that for $p = 1 + 2^{-10}$, the time overhead is at most 32 times the overhead for $p = 1 + 2^{-5}$. Meanwhile, there is a clear benefit of decreasing p closer to 1: we see that the solution size decreases, although less dramatically so. Although hard to see in the figure, there is improvement as p decreases further, although minor: for the smallest values of p tried, the variation in solution size is only one or two sets out of 400,000.

Summary. Clearly, implementations of greedy do not scale to modern data sizes even on modern hardware; our technique appears to scale well across a variety of settings. Perhaps most surprising is that the disk-based results for DFG are very close to those of the memory-based ones—in particular, for the largest instance, our algorithm took about 90 seconds whether in memory or out (the lower duration in external memory is due to different choice of the parameter p). This, and the fact that DFG was slightly faster than the naive algorithm, indicates that the new algorithm is not I/O bound, i.e., is not waiting for disk before it can proceed.

In terms of the quality of the results, our algorithm remains almost identical to greedy, whether in memory (Table 3) or on disk (Table 2), as discussed above. Note that while greedy obtains the same solution, our algorithm was run with different parameters ($p = 1.001$ in memory, $p = 1.05$ on disk), and so obtains slightly different results. However, in both cases the results are close to those found by greedy, and better in some cases.

5. EXTENSIONS

Here, we outline some variations to consider for extensions of this work in the future.

Compact Representation of Covered Items. All our algorithms assume that it is feasible to store the set of items that have been covered so far in memory (for example, in hash table to allow quick search and update). When n, the size of the universe of items, is truly immense, it may be infeasible to store even this in memory exactly. Instead, a more compact representation is needed. If the universe is structured simply as the integers $1, 2, \ldots, n$, then a simple

bitmap index will suffice. More generally, when the items are larger and less well-structured (such as web page URLs), a Bloom filter can capture the set of covered items efficiently with a constant number of bits per universe item [3]. The data structure has a small false positive rate, meaning that the algorithm may leave a few items uncovered. Empirical analysis will be needed to determine the significance of this approximation.

Greedy on external memory. An additional heuristic to try is to attempt to combine the inverted index and multiple pass approaches on external memory. First, partition the sets into two files, based on whether $|S_i| > \tau$ or not. The file with the large sets would be traversed relatively efficiently because each block would have only a small number of sets and therefore would be read only a small number of times even if the sets were read in a somewhat arbitrary order. On the other hand, the blocks with sets of size at most τ would be read in a sequential manner, because the algorithm switches to a multiple pass approach. Choosing a good value of τ to coincide with block size could be done empirically. We should keep in mind, however, that the inverted index might be the dominant cost.

Reducing sets. Our DFG algorithm rewrites a set S_i to disk or memory in a "reduced" form, with the covered elements removed, if it is not immediately added to the solution. Applying this improvement would potentially be useful in other cases, such as the greedy algorithm (especially the multiple pass version), although this might not make enough of a difference to the general poor performance.

6. CONCLUDING REMARKS

Motivated by the increasing size of datasets compared the speed of external memory access, we studied the problem of efficiently finding set covers for large inputs. We observed that previous methods simply do not scale well, but that a simple algorithm has guaranteed performance and scales very gracefully to very large instances. It has best-in-class speed, and best-in-class solution quality, a combination achieved by none of the other algorithms.

A clear next step is to ask the same questions of other common large scale mining and optimization scenarios. In particular, many modern datasets are represented by large graphs with millions of nodes and edges (such as the web graph, social networks, communication graphs). It is natural to ask how to effectively find good quality covers, partitions and dominating sets within such large, disk-resident, graphs.

7. ACKNOWLEDGMENTS

Anthony Wirth's visit to AT&T Labs–Research was supported by the Australian Research Council. We thank David Johnson for helpful comments and conversations.

8. REFERENCES

[1] B. Berger, J. Rompel, and P. Shor. Efficient NC algorithms for set cover with applications to learning and geometry. *Journal of Computer and System Sciences*, 49(3):454–77, 1994.

[2] 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*, pages 254–60, 1999.

[3] A. Broder and M. Mitzenmacher. Survey: Network applications of bloom filters: A survey. *Internet Mathematics*, 1(4), 2003.

[4] F. Chierichetti, R. Kumar, and A. Tomkins. Max-Cover in Map-Reduce. In *Proceedings of the 19th International Conference on World Wide Web*, pages 231–40. ACM, 2010.

[5] U. Feige. A threshold of $\ln n$ for approximating set cover. *Journal of the ACM*, 45(4):634–52, 1998.

[6] 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*, page 18pp, January 2003.

[7] B. Goethals. Frequent itemset mining dataset repository. `http://fimi.cs.helsinki.fi/data/`.

[8] L. Golab, H. Karloff, F. Korn, D. Srivastava, and B. Yu. On generating near-optimal tableaux for conditional functional dependencies. In *VLDB*, 2008.

[9] F. Gomes, C. Meneses, P. Pardalos, and G. Viana. Experimental analysis of approximation algorithms for the vertex cover and set covering problems. *Computers & Operations Research*, 33(12):3520–34, 2006.

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

[11] D. Johnson. Approximation algorithms for combinatorial problems. *Journal of Computer and System Sciences*, 9(3):256–78, 1974.

[12] C. Lucchese, S. Orlando, R. Perego, and F. Silvestri. Webdocs: a real-life huge transactional dataset.

[13] M. Mihail. Set cover with requirements and costs evolving over time. In *Proceedings of the Second International Workshop on Approximation Algorithms for Combinatorial Optimization Problems: RANDOM-APPROX*, pages 63–72, 1999.

[14] K. Munagala, S. Babu, R. Motwani, and J. Widom. The pipelined set cover problem. Technical Report 2003-65, Stanford InfoLab, October 2003.

[15] B. Saha and L. Getoor. On Maximum Coverage in the streaming model & application to multi-topic blog-watch. In *2009 SIAM International Conference on Data Mining (SDM09)*, April 2009.