Lazy and Eager Approaches for the Set Cover Problem

Ching Lih Lim **Alistair Moffat** **Anthony Wirth**

Department of Computing and Information Systems
The University of Melbourne
Victoria 3010, Australia

Abstract

The SET COVER problem is tantalizingly simple to describe: given a collection F of sets, each containing a subset of a universe U of objects, find a smallest sub-collection A of F such that every object in U is included in at least one of the sets in A. However, like many such combinatorial problems, SET COVER is NP-hard, meaning that it is unlikely that an efficient algorithm will be found, and that approximation algorithms must be preferred for non-trivial problem instances. One well-known approximation approach for SET COVER is to repeatedly add the set with the most uncovered items to the solution, continuing until every element in the universe is covered; this GREEDY approach has a provable logarithmic approximation ratio, essentially the best feasible ratio. Here we study the implementation of the GREEDY approach to SET COVER, evaluating eager and lazy versions and other implementation options. Experiments with several large datasets demonstrate that lazy "as required" priority queue updates should be preferred, rather than eager "as soon as possible" ones; and that when implemented in this way, the GREEDY mechanism can solve some large instances of the SET COVER problem very quickly. This practical superiority contrasts with the lazy version's having a demonstrably higher worst-case operation cost.

Keywords: set cover, priority queue, lazy update, approximation algorithm.

1 Introduction

The SET COVER problem arises in a very broad range of logistic and layout problems, including monthly beer production, the roll-out schedule of a network, and data mining of a collection of sequenced DNA data for simple repeats [11, 12]. An instance of SET COVER consists of a universe U of objects, and a collection F of subsets of U. The challenge is to identify a smallest sub-collection of F subject to the constraint that the union of the sets in A must be equal to the union of the sets in F; that is, A must cover the universe [3, 4, 12].

For example, suppose that the city government wishes to select a set of locations for fire stations so that every home is less than 5 km by road from at least one fire station, and so that the number of fire stations required is minimized. In this scenario, the set of homes forms the universe U of items that must be covered by any solution; and the ith possible location for a fire station gives rise to a set S_i of homes that are within 5 km of it. Assuming that the total pool of possible locations is sufficiently numerous that every home is within 5 km of one or more of the locations under consideration (equivalently: that $\bigcup_{S\in F} S = U$), the required SET COVER solution is a smallest set of locations A such that $\bigcup_{S\in A} S$ is also equal to U. That is, what is required is a smallest subset $A \subseteq F$ such that every home in U appears in at least one element in A. If there is more than one minimal-size solution, secondary criteria could be used to choose between them. Other SET COVER scenarios include information retrieval, where a query over words (elements) computes the smallest sub-collection of documents, drawn from a large collection of documents, that as a whole contain all of the query words.

Like many such combinatorial problems, SET COVER is NP-hard. Consequently, to find optimal solutions, some exhaustive search component is required; hence, only trivially small problem instances can be solved exactly within reasonable resource bounds. Approximation techniques must then be used for large-scale problem instances. Any approximation technique is a compromise between solution *effectiveness* and implementation *efficiency*, with high effectiveness usually only possible at the cost of low efficiency. At one extreme in this spectrum, exhaustive methods generate optimally effective solutions, but typically require time that is exponential in the size of the problem instance; at the other extreme, taking (for the SET COVER problem) $A = F$ certainly guarantees coverage and is very fast to compute, but might be ineffective by a very large multiplicative factor. Between these extremes, a good approximation algorithm balances a guarantee on effectiveness (for example, in the case of SET COVER, through a proof that the solution generated by the approximation algorithm is at most some bounded factor larger than the size of an optimal solution) with desirable asymptotic execution analysis. For most NP-hard problems, the latter requirement corresponds to running time that is polynomial in the size of the problem instance.

The GREEDY algorithm for SET COVER, introduced in detail in Section 2, can be measured against these criteria. The solutions it constructs are bounded relative to the size

of a corresponding optimal solution [6, 10], and it executes in polynomial time. It remains an important focus of study because it has provably the best approximation factor possible in polynomial time, is known to perform very well in practice across multiple kinds of instances and often generates solutions that are within 10% of optimum [7, 8]. Recent GREEDY variants include a disk-based version [4], and fast parallel algorithms [2, 3]; these options add to the versatility and usefulness of the approach.

The GREEDY algorithm proceeds by repeatedly adding to the solution the remaining set with the largest number of uncovered elements, seeking to obtain the greatest gain for each additional set that is used. To find the set with the greatest number of uncovered elements a priority queue data structure is employed to track the size of every candidate set. That structure is one of the focuses of this paper, as we consider in detail how to implement the GREEDY approach, and balance the costs of different types of queue update operations. One obvious option is to use a binary heap, but other – simpler – structures can also be used. A critical issue that affects the choice is the rate at which updates are performed – as each set is added to the growing solution, a large number of the other remaining sets might need to have their "uncovered element" counts decreased. In this EAGER-GREEDY implementation, the priority queue must always be able to quickly emit the next set to be added to the solution, meaning that many updates to set sizes might be required at each iteration.

To counter this possible inefficiency, we explore the use of lazy update options, in which changes to sets are deferred as long as possible and until they are actually necessary. The resultant implementation, LAZY-GREEDY, performs a different balance of elementary operations, and executes more quickly than the EAGER-GREEDY version on public datasets. Our experimental results indicate that LAZY-GREEDY outperforms the eager greedy method in terms of running time, and is also competitive against a recently described greedy bucket-based algorithm [4].

The notion of lazy evaluation can be further applied to the process of updating a set, that is, identifying in it (and removing from it) covered items. We explore this option too, terminating those operations as soon as it can be known that the set in question will not be the next one added to the solution.

The remainder of the paper is structured as follows. Section 2 explains the technical background of the SET COVER problem; shows how the GREEDY strategy computes solutions with a guaranteed approximation bound; and describes the EAGER-GREEDY implementation of that approach. Section 3 focuses on the bottleneck issue with EAGER-GREEDY, and describes an alternative lazy-update version of the same GREEDY paradigm. We also describe an input pattern that gives rise to a worst-case number of set updates. Section 4 describes the experiment environment and datasets used to evaluate the pool of SET COVER algorithms; and then demonstrates that the new lazy approach solves large instances of the problem significantly faster than does the eager implementation of the same procedure. The lazy implementation is also competitive against the more complex DF-GREEDY bucket-based implementation of the eager greedy procedure developed by Cormode et al. [4]. Extending the notion of "laziness" to the set difference operator is considered in Section 5. Section 6 then concludes our presentation.

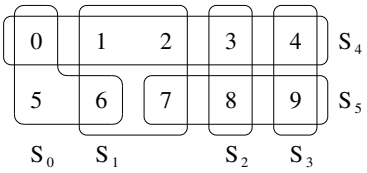

Figure 1: An instance of the SET COVER problem over $U = [0..9]$, with $F = \{S_0, \ldots, S_5\}$, and $S_0 = \{0, 5, 6\}$, $S_1 = \{1, 2, 6, 7\}$, $S_2 = \{3, 8\}$, $S_3 = \{4, 9\}$, $S_4 = \{0, 1, 2, 3, 4\}$, and $S_5 = \{7, 8, 9\}$.

2 The SET COVER Problem

We now state the SET COVER problem, and describe the best-known approximation algorithm for solving it.

2.1 Definition

The universe of objects under consideration is supposed, without loss of generality, to consist of the n integers $[0..n-1]$, where, as a general notation, $[0..z-1]$ stands for the set $\{0, 1, \ldots, z-1\}$. An instance of the SET COVER problem relative to the universe $U = [0..n-1]$ is given by a collection F of m sets $\{S_0, \ldots, S_{m-1}\}$, with $S_i \subseteq U$, and $\bigcup_{i \in [0..m-1]} S_i = U$. That is, it is known that every item in U appears in at least one of the sets S_i in F.

The challenge then posed is to find a smallest sub-collection $A \subseteq F$ such that $\bigcup_{S \in A} S = \bigcup_{S \in F} S = U$ [12]. The *size* of the solution is given by $|A|$. Another way of categorizing the cover size that we do not pursue in this paper is the total number of elements in the sets making up A, that is, the objective is to minimize $\sum_{S \in A} |S|$ subject to the constraint $\bigcup_{S \in A} S = U$. It is convenient to describe the set A in terms of the *indices* of the sets that are being included in the solution. The interpretation of A to be used in a particular situation will always be clear from the context.

Figure 1 illustrates a simple instance of the SET COVER problem in which $U = [0..9]$ and $F = \{S_0, \ldots, S_5\}$. It is also helpful to define a value M as the total of the m sets that are involved in a SET COVER instance, $M = \sum_{i \in [0..m-1]} |S_i|$. In the example, $M = 19$, and is a measure of the size of the input required when describing the instance. The smallest solution for the instance in Figure 1 is $A = \{0, 4, 5\}$, since these three sets cover U, and there is no smaller sub-collection of F that also covers U. The alternative sub-collection $\{S_0, S_1, S_2, S_3\}$ also covers U and is free of redundancy (none of the sets can be removed without coverage being lost), but is not optimal.

The SET COVER problem is NP-hard [12]. Indeed, Feige [6] showed that, unless NP has "slightly super-polynomial time algorithms", SET COVER cannot be approximated within a factor of $(1 - o(1)) \ln n$. The next section presents the GREEDY approach, whose approximation factor is very close to this bound.

2.2 The GREEDY Strategy

Algorithm 1 provides an overview of the GREEDY approximation mechanism for SET COVER. The idea behind it is very simple: at each iteration of the loop, the set with the largest number of uncovered items is identified, and added to a growing solution A. That process is continued until every element in U is covered; the precondition

Algorithm 1 The GREEDY approach

Input: Family F of m sets $S_i \subseteq U = [0..n-1]$, and with $\bigcup_{i \in [0..m-1]} S_i = U$

Output: Set A of indices of sets with $\bigcup_{i \in A} S_i = U$.

```
1: A ← {}
2: covered ← {}
3: while covered ≠ U do
4:     i ← argmax_{j∈[0..m−1]}{|S_j − covered|}
5:     A ← A ∪ {i}
6:     covered ← covered ∪ S_i
7: end while
8: return A
```

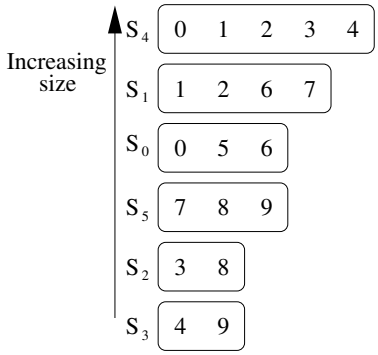

Figure 2: The six sets of the SET COVER instance shown in Figure 1, sorted into increasing size order. The first set to be considered is S_4.

that $\bigcup_{i \in [0..m-1]} S_i = U$ guarantees that the termination condition for the loop will be met.

Figure 2 shows the initial state of Algorithm 1, as applied to the problem instance shown in Figure 1. The GREEDY approach first adds set S_4 to the solution, and in doing so, covers items $\{0, 1, 2, 3, 4\}$. The second set considered is then S_5, since it (now) has three uncovered items, more than the larger set S_1, which only has two uncovered items. Once S_5 is added to the solution A, there are only two uncovered items, 5 and 6, and S_0 covers both of them, whereas S_1 only covers one of them. With S_0 included in the solution, the main loop terminates; in this case, the optimal solution is identified.

Johnson [10] demonstrated that the GREEDY approach constructs a solution that is not more than $1 + \ln n$ times larger than the minimal one; improved bounds were provided by Slavík [13]. Young [14] and Dutta [5] provide surveys of this development. The GREEDY approach also performs well in practice, typically identifying solutions that are close to optimal [3, 4]. Extensions include parallel implementations designed for multi-processor architectures [2]; for disk-based operation [4]; enhancements that improve the worst-case approximation ratio [9]; and methods for on-line SET COVER problems [1].

2.3 Implementing GREEDY: Eager Evaluation

Algorithm 1 leaves many details unspecified. Critical decisions that are required include:

- How to represent the sets S_i;
- How to manage the collection of sets so that the computation implied by the *argmax* evaluation at step 4 can be carried out; and
- How to represent the set *covered* so that the computations at steps 4 and 6 can be carried out.

Algorithm 2 The EAGER-GREEDY implementation

Input: As for Algorithm 1

Output: As for Algorithm 1

```
1: A ← {}
2: covered ← {}
3: pqueue ← {}
4: for j ← 0 to m − 1 do
5:     pqueue.insert(⟨|S_j|, j⟩)
6:     updates[j] ← {}
7: end for
8: while covered ≠ U do
9:     ⟨uc, i⟩ ← pqueue.maximum()
10:    A ← A ∪ {i}
11:    for c ∈ S_i − covered do
12:        covered ← covered ∪ {c}
13:        for each set S_j that contains c do
14:            updates[j] ← updates[j] ∪ {c}
15:        end for
16:    end for
17:    for all j for which updates[j] ≠ ∅ do
18:        ⟨uc, j⟩ ← pqueue.delete(j)
19:        S_j ← S_j − updates[j]
20:        pqueue.insert(⟨uc − |updates[j]|, j⟩)
21:        updates[j] ← {}
22:    end for
23: end while
24: return A
```

Algorithm 2 shows one way in which these issues can be resolved. An explicit priority queue is introduced, with the usual operations assumed: *insert*(), to add a tuple $\langle uc, i \rangle$ consisting of a set label i and queue weight *uc* (steps 5 and 20); *maximum*(), to remove and return the tuple with the largest weight *uc* (step 9); and *delete*(j), to locate and remove the tuple with the specified label j (step 18). As well, details are given of the process to be followed when a set S_i is added to the solution. The loop at step 11 iterates over the newly covered items in S_i, and removes each of them from the every other set S_j in which it appears (step 13). Each such removal decrements the "uncovered symbol count" *uc* of a set S_j, but these changes are all deferred until they can be processed in a batch at step 18, which locates that set in the priority queue, extracts its current *uc* value, and then, at step 20, puts it back in the priority queue with a reduced *uc* value. The batching process ensures that each S_j that shares items with S_i is updated just once per occurrence of step 9.

Two further details require elaboration. First, in order to identify at step 11 the items c that are being newly covered, it is necessary for each element in S_i to be checked for membership in *covered*. The most appropriate mechanism is for *covered* to be stored as an indexed bitvector, with *covered*[c] set to 1 if and only if $c \in$ *covered*. Each lookup to *covered*[c] requires $O(1)$ time; over all sets S_i, the cost is at most $\sum_{i \in [0..m-1]} |S_i| = M$, that is, is linear in the size of the input description. Second, step 13 requires knowledge of the collection of sets S_j that contain a given element c. An efficient way of supporting this need is to pre-compute an inverted index over the sets, so that a mapping is available from items to sets containing them [4]. Building an inverted index requires $O(M)$ time using distribution-sorting techniques, plus a corresponding amount of memory space to store it.

2.4 Priority Queue

With structures for *covered* and for identifying affected sets in place, the overall time required by EAGER-GREEDY is determined by the choice of priority queue structure: there are at most $|F| = m$ priority queue *maximum*() operations required at step 9; and at most M decrease-weight operations required at steps 18–20. If the priority queue is implemented as a binary heap each operation requires $O(\log m)$ time (since there are at most m sets in the heap); and the large number of update operations dominates the running time, giving rise to an $O(M \log m)$ overall cost.

An obvious question is whether a binary heap is in fact the best choice. When a large number of decrease-weight operations must be balanced against a smaller number of extract-maximum operations, other options are also possible. The fact that all of the item weights manipulated in the priority queue are integers between 0 and $n - 1$ also adds flexibility.

In particular, a simple array-plus-lists structure can be used as a priority queue. An array *pqueue* of n elements is maintained, with *pqueue*[*uc*] a list of the indices of the sets that currently have *uc* uncovered elements. A variable *pqueue.top* indicates the largest index in *pqueue* that is non-null; and after every extract-maximum operation, *pqueue.top* is updated via a sequential scan through *pqueue* looking for the next non-null entry. To carry out the decrease-weight operation at steps 18–20, the current entry in *pqueue* for set j is identified via a table indexed by set identifier; and that node is deleted from its current list *pqueue*[*uc*], then re-linked in to *pqueue*[*uc* $-$ |*updates*[j]|] where |*updates*[j]| is the number of changes being made to S_j. This takes $O(1)$ time per set moved. In addition, an auxiliary stack records the values of j for which *updates*[j] is no-empty, to that the loops at step 17 does not introduce unnecessary overhead.

Allowing for the fact that the total cost of all of the scanning required as part of the extract-maximum operations is $O(n)$, the execution time for EAGER-GREEDY is now $O(n + M)$. Moreover, since the collection F covers U, it must be that $M \geq n$. Hence, with this queue structure, the EAGER-GREEDY approach described in Algorithm 2 executes in $O(M)$ time.

3 A Lazy Implementation

Section 2.4 describes one way in which the small number of extract-maximum operations can be balanced against a much larger number of decrease-weight operations. But a significant imbalance remains between the number of extract-maximum operations and the number of decrease-weight operations, and it is interesting to ask if the overall envelope of operations can be decreased by explicitly trading a large drop in the number of decrease-weight operations for a smaller increase in the number of extract-maximum operations.

3.1 Deferred Cleaning

In Algorithm 2, each execution of step 9 returns the next largest set, according to the number of uncovered elements. Then, as that set is added to the solution, every other set that contains common uncovered elements is adjusted. Those adjustments ensure that the counts of uncovered elements for all sets are correct at all times.

Algorithm 3 The LAZY-GREEDY implementation

Input: As for Algorithm 1
Output: As for Algorithm 1

```
 1: A ← {}
 2: covered ← {}
 3: pqueue ← {}
 4: for j ← 0 to m − 1 do
 5:     pqueue.insert(⟨|S_j|, j⟩)
 6: end for
 7: while covered ≠ U do
 8:     ⟨uc, i⟩ ← pqueue.maximum()
 9:     S'_i ← S_i − covered
10:     if |S'_i| = |S_i| then      // all items in S_i were clean
11:         A ← A ∪ {i}
12:         covered ← covered ∪ S_i
13:     else                        // some items in S_i were covered
14:         S_i ← S'_i
15:         pqueue.insert(⟨|S_i|, i⟩)
16:     end if
17: end while
18: return A
```

If those counts were not reduced so relentlessly, the putative sizes of the competing sets – the weights manipulated by the priority queue – would deviate from their true values. But they would always be *upper* bounds, and the worst that could happen is that a set might be emitted by the priority queue as being the next largest, only for a subsequent check to find that in fact it contained covered elements that had not yet been noted, and could not be included as part of the greedy solution just yet.

Algorithm 3 presents the revised process. Step 8 accesses the largest set in the priority queue, denoted as S_i. In recognition of the fact that S_i might include one or more items covered in previous iterations, the first processing undertaken is a set difference $S_i - \mathit{covered}$, in which each element of S_i is checked against the set *covered*. If the set difference operation removes no elements from S_i, then all elements were uncovered, and it can be added to the greedy solution (steps 11–12).

But if there were covered elements present in S_i, it cannot be added to the solution, since it might not be the set containing the largest number of uncovered items – there might be another smaller set that has that claim. Instead, the cleaned set S'_i is retained, and is reinserted into the priority queue (step 15). The main loop then iterates, and the newly largest set is chosen as the next maximum. This process continues until a clean set containing only uncovered items emerges from the priority queue.

With the revised arrangement, the while loop at step 7 executes more than $|A|$ times. But each reinsertion operation at step 15 should move the set S_i down by multiple slots in the priority queue ordering; that is, there might be fewer reinsertions in total, decreasing the total number of queue operations required.

3.2 Example

Consider the example instance shown in Figure 1. The initial state of the priority queue is as shown in Figure 2. The largest set, S_4, is removed, and because no items have been covered yet, all of its elements are clean, and S_4 is added to the solution. The next largest set is S_1, with a putative *uc* of 4. But when S_1 is checked for cleanliness, it is discovered that two of its elements (1 and 2) have become covered, and its uncovered size is actually only two. So a cleaned set $S'_1 = \{6, 7\}$ is added back into the

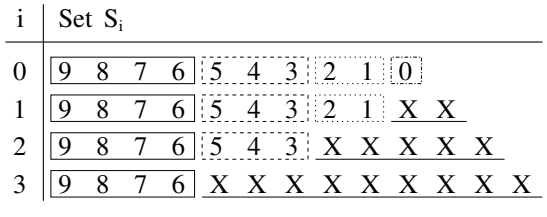

Figure 3: Pathological example with $m = 4$ sets. Items shown as "X" are unique, and appear one time only.

queue, with $uc = 2$; nothing is added to the solution at this loop iteration. Now the next largest set is S_0, but it is also discovered to contain covered items, and is returned to the queue as $S'_0 = \{5, 6\}$. Next, S_5 is emitted from the queue, and it is clean – there are no covered items. So S_5 is added to the solution, and elements 7, 8, and 9 are marked as having been covered.

By this stage, the queue contains S_2, S_3, S'_1, and S'_0, all of size two. Assuming that they are considered in that order, S_2 is found to contain no uncovered elements; then S_3 is likewise found to be of no use; and then S'_1 is reduced to $S''_1 = \{6\}$ and is returned to the queue.

Now the queue contains $S'_0 = \{5, 6\}$ and $S''_1 = \{6\}$. The processing of S'_0 reveals that both of its items are as yet uncovered, and so S_0 is added to the solution A. That addition results in all items being covered, and so the final solution is $A = \{4, 5, 0\}$, the same solution as the EAGER-GREEDY approach.

In this particular example, a total of 8 extract-maximum operations are required compared to just 3 by EAGER-GREEDY. The anticipated payoff for that growth is a reduced number of check-covered operations; that gain does not arise in the example because of its small scale, but should be measurable on larger instances.

3.3 Worst-Case Performance

The change to lazy evaluation means that the number of iterations of step 8 is no longer bounded by m, the number of sets input to the problem. This observation gives rise to an interesting question: What is the maximum number of iterations possible, that is, is there a limit on the number of times that a given set can be reinserted at step 15 before it emerges "clean" and able to be added to the solution?

Consider the set structure shown in Figure 3. Each set contains a hierarchy of items present in other sets in the collection, marked by the various boxes; plus additional unique filler elements denoted "X", whose role is to force the ordering of the queue operations. In a problem instance over m sets S_0 to S_{m-1} there are thus $m(m+1)/2$ different items that appear as part of the hierarchical pattern. In addition, set S_i, contains $i(i+3)/2$ unique filler elements, meaning that the total filler count is

$$\sum_{i=0}^{m-1} \frac{i(i+3)}{2} = \frac{m^3 + 3m^2 - 4m}{6} .$$

Hence, in total, the universe size is given by $n = (m(m+1)/2) + (m^3 + 3m^2 - 4m)/6 = (m^3 + 6m^2 - m)/6$. A similar computation shows that the input size is given by

$$M = \frac{m^3 + 2m^2 - m}{2} \approx \frac{m^3}{2} .$$

When processed by LAZY-GREEDY, the input arrangement shown in Figure 3 gives rise to a repetitive cycle of computations. It first processes set S_{m-1}, finds that all of its elements are clean, and hence covers them. It then cycles through S_{m-2}, S_{m-3} and so on down to S_0, cleaning the same group of m elements out of each. Once all $m-1$ remaining sets are clean, set S_{m-2} returns to the head of the queue in a fully uncovered state, and the cycle is repeated recursively, exactly as if a pathological instance of $m-1$ sets had been input.

Since set S_i cycles through the head of the priority queue a total of $m-i$ times, the total number of executions of step 8 in Algorithm 3 is

$$\sum_{i=0}^{m-1} (m-i) = \frac{m(m+1)}{2} \approx \frac{m^2}{2} .$$

Hence, the number of extract-maximum operations performed on the pathological input instances is $\Theta(M^{2/3})$, and remains sub-linear in the size of the input description. That asymptotic bound also applies to the number of times that step 15 is executed.

The other place in the LAZY-GREEDY approach where the number of iterations might be different from the EAGER-GREEDY approach is at step 9. Each time a set S_i reaches the head of the queue, every element c in it is checked against *covered*, taking $O(1)$ time per element. Elements that have already been covered are removed as they are discovered, and only the uncovered elements are retained in the set S'_i that is reinserted in to the queue. But the elements that were not covered will be checked again when S_i reaches the head of the queue next time, and it is no longer possible to bound the number of "check covered" operations by M. Indeed, for the pathological input structure that is illustrated in Figure 3, the total number of check-covered operations for an instance of size m, is given by the summed size of the (recursive) problem instances from size m down to size 1:

$$\sum_{i=1}^{m} \frac{i^3 + 2i^2 - i}{2} = \Theta(m^4) .$$

The behavior of the pathological input arrangement can be summarized thus:

Observation 1. *There is a* SET COVER *problem instance of size M that requires the* LAZY-GREEDY *method to spend $\Theta(M^{4/3})$ time, including $\Theta(M^{2/3})$ extract-maximum operations in the priority queue.*

Note that this time bound is asymptotically greater than the $O(M)$ time that is achieved by the EAGER-GREEDY implementation. Fortunately, typical problem instances do not display the particular structure required to force this bad behavior, and when executed on typical instances, the LAZY-GREEDY approach shows substantial benefits compared to the EAGER-GREEDY implementation.

4 Experimental Results

We now demonstrate the difference that lazy evaluation can make on typical SET COVER problem instances.

4.1 Datasets and Experiment Environment

Three datasets are used in the experiments – RETAIL, ACCIDENTS and WEBDOCS – taken from the Frequent Item-

Dataset	Variable	RETAIL	ACCIDENTS	WEBDOCS
Universe size	n	16,470	468	5,267,656
Number of sets	m	88,162	340,183	1,692,082
Largest set size		76	51	71,472
Median set size		8	34	98
Average set size		10.31	33.81	177.23
Total input size	M	908,576	11,500,870	299,887,139

Table 1: The properties of the datasets involved in our experiments.

Set Mining dataset repository[1]. The RETAIL dataset represents anonymized data on shopping items purchased by customers in Belgium, with a set-cover solution being a minimal set of customer transactions such that every item stocked in the supermarket is purchased at least once; ACCIDENTS is a set of traffic accident records in a certain time period also in Belgium, with a set-cover representing a subset of them in which every one of the recorded variables occurs at least once; and WEBDOCS represents a crawled collection of Web html documents, with a set-cover solution being a smallest set of documents such that every word that appears in any of the documents appears in at least one of the documents in the subset. Each line in the datasets is a space-separated sequence of integers. Table 1 describes the properties of the datasets.

Experiments were conducted on a server running x86-64 Red Hat Enterprise Linux ES Release 4 (Nahant Update 7) with four 3.2 GHz Intel Xeon processors and 8 GB of primary memory, with programs implemented using the 64-bit version 1.5.0_08 of the Java Runtime Environment. The execution times presented below are all the average of ten runs of the corresponding scenario, in all cases starting with a configuration in which the set data structure is available in memory.

4.2 Implementations

Java implementations of EAGER-GREEDY and LAZY-GREEDY were prepared. All of the implementation executions were restricted to the usage of a single thread by default and had the runtime JIT (Just-In-Time) optimization enabled by default. All data structures were presumed to fit in to main memory. In the case of the EAGER-GREEDY implementation, the first operation carried out was to construct the index that is required, with that time included in the reported execution times.

The internal representations for the input sets and for *covered* were carefully chosen. A Boolean bit-vector of $\lceil n/32 \rceil$ four-byte integers represents the set *covered*, allowing constant-time check-covered operations. Each of the sets was stored in an allocated segment out of an array of M integers, with covered elements rotated to the end of each set's zone, and uncovered elements retained at the front of each zone. There was no resizing or de-fragging the underlying array.

The priority queue *pqueue* for tracking the set sizes was implemented as an array of size n, in which each bucket was indexed by *uc*, and contained the identifier of the first set of that size, with other sets threaded from that first one. In the case of the EAGER-GREEDY mechanism, the priority queue lists were doubly-linked, and contained three fields: a *next* pointer, a *prev* pointer, and the *uc* value for that set, the latter being needed to allow the *pqueue.delete*(j) operation to be efficient (step 18 in Algorithm 2). Double threading of list nodes was required, because the nodes being deleted might be anywhere in their list. Access to the j th set, required at the same step, is achieved in $O(1)$ time by simply storing the set of priority queue nodes in a single large array (rather than as independently malloc'ed objects), with S_j stored in the j th array element. Each list was implemented as last-in first-out structure, with insertions always at the head.

Structure	EAGER-GREEDY	LAZY-GREEDY
covered	$n/32 + O(1)$	$n/32 + O(1)$
pqueue	$n + 3m + O(1)$	$n + m + O(1)$
sets	$m + M + O(1)$	$m + M + O(1)$
index	$n + M + O(1)$	—

Table 2: Space cost, in integers.

In the LAZY-GREEDY approach a similar priority queue is maintained, but because deletions only occur as extract-maximum operations at the front of each list, no *uc* value is required, and single-threading suffices.

These various considerations lead to the space costs shown in Table 2. For the file WEBDOCS, the total nominal space requirement for the EAGER-GREEDY implementation was thus 2,355 MB; with 1,178 MB required for the LAZY-GREEDY implementation. That is, the index that is required for the EAGER-GREEDY approach essentially doubles the amount of memory space required. All of the values are stored as integer indices into either *pqueue* or into the array of sets. Naturally, if any of n, m, or M is greater than 2^{32}, then eight-byte integers would be required rather than the four-byte ones assumed in these calculations.

4.3 Worst-Case Behavior

Table 3 shows execution times and operation counts for the EAGER-GREEDY and LAZY-GREEDY implementations when applied to the extreme problem instances described in Section 3.3. As anticipated by Observation 1, when m doubles, the number of extract-maximum operations required by LAZY-GREEDY increases by a factor of four, and both the number of check-covered operations required and the measured running time increase by a factor of sixteen. As m doubles, the input size M increases by a factor of roughly eight. For SET COVER instances that have this structure, the extra space required by the EAGER-GREEDY approach is a sound investment, and prevents super-linear (in M) execution times.

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

m	M	time (s)		extract-maximum		check-covered	
		E-G	L-G	E-G	L-G	E-G	L-G
200	4,039,900	0.37	1.17	200	20,100	4,039,900	204,681,650
300	13,589,850	1.28	5.65	300	45,150	13,589,850	1,028,283,725
400	32,159,800	2.98	17.47	400	80,200	32,159,800	3,237,393,300
500	62,749,750	5.72	39.92	500	125,250	62,749,750	7,885,510,375
600	108,359,700	9.88	85.36	600	180,300	108,359,700	16,326,134,950

Table 3: Execution time and operation counts LAZY-GREEDY and EAGER-GREEDY on pathological input sequences.

Measurement	Dataset	EAGER-GREEDY	LAZY-GREEDY	DF-GREEDY
Solution size	RETAIL	**5,106**	5,113	5,119
	ACCIDENTS	181	**180**	185
	WEBDOCS	**406,372**	406,400	406,428
Extract-maximum	RETAIL	**5,106**	162,961	—
	ACCIDENTS	**181**	1,080,524	—
	WEBDOCS	**406,372**	3,291,585	—
Check-covered	RETAIL	**92,540**	1,239,924	1,223,062
	ACCIDENTS	**6,630**	18,170,446	17,766,461
	WEBDOCS	**133,008,957**	335,085,502	326,336,293
Queue reinsertions	RETAIL	778,704	74,817	**74,009**
	ACCIDENTS	2,365,299	743,400	**709,066**
	WEBDOCS	46,092,276	1,599,610	**1,562,286**
Running time (s)	RETAIL	0.39	**0.18**	**0.18**
	ACCIDENTS	1.53	0.85	**0.84**
	WEBDOCS	73.10	5.34	**5.11**

Table 4: Performance of EAGER-GREEDY, LAZY-GREEDY and DF-GREEDY on three datasets. The DF-GREEDY method is measured using parameter $p = 1.1$. The best value in each row is highlighted.

4.4 Practical Applications

Table 4 shows what happens on the three problem instances described in Table 1. The final column is discussed in Section 4.5. As currently implemented, the time required to read each file (text format) is 1.17 sec, 4.82 sec, and 111.24 sec, for RETAIL, ACCIDENTS and WEBDOCS, respectively. These quantities are *not* included in the running times in Table 4.

In the first block of values, the solution sizes $|A|$ are listed. The variation between EAGER-GREEDY and LAZY-GREEDY, and the absence of a single "right" answer, is a consequence of differences in the way that equal-sized sets are handled. If a secondary key (such as set number), was introduced to the *pqueue* ordering, the implementations could be made to give the same answer. The drawback of doing that would be that queue reinsertions could no longer be made at the head of the list *pqueue*[*uc*], and would potentially add to the execution time. Nor could there be any expectation that the single answer that would be generated through the use of a tie-breaking rule would always be the smaller of the two alternatives. Hence the retention of the current arrangement, which by chance alone happens to favor the EAGER-GREEDY implementation on two of the three datasets.

The next three blocks in Table 4 give detailed operation counts for extract-maximum, check-covered, and queue-reinsert operations for the three datasets. As anticipated, the EAGER-GREEDY implementation performs a minimal number of extract-maximum operations, just one per set added to the solution A, whose size is at most m. But there is a very large number of queue reinsertion operations performed, close to M, and even with the array-based priority queue described in Section 2.4, each reinsertion is relatively costly, involving multiple tests for boundary cases and multiple pointer assignments.

Compared to EAGER-GREEDY, the LAZY-GREEDY implementation reduces the number of reinsertion operations by a large or (on WEBDOCS) very large factor. The tradeoff is that the number of extract-maximum and check-covered operations increases. But individual check-covered operations are fast (just an array access and a mask operation), and even extract-maximum operations are less costly than reinsertions.

The last block in Table 4 shows the measured execution time, including the cost of index construction in the case of EAGER-GREEDY. On the three test datasets LAZY-GREEDY does indeed execute more quickly, by factors that vary from 2.2 (RETAIL) to 13.7 (WEBDOCS).

4.5 Disk-Friendly Greedy

Cormode et al. [4] have also considered the question of how to implement the GREEDY method for SET COVER. Their DF-GREEDY (disk-friendly greedy) approach is designed to minimize the number of non-sequential operations, and maximize the number of sequential operations over the data, and hence provides good performance when the input data is so voluminous that it cannot be stored in main memory. Cormode et al. introduce a fidelity param-

eter $p > 1$ that permits further imprecision of the output size, in order to control the number of disk seek operations required. That is, the DF-GREEDY approach can be thought of as being an "approximate-approximation" mechanism, with a slightly weakened performance guarantee of $1 + p \ln n$ [4].

The DF-GREEDY implementation has much in common with the LAZY-GREEDY approach described here. The critical difference is that instead of the priority queue being indexed directly by current set size, as given the by *uc* component of the tuple, it is instead indexed by $\lfloor \log_p uc \rfloor$; that is, the number of distinct buckets in *pqueue* is given by $\lfloor \log_p n \rfloor$. Each bucket is represented as a disk file containing explicit sets, and the only requirement for memory space in the on-disk version of DF-GREEDY is the $n/32$ words required for the *covered* bit-vector.

The main processing loop of the DF-GREEDY method works systematically through the sets in a given bucket, for example, the k th bucket containing sets of at least p^k and less than p^{k+1} elements. As each set is read from the corresponding file and processed, it is checked for cleanliness. Provided that a set is clean, or nearly clean – where nearly clean is defined as containing at least p^k uncovered items – it is incorporated into the solution A. On the other hand, if, after cleaning, the set contains fewer than p^k uncovered items, it is written back to disk by appending it to the file storing the corresponding range of set sizes, and held for later processing.

The use of a geometric sequence of sizes means that no set can be processed more than $\lceil \log_p n \rceil$ times, and hence that there are at most $m \log_p n$ read and write operations while the algorithm executes, and, summed across the geometric sequence of set sizes, at most $Mp/(p-1)$ (that is, $O(M)$) words of data written. As many as $\log_p n$ files are required to be open for writing at any given instant, and one file for reading.

The final column of Table 4 shows the performance of an in-memory version of the DF-GREEDY approach. It is implemented using the same data structures as shown in Table 2, using a *pqueue* with $\log_p n + 2m + O(1)$ words required, with set size *uc* and a forward pointer required as part of each element. For the file WEBDOCS, for example, and with $p = 1.1$, there is a net space saving of around 14 MB compared to LAZY-GREEDY.

Needless to say, the DF-GREEDY approach is also fast when implemented this way, since it has a very similar execution profile to the LAZY-GREEDY mechanism. Indeed, the only significant difference between them is that in the LAZY-GREEDY approach, each bucket in *pqueue* represents a single set size, whereas in the in-memory DF-GREEDY implementation, each bucket represents a range of set sizes spanning (when $p = 1.1$) a 10% margin. That is, another way of categorizing the LAZY-GREEDY approach is that it is the limiting implementation, as $p \to 1$, of the in-memory DF-GREEDY mechanism.

The size of the solutions is very slightly higher when the in-memory DF-GREEDY is employed, but they are still close to the range established by the two "exact" approximate implementations. (Of course, the size of "true" solutions to these problem instances is not known.)

Interestingly, on the extreme problem instances described in Section 3.3, DF-GREEDY performs very well. For instance, with $m = 400$, DF-GREEDY with $p = 1.1$ requires just 0.35 seconds to return the solution, checking whether 70,391,613 items are covered. This is ten times faster than EAGER-GREEDY and over forty times faster than LAZY-GREEDY (Table 3). The point is that DF-GREEDY is not concerned when the set in focus has had a small fraction of its elements already covered, and in many cases adds the set to the solution anyway, bypassing the cascading cycle of cleansings that hinder the other two implementations.

Cormode et al. [4] compare in-memory and on-disk versions of their DF-GREEDY method with a range of other implementations, including an EAGER-GREEDY implementation. Using a 2.8 Ghz MacOS machine with 8 GB of main memory, their in-memory DF-GREEDY requires 93 seconds to process WEBDOCS with $p = 1.001$, and around 70 seconds with $p = 1.1$, compared to an EAGER-GREEDY execution time of 199 seconds. One possible explanation for the variation in execution time between their implementation of DF-GREEDY and our implementation of DF-GREEDY (shown as requiring 5 seconds in Table 4) is the use of different data structures; another contributing factor may be different experimental assumptions (for example, we do not include data reading times in Table 4).

5 Partial Set Cleansing

One further avenue was explored, still with the objective of reducing the execution time. Table 4 shows that the LAZY-GREEDY approach requires a minimum of three times more check-covered operations than does EAGER-GREEDY. Much of that effort is spent removing covered items from dirty sets even after it is known that set will not be the next one added to the solution. Removal of covered elements is, of itself, not wasted effort. But checking of the uncovered elements embedded in the same set is in some sense unhelpful, since those uncovered elements will all get checked again the next time this set emerges from the priority queue.

The idea of partial set cleansing is to interrupt the processing of checking a set once some threshold of "dirtiness" is exceeded, knowing that it will be some time before this set emerges again from the queue, and knowing that in the intervening period, more of the elements might have become covered.

Two different strategies were explored. The first was to abandon cleansing set S_i as soon as enough covered items had been identified in it to confirm that it could not be the next one added into the solution. That is, if set S_i was from *pqueue*[*uc*], and the next largest set was in (say) *pqueue*[*uc*′], then as soon as $uc - uc' + 1$ covered items have been found in S_i, it is returned to the queue in bucket *pqueue*[$uc' - 1$], and attention switched to the set or sets in bucket *pqueue*[*uc*′].

To avoid repeatedly testing and retesting elements from the start of each set, a variable *restart_point* is added to each node in the queue, to record the position from which check-covered operations should resume, when and if this set returns to the front of the queue again. Resuming the next loop of check-covered operations from *restart_point* rather than the beginning of the set increases the likelihood of identifying covered items; but does not remove the need for every item in the set to be checked before it can be added to the solution.

The second strategy was to continue checking set elements until some fixed fraction of the set was found to be covered, and reinsert a partially cleansed set into the queue as soon as that condition was met. When set cleansing is resumed, it is again from a stored *restart_point*.

Operations	LAZY-GREEDY	LAZY-GREEDY with partial set cleansing		
		$uc - uc' + 1$	$1 + 0.10 \times uc$	$1 + 0.50 \times uc$
Extract-maximum operations	3,291,585	287,420,896	44,731,342	10,276,636
Check-covered operations	335,085,502	314,325,433	319,843,092	326,240,303
Check-covered per extract-max.	101.80	1.09	7.15	31.75
Data movements	—	19,706,057	25,223,716	31,620,927
Running time (s)	5.29	189.71	37.40	12.37

Table 5: Operation counts for LAZY-GREEDY with full set cleansing, and three levels of partial set cleansing. The threshold of the partial cleansing is the maximum number of covered elements to be discovered prior to a suspension of the set difference. The critical factor determining execution time is the number of check-covered operations required.

Table 5 shows the costs associated with the revised process. Each approach to partial cleansing is able to slightly decrease the number of check-covered operations, but not by enough to also decrease the running time. Indeed, the number of extract-maximum operations grows significantly, adding considerably to the execution cost. It may be that other variants of this idea can be identified that achieve the hoped-for blend of attributes, but to date we have not identified a method that is significantly faster than the LAZY-GREEDY and DF-GREEDY implementations.

6 Conclusion and Future Work

We have explored the GREEDY approach to the SET COVER problem, describing and measuring the performance of a new implementation, the LAZY-GREEDY approach. Compared to the standard EAGER-GREEDY mechanism, the LAZY-GREEDY implementation requires around half the memory space, and as little as 5% of the execution time when applied to typical large problem instances. The trade-off is that the lazy set update process increases the worst-case running time to $\Omega(M^{4/3})$, compared to $O(M)$ for EAGER-GREEDY, where M is the total size of the input.

The LAZY-GREEDY implementation can be viewed as a special-case in-memory version of the DF-GREEDY method of Cormode et al. [4]. An interesting question is whether that relationship can be exploited in some way, perhaps to combine the speed of the DF-GREEDY method with the solution performance bound enjoyed by the LAZY-GREEDY approach in a single sequentially-processing implementation that is oblivious as to whether its data is resident in memory or on disk, and also retain the worst-case usefulness of DF-GREEDY.

We are also intrigued by the observation, arising from our experiments, that tie-breaking in the GREEDY mechanism can lead to solutions of differing quality. As a further part of our ongoing study, we plan to explore randomized choice evaluation to determine if hill-descending approaches might be effective in practice. The idea here would be to specify a total execution time that may be spent, and then execute GREEDY as many times as possible within that time, in order to obtain in aggregate a better solution than is likely to arise from a single execution.

Acknowledgments

Graham Cormode provided valuable input. This work was supported by the Australian Research Council.

References

[1] G. Ausiello, N. Bourgeois, T. Giannakos, and V. T. Paschos. Greedy algorithms for on-line set-covering. *Algorithmic Operations Research*, 4(1):36–48, 2009. URL `http://journals.hil.unb.ca/index.php/AOR/article/view/5928`.

[2] G. E. Blelloch, R. Peng, and K. Tangwongsan. Linear-work greedy parallel approximate set cover and variants. In *Proc. 23rd ACM Symp. Parallelism in Algorithms and Architectures*, pages 23–32, 2011. doi: 10.1145/1989493.1989497.

[3] G. E. Blelloch, H. V. Simhadri, and K. Tangwongsan. Parallel and I/O efficient set covering algorithms. In *Proc. 24th ACM Symp. Parallelism in Algorithms and Architectures*, pages 82–90, 2012. doi: 10.1145/2312005.2312024.

[4] G. Cormode, H. Karloff, and A. Wirth. Set cover algorithms for very large datasets. In *Proc. 19th ACM Int. Conf. Information and Knowledge Management*, pages 479–488, 2010. doi: 10.1145/1871437.1871501.

[5] H. S. Dutta. Survey of approximation algorithms for set cover problem. Master's thesis, University of North Texas, 2009. URL `http://digital.library.unt.edu/ark:/67531/metadc12118/m1/1/high_res_d/thesis.pdf`.

[6] U. Feige. A threshold of $\ln n$ for approximating set cover. *J. ACM*, 45(4):634–652, 1998. doi: 10.1145/285055.285059.

[7] 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): 3520–3534, 2006. doi: 10.1016/j.cor.2005.03.030.

[8] T. Grossman and A. Wool. Computational experience with approximation algorithms for the set covering problem. *Eur. J. of Operational Research*, 101(1):81–92, 1997. doi: 10.1016/S0377-2217(96)00161-0.

[9] R. Hassin and A. Levin. A better-than-greedy approximation algorithm for the minimum set cover problem. *SIAM J. Computing*, 35 (1):189–200, 2005. doi: 10.1137/S0097539704444750.

[10] D. S. Johnson. Approximation algorithms for combinatorial problems. *J. Computer and System Sciences*, 9:256–278, 1974. doi: 10.1016/S0022-0000(74)80044-9.

[11] R. V. Kantety, M. La Rota, D. E. Matthews, and M. E. Sorrells. Data mining for simple sequence repeats in expressed sequence tags from barley, maize, rice, sorghum and wheat. *Plant Molecular Biology*, 48(5-6):501–510, 2002. doi: 10.1023/A:1014875206165.

[12] R. M. Karp. Reducibility among combinatorial problems. In *Fifty Years of Integer Programming, 1958-2008*, pages 219–241. Springer, 2010. doi: 10.1007/978-3-540-68279-0_8.

[13] P. Slavík. A tight analysis of the greedy algorithm for set cover. *J. Algorithms*, 25(2):237–254, 1997. doi: 10.1006/jagm.1997.0887.

[14] N. E. Young. Greedy set-cover algorithms. In *Encyclopedia of Algorithms (Part 7)*. Springer, 2008. doi: 10.1007/978-0-387-30162-4_175.