Practical Algorithms for Finding Extremal Sets

MARTIN MARINOV, Trinity College Dublin
NICHOLAS NASH, Susquehanna International Group, Ireland
DAVID GREGG, Lero, Trinity College Dublin

The minimal sets within a collection of sets are defined as the ones which do not have a proper subset within the collection, and the maximal sets are the ones which do not have a proper superset within the collection. Identifying extremal sets is a fundamental problem with a wide-range of applications in SAT solvers, data-mining and social network analysis. In this paper, we present two novel improvements of the high-quality extremal set identification algorithm, *AMS-Lex*, described by Bayardo and Panda. The first technique uses memoization to improve the runtime of the single-threaded variant of the AMS-Lex, whilst our second improvement uses parallel programming methods. In a subset of the presented experiments our memoized algorithm executes more than 400 times faster than the highly efficient publicly available implementation of AMS-Lex. Moreover, we show that our modified algorithm's speedup is not bounded above by a constant and that it increases as the length of the common prefixes in successive input *itemsets* increases. We provide experimental results using both real-world and synthetic data sets, and show our multi-threaded variant algorithm out-performing AMS-Lex by 3 to 6 times. We find that on synthetic input datasets when executed using 16 CPU cores of a 32-core machine, our multi-threaded program executes about as fast as the state of the art GPU-based program using 512 CUDA cores. Furthermore, in the conducted experiments using real-world input datasets, our multi-threaded algorithm is almost always faster than the GPU-based approach.

Categories and Subject Descriptors: F.2.2 [**Analysis of Algorithms and Problem Complexity**]: Nonnumerical Algorithms and Problems — Computations on discrete structures

General Terms: Algorithms, Extremal Sets, Dataset, Itemset

ACM Reference Format:
Martin Marinov, Nicholas Nash and David Gregg, 2014. A Practical Algorithm for Finding Extremal Sets. *ACM J. Exp. Algor.* 9, 4, Article 39 (November 2014), 15 pages.
DOI: http://dx.doi.org/10.1145/0000000.0000000

1. INTRODUCTION

1.1. Motivation

The problem studied in this paper is that of finding the extremal sets within a dataset (family of sets) D. The extremal sets of D are all the sets in D that are maximal or minimal with respect to the partial order induced on D by the subset relation.

Finding extremal sets is a fundamental problem and has many motivating applications. For example, large-scale SAT solvers use extremal set identification as an optimization step [Eén and Biere 2005]. Extremal sets are also used for performing itemset support queries in data mining [Mielikäinen et al. 2006], and social network analysis [Bayardo and Panda 2011], as well as in trajectory-based query algorithms

This work is supported by the Irish Research Council (IRC).
Author's addresses: M. Marinov and D. Gregg, Department of Computer Science, Trinity College Dublin. N. Nash, Susquehanna International Group, Dublin, Ireland.

DOI: http://dx.doi.org/10.1145/0000000.0000000

with applications in surveillance [Vieira et al. 2009]. Early theoretical algorithms were motivated by problems in propositional logic [Pritchard 1991].

In this paper we present two optimization techniques that we apply to the AMS-Lex algorithm to achieve a faster runtime — the first one uses memoization and the second one parallel programming techniques. Using experimental evaluation we demonstrate the speedup achieved of both of them when compared to the highly efficient implementation of the AMS-Lex[1] algorithm described by Bayardo and Panda [2011].

1.2. Related Work

We denote by N the sum of the cardinalities of all the sets in the input dataset D, and informally refer to it as the size of the input. Although the algorithms for computing extremal sets are almost quadratic in N in the worst case, due to the nature of datasets in applications, practical algorithms can operate efficiently for very large N [Bayardo and Panda 2011]; i.e, in this paper we provide experimental results for $N = 7.2 \times 10^8$.

Yellin [1992] described algorithms for maintaining a dynamic family of sets, under insertion, deletion, intersection and subset query operations. He presents an output sensitive algorithm for identifying extremal sets after a sequence of n operations that operates in $O(mn)$ time, where m is the number of maximal sets. Note that n is the sum of N and the number of sets in the dataset, and hence $n > N$.

Early sub-quadratic time algorithms for finding extremal sets were described by Yellin and Jutla [1993], operating in $O(N^2/\log N)$ expected time, and by Pritchard [1997] who provided a matching worst-case time bound. Pritchard [1991] described the first algorithms that required sub-quadratic space, providing algorithms requiring $O(N^2/\log N)$ space.

Sheni and Evans [1996] also studied algorithms for maintaining a dynamic family of sets, operating in time $O(N^2/\log^2 N)$ and requiring $O(N^2/\log^3 N)$ space. We do not study this dynamic version of the extremal set problem in this paper.

Pritchard [1997] described the first algorithm to make use of a lexicographic ordering of the input sets. Among the practical algorithms for computing extremal sets is the highly efficient implementation of the AMS-Lex algorithm described by Bayardo and Panda [2011]. AMS-Lex is the state of the art practical algorithm for finding extremal sets that is designed to run on commodity CPUs. In this paper we give a detailed explanation of AMS-Lex in section 2 as it is the basis point of our work.

Fort *et al.* [2013] described a GPU based algorithm that out-performs Bayardo and Panda's algorithm. However, these results compare Bayardo and Panda's algorithm that is suitable for execution on a single CPU core to a GPU based algorithm running on 512 CUDA cores. The single-threaded algorithm we described in this paper is targeted at running on an ordinary commodity CPU and therefore we compare its performance to the algorithm of Bayardo and Panda [2011]. In the experimental evaluation section 5 we compare the runtime of our two new algorithms to Fort *et al.* [2013]'s reported runtime by evaluating on synthetically generated datasets.

[1]Bayardo and Panda have made their implementation of the AMS-Lex algorithm publicly available at https://code.google.com/p/google-extremal-sets/

1.3. Contributions

The main contributions of this work can be summarized as:

— A memoized version of AMS-Lex that takes advantage of common prefixes among itemsets.
— We outline a parallel modification of the AMS-Lex extremal sets algorithm.
— We present experimental results over both real-world and synthetic data for both the memoized and parallel modifications of the AMS-Lex extremal sets algorithms. We find that the speedup of the memoized algorithm increases as the length of the common prefixes of itemsets in the input dataset increases. Also that, the speedup of the parallel algorithm increases as the number of CPU cores in the system used for evaluating increase.

2. BACKGROUND

Practical algorithms for computing the extremal sets of a dataset D assume that the elements of D are *sets of items*, called *itemsets*. Furthermore, these algorithms assume that there is an ordering on the *itemsets* themselves. An input to an extremal set algorithm is then an ordered multiset of itemsets, referred to as a *dataset* D.

The choice of the ordering on the itemsets gives rise to alternative algorithms for computing extremal sets. For example, if itemsets are ordered by cardinality then the simple observation that if itemset a is a subset of itemset b then the cardinality of a is less than the cardinality of b can be used to prune the search space. This gives rise to an algorithm referred to as *AMS-Card* by Bayardo and Panda [2011].

Pritchard [1997] exploited a lexicographic ordering of itemsets to obtain more efficient algorithms for identifying extremal sets. In particular he noted the following:

THEOREM 2.1. *Let a and b be $itemsets$ such that $a \subset b$ then either a is a $proper\ prefix$ of b or a is lexicographically larger then b.*

The most efficient practical algorithm, AMS-Lex, for identifying extremal sets, described by Bayardo and Panda [2011], makes use of this lexicographic ordering of the preceding property to substantially prune the search space. In order to present our improvements we must first describe in detail the AMS-Lex algorithm.

2.1. The AMS-Lex algorithm

In this section we reproduce the AMS-Lex algorithm, we re-use the notation [Bayardo and Panda 2011] when referring to the input ordered dataset D:

— $D[i]$ denotes the i^{th} itemset in D
— $D[i][j]$ denotes the j^{th} item of itemset $D[i]$.
— $D[i:j]$ denotes the ordered multiset of itemsets $\{D[k] \mid k = i \ldots j\}$ in that order.
— $D[i][j:k]$ denotes the ordered multiset of items $\{D[i][l] \mid l = j \ldots k\}$.

We also re-use Bayardo and Panda's *subsumed* notation: an itemset A is subsumed by B iff A is a subset of B.

The pseudo code of the AMS-Lex algorithm itself is shown in Algorithm 2, and it applies the result from of Theorem 2.1 directly to first identify the proper prefixes that are subsumed, and then searching among the remaining itemsets using Contains-Subset-Of. The function Contains-Subset-Of takes as input an itemset S and dataset D and returns all $x \in D$ such that $x \subset S$ and x is lexicographically larger than S. Contains-Subset-Of makes use of the known common prefixes of itemsets in D as well as the lexicographic order of D. Since the items in the itemsets themselves are ordered lexicographically, the functions NextBeginRange, NextEndRange, and NextItem can be implemented using binary search.

Contains-Subset-Of Explanation. The Contains-Subset-Of function exploits the common prefixes of itemsets in D by taking advantage of the lexicographic order of D. The function is designed to efficiently find all itemsets in the range $D[b:e]$ that are subsets of S (i.e., that are subsumed by S). The itemsets in D are processed in ranges which share a common prefix of length at least d.

The first thing we check in the function is if the next item ($D[b][d+1]$) is contained in S by finding the first element of S which is greater than or equal to $D[b][d+1]$. If all elements of S smaller then $D[b][d+1]$ we can safely deduce that there are no subsumed itemsets by S in the range $D[b:e]$. This is because all itemsets in $D[b:e]$ are ordered lexicographically in ascending order. Hence if $S[|S|] < D[b][d+1]$ then $S[|S|] < D[i][d+1]$ for all i in the range $[b, e]$. Hence we reach a state where we know that the element $S[j] \geq D[b][d+1]$.

If $S[j] = D[b][d+1]$ then we know that it is possible for $D[b]$ to be a subset of S. Hence we have to make a recursive call to Contains-Subset-Of. In order to do this we have to first find a new end range e' such that all elements in $D[b:e']$ have a common prefix of length at least $d+1$. Then check if there are any subsumed itemsets. Next we check if the requirements of the recursive call to Contains-Subset-Of that we want to make are met. If this is the case then we mark subsumed items by S in the range $D[b:e]$. Since we have already covered the range $D[b:e']$ we set the current start of our range b to e'.

If $S[j] > D[b][d+1]$ then we know that $D[b]$ cannot be subsumed by S. Hence we search for the first element in $D[b:e]$ which has a value at index $d+1$ greater then or equal to $S[j]$, this operation is referred to as subroutine NextBeginRange.

Lastly we check if the current begin range is smaller then the current end range and if it is the case we mark all subsumed sets of S in the range $D[b:e]$ by making a recursive call to Contains-Subset-Of.

3. AN IMPROVED ALGORITHM FOR IDENTIFYING EXTREMAL SETS

3.1. Observations

Our improved algorithm for extremal set identification memoizes successive calls to the function Contains-Subset-Of, defined in Algorithm 1. As we explain below, Bayardo and Panda's algorithm AMS-Lex presented in Algorithm 2 duplicates work in successive calls to Contains-Subset-Of where itemsets share a non-empty common prefix. We now show more precisely how this work is duplicated, in terms of the call-graphs resulting from successive calls to Contains-Subset-Of.

Definition 3.1. The directed call graph of an itemset S and the function Contains-Subset-Of($D[b:e], S, j, d$) is defined as a graph $G(S) = (V, E)$, where $V = \{(b, e, S, j, d) \mid b, e, S, j$ and d meet the input requirements of Contains-Subset-Of$\}$, and $(v_1, v_2) \in E$ iff Contains-Subset-Of($v_1.b,\ v_1.e,\ v_1.S,\ v_1.j,\ v_1.d$) makes a recursive call to Contains-Subset-Of($v_2.b,\ v_2.e,\ v_2.S,\ v_2.j,\ v_2.d$).

Remark 3.2. Note that since the Contains-Subset-Of function in Algorithm 1 performs at most two recursive calls, hence the out-degree of any vertex in a call-graph $G(S)$ is at most two.

NOTATION 3.3. *For a call graph $G(S) = (V, E)$ and any $v = (b, e, S, j, d) \in V$, we refer to the values of v as $v.b$, $v.e$, $v.j$, $v.d$, $v.t$ and $v.m$; and we refer to the children of v as $v.c_1$ and $v.c_2$. We denote $v.t$ as a boolean field which is true iff there exists a subset of S in the range $[v.b; v.e)$ that is of size $v.d+1$. We refer to $v.m$ to be the maximum index that is accessed from the itemset S without considering any recursive calls of Contains-Subset-Of.*

Remark 3.4. Note that at any *single* call-graph node corresponding to a call to function $Contains - Subset - Of(D[b:e], S, i, d)$ the only indices of S that are required are those between j and NextItem$(S, j, D[b][d+1])$, and we refer to the maximum value as $v.m$ for any $v \in V$.

LEMMA 3.5. *Let S and T be itemsets with a common prefix P. Let $G(S) = (V_S, E_S)$ and $G(T) = (V_T, E_T)$. Suppose that $v_1, v_2 \in V_S$, where $v_1 = (b, e, S, j, d)$, and $v_2 = (b', e', S, j', d')$ such that $j' < |P|$, and that $(v_1, v_2) \in E_S$. Then $(w_1, w_2) \in E_T$ where $w_1 = (b, e, T, j, d)$ and $w_2 = (b', e', T, j', d')$.*

PROOF. Referring to Algorithm 1 note that because S and T have a common prefix P of length greater than j' all requirements of Contains-Subset-Of are met for the inputs represented by w_1 and w_2. Hence we have $w_1, w_2 \in V_T$. We now need to show that there is an edge between w_1 and w_2. Since $(v_1, v_2) \in E_S$ and from Remark 3.4 the only values required of S by Contains-Subset-Of are in the range $[j, j']$ and as a result of the further assumption that $j' < |P|$ it follows immediately that $(w_1, w_2) \in E_T$. □

Remark 3.6. Note that for any itemset S, the call graph $G(S) = (V, E)$ is acyclic because in all recursive calls to Contains-Subset-Of the range $[b, e]$ gets smaller, S is always constant, j increases and d increases.

NOTATION 3.7. *For any itemset S, we refer to the subgraph of $G(S) = (V, E)$ identified by $V' = \{(b, e, S, j, d) \in V \mid j < |P|\}$ as $G(S)|_{j<|P|}$.*

COROLLARY 3.8. *Let S and T be itemsets with a common prefix P. Then $G(S)|_{j<|P|} = G(T)|_{j<|P|}$.*

PROOF. Use induction to apply Lemma 3.5 multiple times starting from the root of $G(S)$ identified by the vertex $(b, e, S, j = 1, d = 0)$. □

3.2. Algorithm

The pseudo code of our modified algorithm for identifying minimal sets is presented in Algorithm 4 and we now give an informal description of its behaviour. For each call made to Contains-Subset-Of$(D[i+1, n], D[i], 1, 0)$ we memoize the call graph $G(D[i])$ of the execution path. When we get to the point when we need to find if there is a subsumed itemset by $D[i+1]$ we first identify the common prefix P of $D[i]$ and $D[i+1]$. Then we traverse $G(D[i])$ using depth first search. For each vertex v we check if a recursive call is made to Contains-Subset-Of with some $j \geq |P|$. If this is the case then we execute the function Contains-Subset-Of with input v; otherwise we recursively traverse the children of v. This is a direct result from Corollary 3.8. In practice we note that, we need not memoize the full call graph $G(D[i])$ as we are only ever going to use nodes $w \in G(D[i])$ for which $w.j < |P|$.

Remark 3.9. It is important to note that we use a modified version of the function Contains-Subset-Of by assuming that it returns a pair of a boolean result as per the specification from Algorithm 1 and the call graph representing its execution path. We use this in the pseudo code of the memoized version of the memoized version of AMS-Lex presented in Algorithm 4.

3.3. Complexity Analysis

Worst Case Time Complexity. It is easy to see that in the worst case (when no two itemsets have a common prefix), the complexity of our algorithm is equal to that of AMS-Lex, that is $O(N^2/\log(N))$, where N is the sum of the cardinalities of all itemsets in the input dataset.

Runtime Comparison to AMS-Lex. Our algorithm's run time is clearly bounded above by the time required by AMS-Lex. Moreover, as the number of common prefixes among the $itemsets$ increases, the faster (comparatively) our algorithm becomes. Essentially by executing Contains-Subset-Of fewer times, we save run time consumed by the low level searching routines $NextItem$, $NextEndRange$, and $NextBeginRange$ which are the bottleneck of the AMS-Lex algorithm as per [Bayardo and Panda 2011].

Space Complexity. In addition to the memory required by AMS-Lex, Algorithm 2 stores (part of) the call graph of Contains-Subset-Of. Clearly the size of the call graph is bounded above by the size of the input, denoted as N. Since only the required portion of the call graph, as defined by Corollary 3.8 to be stored in practice, the extra space required is often much less than the size of the input.

3.4. Implementation Details

We implemented our algorithm as a modification to the publicly available implementation[2] of the AMS-Lex algorithm, only introducing the memoization described in Algorithm 4. We regard this as valuable since it allows us to directly measure the improvement in performance resulting from memoization.

4. A PARALLEL ALGORITHM FOR IDENTIFYING EXTREMAL SETS

We use the complexity analysis of the function AMS-Lex [Bayardo and Panda 2011] to identify the bottleneck of the existing algorithm. In the worst case, finding all proper prefix subsumed itemsets takes $\mathrm{O}(N)$ computational steps and finding the remaining non-minimal itemsets takes $\mathrm{O}(N^2/log(N)) > \mathrm{O}(N)$, where N is the size of the input. Consequently, the novel work presented in this section is a parallel algorithm that finds the non-proper prefix subsumed itemsets of D, i.e. we present a parallel implementation of the function Get-Minimal-Itemsets-Lex from Algorithm 2.

4.1. Observation

The first observation we make is that the pseudo code of the function Contains-Subset-Of, presented in Algorithm 1 that is a reproduction of Contains-Subset-Of [Bayardo and Panda 2011], does not modify the input dataset D. Hence, this makes the algorithm of finding all minimal itemsets within D embarrassingly parallel. Note that, for simplicity of explanation of this parallel algorithm we have chosen to present in depth analysis of the method for finding minimal itemsets within a dataset in this paper rather than maximal ones. To obtain a version of this algorithm for identifying the maximal itemsets, we would need a small modification to the Contains-Subset-Of method to make sure it does not alter the dataset D which is trivial.

4.2. Algorithm

The pseudo code for our parallel algorithm of finding the minimal itemsets within a lexicographically ordered dataset is presented in Algorithm 5.

Entry Point. We first mark every itemset within the dataset D as minimal. Next, we mark all itemsets as no minimal for which there exists a proper prefix subsumed itemset within the dataset. We then start P parallel instances of the thread functor whose job is to mark itemsets as non-minimal for which there exists a non-prefix (lexicographically larger) subsumed itemset.

Thread Functor. All of the parallel instances of the Thread-Functor function share a common integer variable $index$ which points to the next unprocessed itemset

[2]https://code.google.com/p/google-extremal-sets/

$D[index] \in D$ within the datasets tarting at 1. To process the itemset $D[index]$ means to check if there exists a non-prefix subsumed within D of $D[index]$. We begin by atomically assigning the current value of $index$ to the variable i and incrementing $index$; ensuring that every itemset in D will be processed exactly once by some Thread-Functor. We then use the function Contains-Subset-Of from Algorithm 1 to check if a subset of $D[i]$ is found. Finally, we try to take a new unprocessed itemset from D and process it in the same manner.

4.3. Complexity

Here we give the worst case time and space complexity of the functions presented in Algorithm 5. From Bayardo and Panda [2011]'s complexity analysis of AMS-Lex we know that the worst case time complexity of AMS-Lex is equal to $\mathrm{O}(N)$ to identify the prefix subsumed itemsets and additional $\mathrm{O}(N^2/log(N))$ to find the non-prefix subsumed ones; recall that N denotes the sum of the cardinalities of all the sets in the input dataset D. Since in this section we showed that, the function Contains-Subset-Of requires only read-only access to the dataset D and we have P threads at our disposal we deduce that worst case runtime of the function Get-Minimal-Itemsets-Lex-Parallel is $\mathrm{O}(N) + \mathrm{O}(N^2/(log(N) \times P)) = \mathrm{O}(N + N^2/(log(N) \times P))$; note that $1 \leq P \leq n$. As for the space complexity of the Get-Minimal-Itemsets-Lex-Parallel algorithm it is equal to that of Get-Minimal-Itemsets-Lex [Bayardo and Panda 2011] which is proportional to the size of the input, i.e. $\mathrm{O}(N)$.

5. EXPERIMENTS

Here we describe the experimental comparison of our algorithm with Bayardo and Panda's algorithm AMS-Lex for identifying the minimal itemsets within a dataset. We measure runtime speedup as the ratio of AMS-Lex algorithm runtime divided by our algorithm's runtime. Hence, a speedup of 2 means that our algorithm executed in half the time, and a value of 1 means that both algorithms have the same runtime. For every input, we also measure the total number of calls that each algorithm made to the subroutines $NextBeginRange$ and $NextEndRange$, because as described in [Bayardo and Panda 2011], these subroutines are the bottleneck of the AMS-Lex algorithm. In our experimental evaluation we provide a link between the decrease in the number of range searches performed by our algorithm in comparison to AMS-Lex and the relative to AMS-Lex runtime speedup.

Although not presented below, we also conducted experiments with the Bayardo and Panda's AMS-Card Algorithm on a subset of the data and it performed slower on all cases, compared to the AMS-Lex algorithm. That is expected as stated in [Bayardo and Panda 2011] the cardinality approach is faster then the lexicographic one mostly in very obscured cardinality distributions. Furthermore, the goal of this paper is to present faster than AMS-Lex methods of finding extremal sets that are based on Pritchard's lexicographic subsumption property from Theorem 2.1.

5.1. Experimental Setup

For all of our experiments we used a machine with four Intel Xeon CPU E7- 4820 clocked at $2.00GHz$, a third level cache size of $18MB$ and $128GB$ of main memory. Note that our experiments investigate the case when the entire input fits in main memory. We used uniform random data as well as publicly available data as input to evaluate our two new algorithms and AMS-Lex. All of the results presented below are averaged over 3 different runs.

5.2. Real-World Data

A summary of the conducted experiments using real-world input datasets is presented in Figure 1. We have evaluated the AMS-Lex algorithm, our memoized approach and the parallel method using different degrees of parallelism over the real-world datasets:

— **PubMed** dataset represents significant terms in the PubMed abstract. It consists of 8 million itemsets stored in a $2GB$ file.
— **DBLP** dataset consists of 1 million itemsets and is used in the area of similarity joins. The file size is $50MB$.
— **SN_9_4** dataset consists of 2 million itemsets with an average size of 30.3 and an alphabet size of 2^9. This data is derived from the domain of 9-input sorting networks by generating all non maximal networks of depth 4. The file size is $252MB$.
— **SN_9_5** dataset consists of 2 million itemsets with an average size of 30.3 and an alphabet size of 2^9. This data is derived from the domain of 9-input sorting networks by generating all non maximal networks of depth 5 by using the minimal ones of depth 4. The file size is $578MB$.

Memoized vs AMS-Lex. For the $DPLP$ and $PubMed$ datasets the memoized approach is marginally faster than the AMS-Lex algorithm because there are very few itemset pairs that share a common prefix. On the other hand, for the SN_9_4 dataset the memoized algorithm is 4.06 times faster than AMS-Lex; and 2.96 times faster for the SN_9_5 dataset. The sorting network input datasets tend to share long common prefixes as the size of the alphabet is very small compared to the size of the input which favours our memoization technique over AMS-Lex. It is important to note that in the sorting network datasets there are no trivially subsumed itemsets.

Parallel vs AMS-Lex. Note that our parallel algorithm is executed on a machine with 32 physical cores and all real-world experimental results are presented in Figure 1. For the $DBLP$ dataset we see that the speedup of the parallel algorithm over AMS-Lex is about 3.5 for degrees of parallelism $P = 4, 8$ and 16 whereas for $P = 32$ we see a reduced speedup. For the $PubMed$ dataset we see substantial speedup for all of the parallelism factors with $P = 16$ executing 5.6 times faster than AMS-Lex. Substantial runtime speedups are evident in the SN_9_4 and SN_9_5 datasets both of them peeking at $P = 16$ with maximum speedup factors of 5.3 and 5.9 respectively. It important to note that these real-world data runtime speedups are comparatively equal and/or better than the ones that the GPU [Fort et al. 2013] approach achieves over the AMS-Lex algorithm. Hence, we can conclude that our parallel version of AMS-Lex is faster than AMS-Lex on real-world data and at least as fast as the GPU algorithm.

5.3. Synthetic Data

Input Dataset Generation. We now describe the process of generating random input data using a random data generator program $g(n, d, f_{min})$. The input to the generator is the number of itemsets n, the number of distinct items d in the alphabet and the minimal item frequency f. Then for each of the d items we choose a frequency f_i from the range $[f_{min}, 1]$ which indicates the number of itemsets which contain this item. Then we insert this item to a set of randomly chosen $\lfloor f_i \times n \rfloor$ itemsets. Then we use Bayardo and Panda's open source implementation to sort the input data in the format required by the algorithms. Note that the higher the value of the minimal frequency f_{min} the greater the probability that two itemsets will share a common prefix. We use the value of f_{min} to evaluate our hypothesis that our algorithm is faster than AMS-Lex on inputs consisting of itemsets sharing large common prefixes.

Memoized vs AMS-Lex. Figure 2 shows the runtime speedup factor of our memoized algorithm over AMS-Lex for datasets consisting $n = 100\,000$, $n = 500\,000$ and $n = 1\,000\,000$ itemsets with alphabet size of 40, 60, 80, 100, 120 and 140. We notice that as the minimal item frequency increases, the speedup factor increase drastically. The maximum runtime speedup factor of 406 is achieved by a dataset consisting of $N = 1\,000\,000$ itemsets with alphabet size of $D = 140$ and minimal frequency of $F = 0.95$. We also note that there is an approximately constant correlation between the runtime speedup of our algorithm and the factor of reduction in range search calls. That is an expected correlation because these low level subroutines are described as the bottleneck of AMS-Lex [Bayardo and Panda 2011].

In Section 3 we showed that the more common prefixes that itemsets have, i.e. as f_{min} increases and we keep n and d fixed, the bigger the expected speedup factor, which is experimentally verified by this figure. We note that fixing the size of the alphabet d and the minimal item frequency f_{min}, in Figure 2 we see that as the number of itemsets n increases, the runtime speedup of the memoized algorithm over AMS-Lex increases. Also, if we fix n and f_{min} we see that as d increases the runtime speedup is non-decreasing in all of the conducted experiments.

Another interesting summary of our experiments is shown in Figure 3 which gives the runtime speedup with respect to the cardinality of the resulting minimal itemsets by presenting three different graphs for $n = 100\,000$, $n = 500\,000$ and $n = 1\,000\,000$. Our first impression is that all of the graphs look very similar to each other besides the scale of the runtime speedup access. Our second observation shows that the largest speedups are almost always achieved at the smallest resulting minimal sets count for every d and n. Moreover, as d increases the absolute maximum speedup increases as well and all speedups tend to 0 when the size of the result is close to the size of the input (0.9 to 1.0).

Parallel vs AMS-Lex. We have summarised the conducted experiments in Figure 4 which presents the runtime speedup of the parallel algorithm over AMS-Lex using degrees of parallelism $P = 4, 8, 16$ and 32 on a machine with 32 physical cores. As input to the algorithm we used datasets with $n = 1\,000\,000$ itemsets with alphabet size of 40, 60, 80, 100, 120 and 140; note that these datasets are the same as the ones used for experimentally comparing the memoized approach versus AMS-Lex consisting of one million itemsets. From the figure, we see that as d increases and keeping n and f_{min} fixed we see that the runtime speedup increases, but it does tend to reach maximum unlike the analogous comparison of memoized over AMS-Lex. We note very small difference in the speedups with $P = 8$ and $P = 16$, whereas as they are both slightly larger then the speedups achieved using 4 threads.

It is very interesting and important to note that in the case of $P = 32$ we have a significant decay in the speedup over AMS-Lex in comparison to $P = 4, 8$ and 16. Also, this is the only example we encountered that any of our algorithms is even by a very small amount slower (speedup smaller than 1 on the graphs) than AMS-Lex. That is explained with the fact that the AMS-Lex algorithm and all of its variations presented here are not computationally intensive but rather memory read access bounded. In this case when P equals the number of physical cores, we found more L3 cache misses in comparison to smaller parallelism factors P; also there is a competition for the memory bus and as P increases we inevitably hit the limit of the bus. The cache locality and the memory insensitivity of the application arguments also explains the observed maximum speedups of around 4 because the machine we used consists of 4 physical CPU chips, each with its on L3 cache.

Comparison to GPU Approach. Fort's algorithm for finding extremal sets on a GPU is compared to the AMS-Lex algorithm in [Fort et al. 2013]. By carefully analysing the

experimental comparison of Fort's algorithm to AMS-Lex we see that when we exclude the time to pre-process and sort the input dataset to the required format by AMS-Lex then when evaluated on synthetic data the GPU algorithm is between 4 and 5 times faster than AMS-Lex. Moreover the runtime speedup demonstrated by the GPU algorithm seems to be constant over AMS-Lex. As presented in Figure 4, our parallel algorithm is between 3 and 4.5 times faster than AMS-Lex when executed with $P = 16$ on a 32 core machine which is similar to the speedup of the GPU algorithm over AMS-Lex. One the other hand, the speedup of our memoized approach over AMS-Lex is not bounded above by a constant as demonstrated. The runtime speedup of our memoized method for datasets with $1\,000\,000$ itemsets over AMS-Lex is as high as 400 which is much bigger than the speedup achieved by the GPU approach over AMS-Lex.

6. CONCLUSION

This paper has presented two improved algorithms for identifying extremal sets within a dataset. We have experimentally demonstrated that both techniques improve the performance of the AMS-Lex algorithm on both real world and synthetic datasets. Our first improved algorithm uses memoization to remove redundant work from the AMS-Lex [Bayardo and Panda 2011] requiring at most twice the memory of AMS-Lex. In a subset of the conducted experiments the memoized algorithm executes more than 400 times faster than AMS-Lex. We show in theory and practice, that the efficiency of this improved algorithm increases as the common prefixes shared by itemsets increases, hence the speedup when compared to AMS-Lex is not bounded above by a constant which is also evident in the experiments provided. The second improved algorithm uses parallelism to speedup the AMS-Lex algorithm. In the conducted experiments we show that our parallel approach outperforms Bayardo and Panda's implementation of AMS-Lex on both real-world and synthetic datasets. This parallel approach is about as fast as the GPU algorithm of Fort *et al.* [2013] on synthetic data, and faster than it on almost all real-world datasets.

REFERENCES

Roberto J. Bayardo and Biswanath Panda. 2011. Fast Algorithms for Finding Extremal Sets. In *SDM*. SIAM / Omnipress, 25–34.

Niklas Eén and Armin Biere. 2005. Effective Preprocessing in SAT Through Variable and Clause Elimination. In *SAT (Lecture Notes in Computer Science)*, Fahiem Bacchus and Toby Walsh (Eds.), Vol. 3569. Springer, 61–75.

Marta Fort, J. Antoni Sellars, and Nacho Valladares. 2013. Finding extremal sets on the GPU. *J. Parallel and Distrib. Comput.* 0 (2013), –.

Taneli Mielikäinen, Pance Panov, and Saso Dzeroski. 2006. Itemset Support Queries Using Frequent Itemsets and Their Condensed Representations. In *Discovery Science (Lecture Notes in Computer Science)*, Ljupco Todorovski, Nada Lavrac, and Klaus P. Jantke (Eds.), Vol. 4265. Springer, 161–172.

Paul Pritchard. 1991. Opportunistic Algorithms for Eliminating Supersets. *Acta Inf.* 28, 8 (1991), 733–754.

Paul Pritchard. 1997. An Old Sub-Quadratic Algorithm for Finding Extremal Sets. *Inf. Process. Lett.* 62, 6 (1997), 329–334.

Hong Sheni and D. J. Evans. 1996. Fast sequential and parallel algorithms for finding extremal sets. *International Journal of Computer Mathematics* 61, 3-4 (1996), 195–211. DOI:http://dx.doi.org/10.1080/00207169608804512

Marcos R. Vieira, Petko Bakalov, and Vassilis J. Tsotras. 2009. On-line discovery of flock patterns in spatio-temporal data. In *GIS*, Divyakant Agrawal, Walid G. Aref, Chang-Tien Lu, Mohamed F. Mokbel, Peter Scheuermann, Cyrus Shahabi, and Ouri Wolfson (Eds.). ACM, 286–295.

Daniel M. Yellin. 1992. Algorithms for Subset Testing and Finding Maximal Sets. In *SODA*, Greg N. Frederickson (Ed.). ACM/SIAM, 386–392.

Daniel M. Yellin and Charanjit S. Jutla. 1993. Finding Extremal Sets in Less than Quadratic Time. *Inf. Process. Lett.* 48, 1 (1993), 29–34.

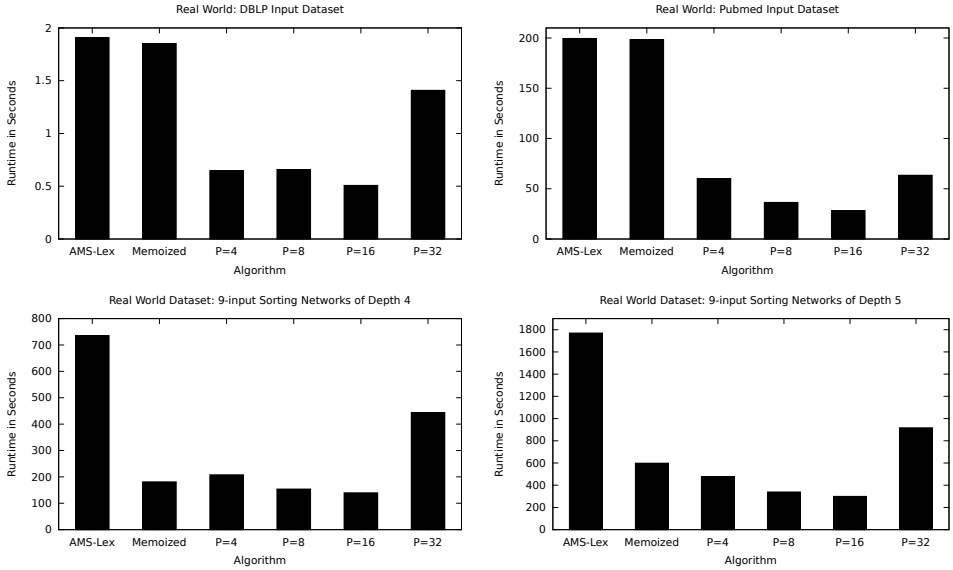

Fig. 1. Experimental results using real world datasets, comparing AMS-Lex with the memoized (section 3) and parallel (section 4) approach. For these results we have used a machine with 32 physical cores and used parallelism factors $P = 4, 8, 16$ and 32 for our parallel modification of AMS-Lex.

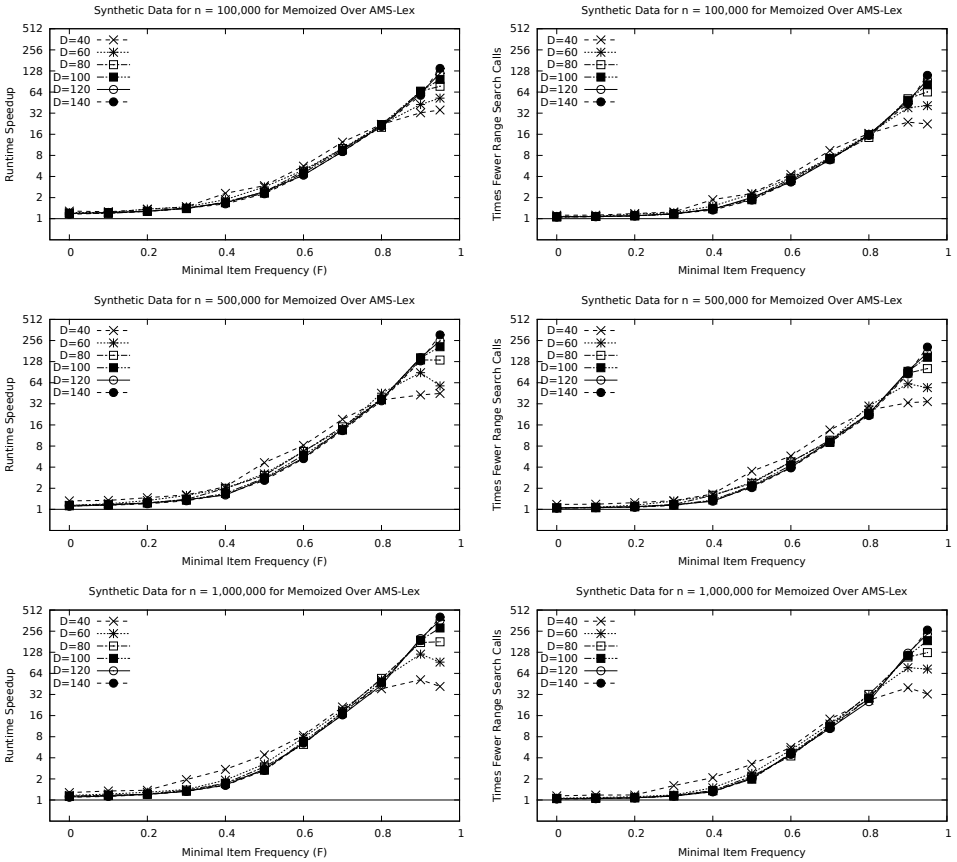

Fig. 2. Experimental results using synthetic data for $n = 100\,000$, $n = 500\,000$ and $n = 1\,000\,000$ of comparing our memoized version of AMS-Lex (section 3) over AMS-Lex. Here d is the cardinality of the domain of the itemsets. These results show the minimal item frequency (f_{min}) described in Section 5 against the resulting runtime speedup as well as the decrease in range search calls of our memoized algorithm compared over AMS-Lex. Note that the y-axis in every graph uses a $\log_2$ scaling for visual clarity of the presented graphs.

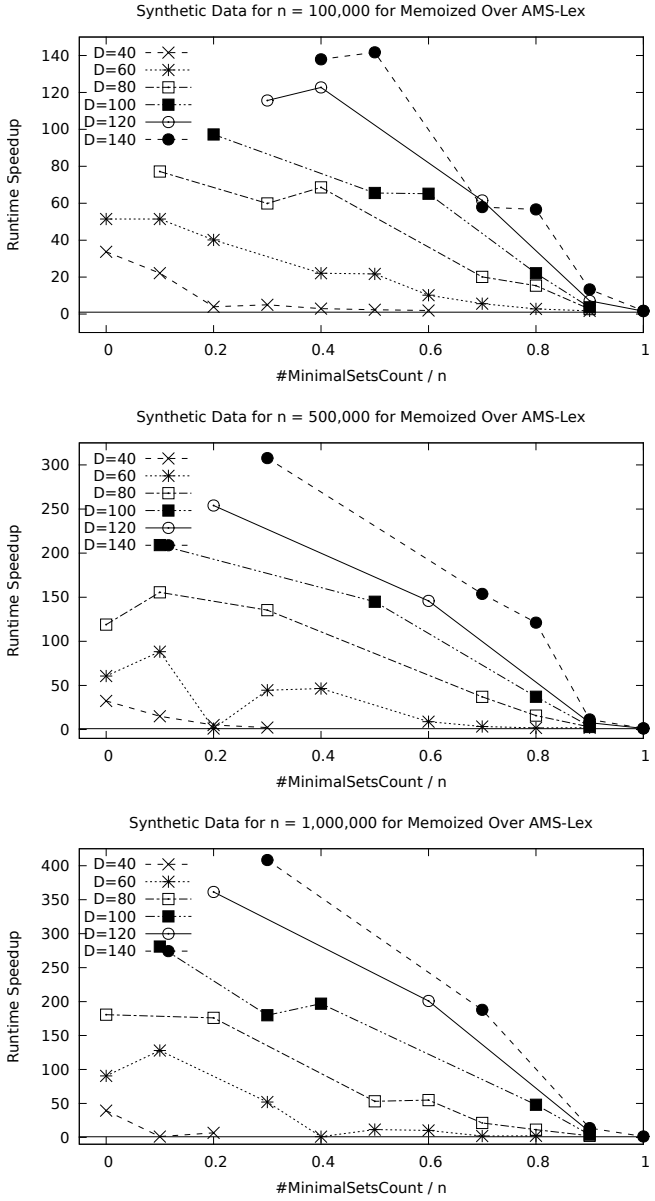

Fig. 3. Experimental results using synthetic data for $n = 100\,000$, $n = 500\,000$ and $n = 1\,000\,000$ of comparing our memoized version of AMS-Lex (section 3) over AMS-Lex. Here d is the cardinality of the alphabet. These results show the number of minimal itemsets against the resulting runtime speedup of our memoized algorithm compared to AMS-Lex.

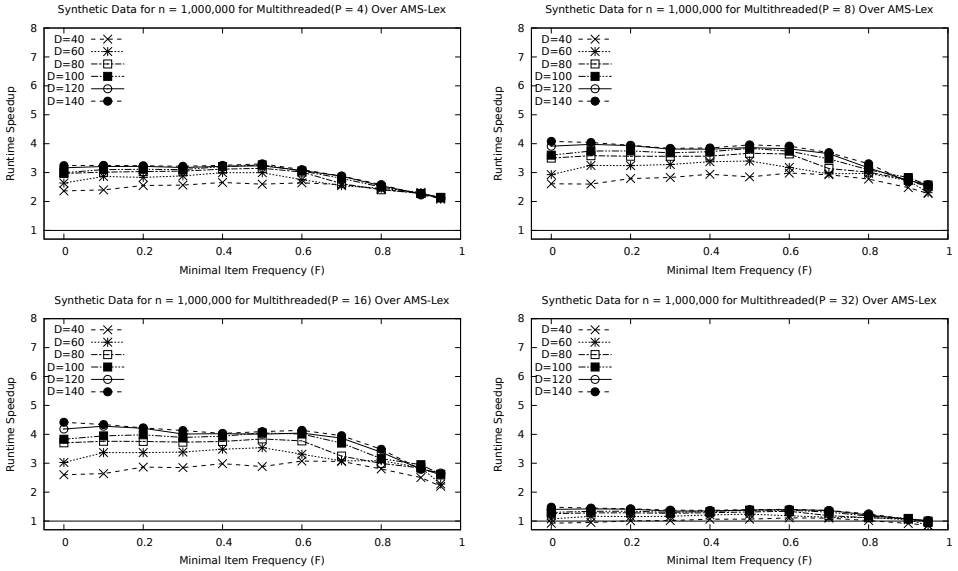

Fig. 4. Experimental results for synthetic data for $n = 1\,000\,000$ of comparing our parallel version of AMS-Lex over AMS-Lex. Here D is the cardinality of the domain of the itemsets. These results show the minimal item frequency described in Section 5 against the resulting runtime speedup of our parallel algorithm compared to AMS-Lex. For these results we have used a machine with 32 physical cores and used parallelism factors of 4, 8, 16 and 32 for our parallel modification of AMS-Lex described in section 4.

ALGORITHM 1: Pseudo code for finding if the input dataset $D = \{D_1, D_2, \ldots, D_n\}$ contains a proper subset of $D[i]$. A reproduction of the function MarkSubsumed, described by Bayardo and Panda, but used for finding the minimal itemsets rather than the maximal ones, i.e. for finding the minimal itemsets within the dataset D we do not mark the subsumed itemsets but rather return true if a properly subsumed itemset by $D[i]$ exists within D and false otherwise.

```
Function Contains-Subset-Of(D[b : e], S, j, d)
    Input: The ordered multiset of itemsets D[b : e], an itemset S and two integers j and d. The
           parameter j specifies we need only consider S[j : |S|] and d is the size of the
           common prefix shared by all I ∈ D[b : e] and S.
    Output: Returns true iff there exists a proper subset of D[i] within D[b : e], and false
            otherwise.
1   if S[j] < D[b][d + 1] then
2       j ← NextItem(S, j, D[b][d + 1]);
3       if j is null then
4           return false;
        end
    end
5   if S[j] = D[b][d + 1] then
6       e' ← NextEndRange(D[b : e], S[j], d);
7       if |S| > d + 1 then
8           if b ≤ e' and |D[b]| = d + 1 then
                /* D[b] is a proper subset of S.                                    */
9               return true;
            end
        end
10      if j + 1 < |S| and b ≤ e' then
11          if Contains-Subset-Of(D[b : e'], S, j + 1, d + 1) then
12              return true;
            end
        end
13      b ← e';
    else
14      b ← NextBeginRange(D[b : e], S[j], d);
    end
15  if b ≤ e then
16      return Contains-Subset-Of(D[b : e], S, j, d);
    end
17  return false;
```

ALGORITHM 2: Pseudo code for finding the minimal itemsets within the dataset $D = \{D_1, D_2, \ldots, D_n\}$ by using the lexicographic constraint (Theorem 2.1). A reproduction of the AMS-Lex algorithm described by Bayardo and Panda, but used for finding the minimal itemsets rather than the maximal ones, i.e. for finding the minimal itemsets within the dataset D we do not mark the subsumed itemsets but rather mark an itemset as non-minimal if it is a superset another one.

Function `Get-Minimal-Itemsets-Lex(`D`)`

Input: Dataset $D = \{D_1, D_2, \ldots, D_n\}$ that is ordered lexicographically and every itemset $I \in D$ is also ordered lexicographically.

Output: The minimal itemsets within the dataset $D = \{D_1, D_2, \ldots, D_n\}$.

1 $bool\ is\_min[n] \longleftarrow \{true, true, \ldots, true\}$;

`/* Find itemsets subsumed by proper prefix. */`

2 $S \longleftarrow D[1]$;

3 **for** $i = 2$ **to** n **do**

4 **if** $|S| \leq |D[i]|\ \ \&\ \ D[i][1 : |S|] = S$ **then**

`/*` S `is a proper prefix of` $D[i]$`. */`

5 $is\_min[i] \longleftrightarrow false$;

end

else

6 $S \longleftarrow D[i]$;

end

end

`/* Find itemsets subsumed by non-proper prefix. */`

7 **for** $i = 1$ **to** $n - 1$ **do**

8 **if** $is\_min[i]$ & `Contains-Subset-Of(`$D[i+1:n], D[i], 1, 0$`) /* see Algorithm 1 */` **then**

9 $is\_min[i] \longleftarrow false$;

end

end

10 **return** $\{D_i \in D \mid is\_min[i] = true\}$;

ALGORITHM 3: Pseudo code for finding if the input dataset $D = \{D_1, D_2, \ldots, D_n\}$ contains a proper subset of $D[i]$ by using memoization.

Function `Contains-Subset-Of-Memoized`(v, i, p)

Input: The ordered multiset of itemsets $D[b:e]$, an itemset S and two integers j and d. The parameter j specifies we need only consider $S[j:|S|]$ and d is the size of the common prefix shared by all $I \in D[b:e]$ and S.

Output: Returns $true$ iff there exists a proper subset of $D[i]$ within $D[b:e]$, and $false$ otherwise.

1 **if** $v.max\_j \geq p$ **then**
  `/* The maximum index that was accessed from the method Contains-Subset-Of in the memoized iteration represented by` v `is larger then the size of the common prefix, so we must invoke the the function to find the non-proper subsets of` $D[i]$ `as no more memoized results can be used. */`
2 $b \leftarrow max(v.b, i+1)$;
3 **if** $b < v.e$ **then**
    `/* We assume a modified version of the function Contains-Subset-Of which returns a pair consisting of a boolean variable and a node representing the call stack of the function. */`
4 $\langle res, v \rangle \longleftarrow$ `Contains-Subset-Of`$(D[b:v.e], D[i], v.j, v.d)$;
5 **return** res;
  end
6 $v \longleftarrow null$;
7 **return** $false$;
end
8 **if** $v.c_1 \neq null$ **then**
9 **if** `Contains-Subset-Of-Memoized`$(v.c_1, i, p)$ **then**
10 $res \longleftarrow true$;
  end
end
11 **if** $v.c_2 \neq null$ **then**
12 **if** `Contains-Subset-Of-Memoized`$(v.c_2, i, p)$ **then**
13 $res \longleftarrow true$;
  end
end
`/* recall that` $v.t$ `equals true iff a subset was found in the execution of the function Contains-Subset-Of without considering the recursive calls; i.e. there exists a non-proper subset of` $D[i]$ `of size smaller then the length of the common prefix` p `between` S `and` $D[i]$. `*/`
14 **return** $v.t$;

ALGORITHM 4: Pseudo code for finding the minimal itemsets within the dataset $D = \{D_1, D_2, \ldots, D_n\}$ by using memoization and the lexicographic constraint (Theorem 2.1).

```
Function Get-Minimal-Itemsets-Lex-Memoized(D)
    Input: Dataset D = {D_1, D_2, ..., D_n} that is ordered lexicographically and every itemset
           I ∈ D is also ordered lexicographically.
    Output: The minimal itemsets within the dataset D.
1   bool is_min[n] ⟵ {true, true, ..., true};
    /* Find itemsets subsumed by proper prefix.                                    */
2   S ⟵ D[1];
3   for i = 2 to n do
4       if |S| ≤ |D[i]| & D[i][1 : |S|] = S then
            /* S is a proper prefix of D[i].                                       */
5           is_min[i] ⟷ false;
        end
        else
6           S ⟵ D[i];
        end
    end
    /* Find itemsets subsumed by non-proper prefix.                                */
7   S ⟵ null;
8   v ⟵ null;
9   for i = 1 to n − 1 do
10      if is_min[i] then
11          if v = null then
                /* defined in Algorithm 1 but assuming that it returns a pair of a
                   boolean value res and the call stack represented by v.          */
12              ⟨res, v⟩ ⟵ Contains-Subset-Of(D[i + 1 : n], D[i], 1, 0);
13              if res then
14                  is_min[i] ⟵ false;
                end
            end
            else
                /* largest common prefix of S and D[i].                            */
15              p ⟵ max({0 ≤ j < min(|D[i]|, |S|) | D[i][1 : j] = S[1 : j]});
                /* note that the function Contains-Subset-Of-Memoized modifies the
                   node v.                                                         */
16              if Contains-Subset-Of-Memoized(v, i, p) then
17                  is_min[i] ⟵ false;
                end
            end
18          S ⟵ D[i];
        end
    end
19  return {D_i ∈ D | is_min[i] = true};
```

ALGORITHM 5: Pseudo code for finding the minimal itemsets M of the input dataset $D = \{D_1, D_2, \ldots, D_n\}$ using P threads. We present a subroutine Find-Min-Lex which identifies the minimal itemsets of D using P parallel threads. It is important to note that in the Thread-Functor subroutine the variables $index$ and is_min are passed by reference, meaning that they are shared between threads.

Input: Dataset $D = \{D_1, D_2, \ldots, D_n\}$ and the degree of parallelism P.
Output: The minimal itemsets within the dataset D. i.e. $Min(D)$.
Function `Get-Minimal-Itemsets-Lex-Parallel(`**dataset** D**,** **integer** P`)`

```
1   atomic < bool > is_min[r] ⟵ {true, true, ..., true};
    /* atomic boolean variables.                                          */
    /* Find itemsets subsumed by proper prefix.                           */
2   S ⟵ D[1];
3   for i = 2 to n do
4       if |S| ≤ |D[i]| & D[i][1 : |S|] = S then
            /* S is a proper prefix of D[i].                              */
5           is_min[i] ⟷ false;
        end
        else
6           S ⟵ D[i];
        end
    end
    /* Find itemsets subsumed by non-proper prefix using P parallel threads. */
7   atomic < int > index ⟵ 1;
    /* the index that is to be processed next.                            */
8   start P parallel instances of Thread-Functor (D, index, is_min);
9   wait for all P instances to finish working;
10  return {D_i ∈ D | is_min[i] == true};
```

Function `Thread-Functor(`**dataset** D**, atomic** $<$ **integer** $>$ $index$**, atomic** $<$ **bool** $>$ $m[r]$`)`

```
11  i ⟵ fetch-and-increment(index);
    /* an atomic operation                                                */
12  while i ≤ n do
        /* It is safe to invoke the function Contains-Subset-Of from multiple threads
           at the same time as it requires only read-only access to the dataset D.
           */
13      if Contains-Subset-Of (D[i + 1 : n], D[i], 1, 0) /* as per Algorithm 1 */
        then
            /* mark the i-th itemset as non-minimal because the dataset D contains a
               proper subset of the itemset D[i].                         */
14          m[i] ⟵ false;
            /* atomically setting the i-th boolean value.                 */
        end
15      i ⟵ fetch-and-increment(index);
    end
```