A Practical Algorithm for Finding Extremal Sets up to Permutation

MARTIN MARINOV, Trinity College Dublin
DAVID GREGG, Lero, Trinity College Dublin

In this paper we address the problem of finding extremal itemsets within a dataset F over a domain D up to any permutation of D. We present a parallel naive algorithm with novel search space pruning techniques. We present properties that must hold for an itemset A to be a subset of B up to permutation together with efficient method of checking if these conditions are satisfied; we also provide a method of reducing the number of permutation candidates π that need to be tested to check if $\pi(A) \subseteq B$. The experimental evaluation of our algorithm is performed on real world input datasets derived from the domain of sorting networks. The speedup factor achieved over the current state of the art method is in the region of one thousand for a $216KB$ dataset containing $1\,155$ itemsets. Although the worst case complexity of the presented algorithm is exponential, we demonstrate that our method is applicable to multi-gigabyte real world datasets containing more than ten million itemsets.

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, Extremal Sets up to Permutation, Dataset, Itemset

ACM Reference Format:
Martin Marinov and David Gregg, 2014. A Practical Algorithm for Finding Extremal Sets up to Permutation. *ACM J. Exp. Algor.* 9, 4, Article 39 (November 2014), 12 pages.
DOI: http://dx.doi.org/10.1145/0000000.0000000

1. INTRODUCTION

The problem of finding extremal sets (not up to permutation) has received significant attention in recent years [Fort et al. 2013], [Bayardo and Panda 2011], [Pritchard 1997]. In this paper we focus on a generalized version of this problem where we allow for any permutation of the domain to be applied to the elements of one itemset when checking if it is a subset of another, rather than directly performing the check. A practical application of this problem is found in the domain of sorting networks where Bundala and Zavodny [Bundala and Zavodny 2014] describe an algorithm for finding the minimal itemsets up to permutation within a dataset. The single-threaded implementation of our algorithm executes around $1\,000$ times faster than Bundala's one when evaluated on the same $216KB$ input dataset whilst producing the same output. We show that our multi-threaded program is able to solve multi-gigabyte input real-world datasets in less than 30 hours. In order to describe our work we must first give a precise definition of the problem together with terminology that is used throughout this paper.

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.

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

1.1. Terminology

Throughout the paper we assume that we are working over the domain $D = \{d_1, d_2, \ldots, d_n\}$. An *item* is a set of elements form D, an *itemset* is a set of items. The ordered set of itemsets $F = \{F_0, F_1, \ldots, F_{r-1}\}$ is called a *dataset* iff $|F_i| < |F_j|$ for all $0 \leq i < j < r$. The set of all permutations of n elements is denoted as Π_n. Let $\pi \in \Pi_n$ be a permutation, $I \subseteq D$ be an item and S be an itemset, then $\pi(I) = \{d_{\pi(i)} | d_i \in I\} \subseteq D$ and $\pi(S) = \{\pi(J) | J \in S\}$. Denote the number of elements within an itemset and dataset respectively as $||S|| = \sum_{I \in S} |I|$ and $||F|| = \sum_{S \in F} ||S||$. Let $a = \{a_1, \ldots, a_k\}$ and $b = \{b_1, \ldots, b_k\}$ be vectors over the integers domain; we say that $a < b$ iff the smallest index i for which $a_i \neq b_i$ implies that $a_i < b_i$; the $max(a, b) = \{max(a_1, b_1), \ldots, max(a_k, b_k)\}$.

1.2. Problem Definition

Let $F = \{F_0, F_1, \ldots, F_{r-1}\}$ be a dataset. We say that the itemset F_i is minimal over F iff there does not exist a $j > i$ such that $F_i \subseteq F_j$. Intuitively, we say that F_i is minimal over F up to permutation iff for all $\pi \in \Pi_n$ there does not exist a $j > i$ such that $\pi(F_i) \subseteq F_j$. In this paper we focus on the problem of finding all itemsets $F_i \in F$ which are minimal over F up to permutation; we denote this set as $Min_\pi(F)$.

1.3. Related Work

The extremal sets problem is well studied in recent years where Pritchard [Pritchard 1997] was the first person to present a sub-quadratic algorithm for the problem. Then Bayardo and Panda developed two distinct practical algorithms that build on top of Pritchard's approach. Bayardo and Panda [Bayardo and Panda 2011] take advantage of the frequency of every item by using precise ordering of the input to best suit their proposed solution. In our method for solving the extremal sets up to permutation problem we use their public provided implementation to initially reduce the size of the input dataset, and only then apply our new techniques. Fort [Fort et al. 2013] describes an algorithm for solving the extremal sets problem that is designed to run on a GPU. Although being quadratic, Fort's algorithm performs much faster than Bayardo and Panda's one according to the experimental results presented in [Fort et al. 2013].

The problem of finding sorting networks of minimal depth and related sub-problems are actively being researched in recent years [Bundala and Zavodny 2014], [Codish et al. 2014]. The problem of finding the minimal itemsets over the dataset F up to permutation is encountered by Bundala [Bundala and Zavodny 2014] when computing the representative comparator networks consisting of two layers. Bundala shows that every n-input comparator network C can be represented by the set of its outputs, denoted by outputs(C), which is an itemset over the domain $D = \{1, 2, \ldots, n\}$. He also shows that if A and B are both two layered n-input comparator networks and that $outputs(A)$ is a subset of $outputs(B)$ up to a permutation of D then it is enough for his algorithm to consider only the comparator network A. Hence, given the family of outputs (referred to as itemsets in this paper) of all n-input comparator networks of a depth 2 his algorithm needs to compute the minimal itemsets up to permutation. Bundala [Bundala and Zavodny 2014] describes a naive algorithm with pruning to solve this problem together with experimental results for his implementation. In our experiments (section 6) we find that our single-threaded algorithm is about $1\,000$ times faster on finding the minimal outputs of 13-input comparator networks consisting of exactly two layers.

Codish [Codish et al. 2014] presents a regular expression for the representative minimal saturated n-input comparator network of depth two. Given that his work is based on a conjecture which may or may not be true, the problem described in this paper still

has a valid application in the domain of sorting networks; furthermore the problem described in this paper is more general and the sorting network domain is only one practical application of our work.

1.4. Contributions

The main contributions of this work can be summarized as follows.

— *Search Space Pruning Techniques* — in this paper we present new search space pruning techniques that significantly reduce the search space size when compared to the naive approach. We present a method for detecting when an itemset cannot be a subset of another one without applying any permutations. We also present a method for reducing the number of permutation candidates that are to be tested.
— *Data Structure* — we present a binary tree data structure that allows us to efficiently apply the pruning techniques described in section 3.1 that test if one itemset can be a subset of another. This data structure proves to be very useful in the experiments conducted.
— *Parallel Algorithm* — we present a parallel algorithm for finding the minimal itemsets up to permutation within a dataset which can be easily adapted to find the maximal itemsets. Experiments are performed using real world datasets from the domain of sorting networks where substantial speedup is achieved over the existing state of the art method.

2. A NAIVE ALGORITHM

In this section we present a naive algorithm for finding the minimal itemsets up to permuting within a dataset. For every pair of itemsets F_i, F_j such that $0 \leq i < j < r$ we check if there exists a permutation $\pi \in \Pi_n$ such that $\pi(F_i) \subseteq F_j$. Note that we can restrict $i < j$ because it is a necessary condition that $|F_i| \leq |F_j|$ for F_i to be a subset of F_j up to permutation. The worst case space complexity is $\mathrm{O}\big(||F||\big)$ since we need to store only the input and no additional memory is required for performing calculations. The worst case runtime complexity of this algorithm is $\mathrm{O}\big(r \times n! \times ||F||\big)$, since for every pair F_i, F_j we need to apply all $n!$ permutations to F_i and check if $\pi(F_i) \subseteq F_j$. Checking if $\pi(F_i) \subseteq F_j$ takes $\mathrm{O}\big(||F_j||\big)$ computational steps as we can store the itemsets in F in ascending lexicographic order.

The practical parallel algorithm presented in this paper in section 5 is an optimized version of the naive algorithm. In the next section we present our novel search space pruning techniques which prove vital in the experimental evaluation presented in section 6.

3. SEARCH SPACE PRUNING

In this section we present search space pruning techniques for checking whether the itemset A is a subset of the itemset B up to permutation of the domain D. The safety checks can be logically grouped into two categories - the first one is aimed at checking whether a permutation candidate π can exist that could satisfy $\pi(A) \subseteq B$, whilst the second one is aimed at reducing the number of candidates $\pi \in \Pi_n$ for which we perform the subset up to permutation check. The necessary conditions described in this section are presented as lemmas. If a lemma can be applied to A and B then we say that A and B meet the necessary condition described by this lemma.

3.1. Necessary conditions for the itemset A to be a subset of B up to permutation

We present necessary conditions for checking whether the itemset A can be a subset of the itemset B up to permutation of the domain D. If any one of these conditions is not met by A and B then we can safely deduce that A is not a subset of B up to permu-

tation. The algorithmic method of applying these conditions in practice is described in section 4 where we present a data structure that allows us to check these conditions in a reduced number of operations compared to the naive approach.

The first necessary condition, presented in Lemma 3.2, is based on counting the number of items from the itemsets A and B that contain the element $d_i \in D$ for all $1 \leq i \leq n$. We then order these counts in ascending order to produce two non-decreasing sequences of numbers $a_1, \ldots, a_n$ and $b_1, \ldots, b_n$ for A and B respectively. We show that it is necessary that $a_i \leq b_i$ for all $1 \leq i \leq n$ to exist a permutation $\pi \in \Pi_n$ such that $\pi(A) \subseteq B$.

Definition 3.1. Let S be an itemset. Denote by $count_i(S)$ the set of items in the itemset S that contain the element d_i, formally $count_i(S) = \{I \in S \mid d_i \in I\}$ for $1 \leq i \leq n$. Let $A = \{A_1, \ldots, A_a\}$ be an itemset over the domain $D = \{d_1, \ldots, d_n\}$ then define the vector $C(A) = \langle C_1(A), \ldots, C_nA\rangle$ to be the sorted in ascending order vector of $\langle |count_1(A)|, \ldots, |count_n(A)|\rangle$.

LEMMA 3.2. *Let A and B be itemsets over the domain $D = \{d_1, \ldots, d_n\}$. If there exists a $j \in \{1, \ldots, n\}$ such that $C_j(A) > C_j(B)$ then $\nexists \pi \in \Pi_n$ such $\pi(A) \subseteq B$, i.e. A cannot be a subset of B up to permutation.*

PROOF. If $\pi(A) \subseteq B$ then the following inequality holds $|count_{\pi(i)}(A)| < |count_i(B)|$ for all $i \in \{1, \ldots, n\}$. Hence, if there exists a $j \in \{1, \ldots, n\}$ such that $C_j(A) > C_j(B)$ then there cannot exist a permutation π such that $\pi(A) \subseteq B$. □

The next condition, presented in Lemma 3.4, is based on the observation that applying any permutation π to any item I over D preserves the size of I, since π is bijective. Hence, the number of items from the itemset A that have a size equal to s must be less than or equal to the number of sets from B that have a size equal to s, for all $0 \leq s \leq n = |D|$.

Definition 3.3. Denote by $P_j(S)$ the set of items from the itemset S that consist of exactly j elements, formally $P_j(S) = \{I \in S \mid |I| = j\}$ for $1 \leq j \leq n$; define the vector $P(S) = \langle P_0(S), \ldots, P_nS\rangle$.

LEMMA 3.4. *Let A and B be itemsets over the domain $D = \{d_1, \ldots, d_n\}$. If there exists a $j \in \{0, \ldots, n\}$ such that $|P_j(A)| > |P_j(B)|$ then $\nexists \pi \in \Pi_n$ such $\pi(A) \subseteq B$, i.e. A cannot a subset of B up to permutation.*

PROOF. Since any permutation $\pi \in \Pi_n$ is bijective, applying π to any item I over D preserves the size of I; hence $|P_i(A)| = |P_i(\pi(A))|$ for any $i \in \{0, \ldots, n\}$. Hence, if $\exists j \in \{0, \ldots, n\}$ such that $|P_j(B)| > |P_j(A)| = |P_j(\pi(A)|$, then clearly $\pi(A)$ cannot be a subset of B up to permutation. □

The condition presented formally in the following lemma is a combination of the previous two results where we first group the items from the itemsets A and B by their cardinality and then apply Lemma 3.2. Since we know that permutations preserve the cardinality of items, we can restrict the input to Lemma 3.2 to $\{A_i \in A \mid |A_i| = s\}$ and $\{B_i \in B \mid |B_i| = s\}$ for some fixed $s \in \{0, 1, \ldots, n\}$, and if the necessary condition is not met then we can deduce that A can not be a subset of B up to permutation.

LEMMA 3.5. *Let A and B be itemsets over the domain $D = \{d_1, \ldots, d_n\}$. If there exists $i \in \{1, \ldots, n\}$ and $j \in \{0, 1, \ldots, n\}$ such that $C_i(P_j(A)) > C_i(P_j(B))$ then $\nexists \pi \in \Pi_n$ such that $\pi(A) \subseteq B$, i.e. A cannot a subset of B up to permutation.*

PROOF. Recall that applying $\pi \in \Pi_n$ to any item I over D preserves the size of I. Fixing $j \in \{0, \ldots, n\}$, we apply to Lemma 3.2 to $P_j(A)$ and $P_j(B)$ to show that if there

exists an $i \in \{1, \ldots, n\}$ such that $C_i(P_j(A)) > C_i(P_j(B))$ $\pi(A)$ cannot be a subset of B up to permutation. □

In the algorithm presented in this paper (section 5) we first pre-compute the respective counts required for checking if each of the above lemmas hold, i.e. $C(F_i)$, $P(F_i)$, $C(P(F_i))$. We then check the necessary conditions from Lemmas 3.2 and 3.4 because they are a factor of n computationally cheaper to check than the necessary condition described in Lemma 3.5 as the former require the comparison of $\mathrm{O}(n)$ integers and the latter requires $\mathrm{O}(n^2)$ computational steps.

3.2. Pruning Permutation Candidates $\pi \in \Pi_n$ for $\pi(A) \subseteq B$

The second group of search space pruning techniques is designed to reduce the number of permutations that need to be checked. Given that the itemset A can be a subset of B up to permutation according to the necessary conditions then we must check if a permutation π exists that would satisfy $\pi(A) \subseteq B$. We present novel techniques that are used to discard permutation candidates. The necessary conditions are designed to reject bad stubs in our recursive method for generating all feasible permutation candidates, presented in Algorithm 3. In all our experiments the total number of permutations performed was less than the total number of itemset pairs tested, see Figure 1.

Necessary Condition for $i \mapsto \pi(i)$, for $1 \leq i \leq n$. The first technique is based on the observation made in Lemma 3.2 that we need to consider only permutation candidates π that satisfy $C_i(A) \leq C_{\pi(i)}(B)$ for all $1 \leq i \leq n$, i.e. if $C_i(A) > C_{\pi(i)}(B)$ then we need not consider the permutation π as it is impossible for $\pi(A) \subseteq B$. This observation gives us a method of rejecting permutation candidates even without having to apply them to the itemset A.

Necessary Condition for $i \mapsto \pi(i)$ and $j \mapsto \pi(j)$, for $1 \leq i, j \leq n$. In this method for rejecting permutation candidates we check pairs of permuted elements. By the above observation we can safely assume that $C_i(A) \leq C_{\pi(i)}(B)$ and that $C_j(A) \leq C_{\pi(j)}(B)$. We can strengthen this criteria by observing that $|C_j(C_i(A))| < |C_j(C_i(B))|$ must hold, which is an immediate consequence of Lemma 3.5.

Algorithm for checking if A is subset of B up to permutation. The pseudo code of the method for checking if there exists a permutation $\pi \in \Pi_n$ that satisfies $\pi(A) \subseteq B$ is presented in Algorithm 3. The function Is-Subset-Up-To-Perm recursively generates all permutation candidates π that meet the two necessary conditions presented in this section. At line 10 we check if it is feasible for $\pi(index) = i$ for A to be a subset of B up to this π — our first search space reduction technique. The pruning technique for checking the feasibility of every pair of permuted elements of the domain D is encoded in our pseudo code at line 3 of Algorithm 3. For the worst case complexity analysis of the function Is-Subset-Up-To-Perm we assume that we have precomputed the sizes of the sets $C_i(A)$, $C_i(C_j(A))$, $C_i(B)$ and $C_i(C_j(B))$ for all $1 \leq i, j \leq n$ which requires a total of $\mathrm{O}(n^2)$ space. The worst case time and space complexity is achieved when all candidate $n!$ permutations meet the two necessary conditions that we have described. In the worst case scenario the function Is-Subset-Up-To-Perm(A, B) requires space $\mathrm{O}(||A|| + ||B|| + n^2) = \mathrm{O}(||B|| + n^2)$ since $||A|| < ||B||$. The worst case time complexity is $\mathrm{O}(n! \times ||B||)$ computational steps because after having applied a permutation we use a trivial subset checking method that requires $\mathrm{O}(||A|| + ||B||)$ computational steps.

4. DATA STRUCTURE

In this section we describe a method to apply in practice the necessary conditions described in section 3.1. We have designed a data structure to efficiently find all itemsets

$F_j \in F$ which can be supersets of F_i up to permutation that meet the necessary condition Lemma 3.2. In our Algorithm 4 we then take the resulting list of candidates and apply Lemmas 3.5 and 3.4 to efficiently produce a list of candidates which meet all the necessary conditions described in section 3.1.

Goal. Given the dataset $F = \{F_0, F_1, \ldots, F_{r-1}\}$ and a fixed itemset F_i, our goal is to efficiently find all F_j where $i < j < r$ such that F_i can be a subset of F_j up to permutation using the necessary condition described by Lemma 3.2. One obvious way to achieve this goal is to apply the necessary condition check to all F_i and F_j for all $i < j < r$ but this is not an efficient way to solve this problem in practice.

Observation. Using the result from Lemma 3.2, we see that $C(M) < C(A)$ is necessary for M to be a subset of A up to permutation, and similarly that $C(M) < C(B)$ for M to be a subset of B up to permutation. We notice that $C(M) < max(C(A), C(B))$ is a necessary condition for M to be a subset of A or B up to permutation; i.e. if $C(M)$ is not smaller than $max(C(A), C(B))$ then M cannot be a subset of A up to permutation and M cannot be a subset of B up to permutation.

4.1. Initialization

Algorithm. We propose a binary tree data structure consisting of $r = |\{F_0, F_1, \ldots, F_{r-1}\}|$ leaves to efficiently achieve our goal. At the i-th leaf we store the vector $C(F_i)$, as described in Lemma 3.2. The value stored in a node v that is a parent of the leaves in the range $[i, j]$ is defined as the vector $v_c = max(C(F_i), C(F_{i+1}), \ldots, C(F_j))$. Noticing that taking the maximum across multiple vectors is a commutative operation, when initializing the binary tree data structure, for each node we need to know only the values of its immediate children. Hence, in the pseudo code presented in Algorithm 1 we use a bottom-up approach to initialize the data structure.

Complexity. The function Initialize has a worst case space complexity of $\mathrm{O}(r \times n)$ because the tree has $\mathrm{O}(r)$ nodes and each node contains a vector of integers of length exactly n and we require no extra space to compute the vectors stored at each tree node. The time complexity of the Initialize(F) depends on the computation of $C(F_i)$ for each of the r leaves of the tree which takes $\mathrm{O}(||F_i||)$ computational steps per leaf (itemset). Hence, the worst case time complexity of the Initialize(F) function is $\mathrm{O}(||F|| + r \times n)$ where $||F|| = ||F_0|| + ||F_1|| + \cdots + ||F_{r-1}||$.

4.2. Query

Algorithm. The binary tree data structure is designed to be queried with a fixed itemset F_i to efficiently find all itemsets F_j for which $C(F_i) < C(F_j)$ and $i < j$. The pseudo code presented in Algorithm 2 of the query function performs a top-down recursive walk of the binary tree, starting at the root node. Let the current node in the recursive walk be denoted as v that is a parent of the leaves in the range $[l, r]$ and its value vector be v_c, as described in the initialization paragraph. Since we want all $j > i$, we can safely terminate the search if $r < i$. Then, based on our observation, we can safely say that if $C(F_i)$ is not smaller than v_c then F_i can not be a subset of F_k up to permutation for all $k \in [l, r]$, hence we do not need to recursively visit any of the children of v. On the other hand, if $C(F_i)$ is smaller than or equal to v_c then we need to recursively visit both children of v. If v has no children, i.e. v is a leaf, then we add it to the collection of itemsets that can be supersets of F_i up to permutation, as per the necessary condition described in Lemma 3.2.

Concurrency. It is important to note that the binary tree data structure T is immutable after initialization. Hence, we can make parallel function calls to Query(T, A) and Query(T, B) without having to use any locks.

Complexity Analysis. The number of nodes in the binary tree T that are initialized with the dataset $F = \{F_0, F_1, \ldots, F_{r-1}\}$ is $\mathrm{O}(r)$, as discussed in the complexity analysis of the Initialize(F) method. In the worst case, the function Query-Rec visits all of the r leaves of the tree, and hence all of the nodes in the tree. In the worst execution path of Query-Rec we perform a comparison of two vectors of length n, hence we have a time complexity of $\mathrm{O}(r \times n)$ for Query-Rec. The space required by the function Query is proportional to the size of the output as we require only $\mathrm{O}(1)$ extra intermediate integer variables to perform the required calculations. Since the maximum number of itemsets that the function Query can return is r then we can deduce that the worst-case space complexity of the Query function is $\mathrm{O}(||F||)$, i.e. the space required by the Query function is asymptotically equal to the space required to store the input dataset F.

4.3. Extensions

We can easily extend our data structure to efficiently retrieve all itemsets that satisfy the conditions from Lemmas 3.2 and 3.4. To achieve this task we need to additionally store the vector $P(F_i)$ as described in Lemma 3.4 in the tree leaves and nodes. Following the same logic in the initialization process, in every node we need to store the maximum P-vector. And as for querying, we would only visit a node if both conditions — from Lemmas 3.2 and 3.4 — are met. A similar argument can be constructed for including the information required to check the necessary condition from Lemma 3.5 but in this case it would require a two dimensional vector to be stored in every node. We found that these extensions to the data structure are not necessary for input datasets derived from the domain of sorting networks.; i.e. for other domains these proposed extensions could actually prove more useful, as might the order in which they are applied.

5. PARALLEL ALGORITHM

So far we have presented necessary conditions for checking whether the itemset A can be a subset of B up to permutation. We have described a data structure to efficiently check these conditions. We have also presented a practical method for checking if A is a subset of B up to permutation. In this section we describe a parallel algorithm which identifies the minimal up to permutation itemsets within the dataset $F = \{F_0, F_1, \ldots, F_{r-1}\}$. We first describe the methodology of the entry point of the algorithm, then the thread worker functor and finally we present worst case complexity analysis. The pseudo code of the parallel algorithm described in this section is presented in Algorithm 4.

5.1. Entry Point

The function Find-Min-Rep-Perm takes as input the dataset F and the degree of parallelism P as an integer. First the binary tree data structure T is initialized as described in section 4. Then every itemset in the dataset F is marked as minimal. We then start P parallel instances of the function Thread-Functor to process every itemset $F_{index} \in F$ and mark all of the supersets of F_{index} as non-minimal. When all threads have finished processing, the function simply returns all sets that are still marked as minimal.

5.2. Thread-Functor

The variable $index$ is shared between all instances of the Thread-Functor; it points to the next not processed itemset $F_{index} \in F$. To process F_{index} means to check for every $j > index$ if F_{index} is a subset of F_j up to permutation and mark F_j as non-minimal if this is the case. We begin by atomically assigning the current value of $index$ to the variable i and decrementing $index$; ensuring that every itemset in F will be processed exactly once. We then query the binary tree data structure T to find the list of candidate supersets up to permutation $D = \{F_j \in F | j > i$ and F_i, F_j meet necessary condition Lemma 3.2 $\}$. Next, we filter the set of candidates D by applying the necessary conditions from Lemmas 3.5 and 3.4 to F_i and $F_j \in D$ to further reduce the number of superset candidates. Then, we check for each candidate $F_j \in D$ if F_i is a subset of F_j up to permutation using Algorithm 3; if the result is positive we mark F_j as non-minimal in the shared across threads is_min collection. Finally, we try to take a new unprocessed itemset from F and process it in the same manner.

Noticing that the "subset up to permutation" operation $\leqq_\pi$ is transitive; i.e. $A \leqq_\pi B$ and $B \leqq_\pi C$ implies that $A \leqq_\pi C$, when processing F_i we can check at any point whether it is still marked as minimal, and if not then we can safely stop processing it and move on to the next non processed itemset. Hence, if $B = F_i$ is marked as non-minimal during the processing of some other itemset A (possible in a different thread) then any itemset that is a superset of B up to permutation is also a superset of A up to permutation. i.e. the operation $\leqq_\pi$ induces a partial ordering of the dataset F.

5.3. Complexity

Here we give the worst case time and space complexity of the functions presented in Algorithm 4 by first analysing the Thread-Functor function. From section 3.2 we know that the worst case time complexity of the function Is-Subset-Up-Perm(A, B) is $\mathrm{O}\big(n! \times ||B||\big)$ and from section 4 we know that the worst case time complexity of the Query function is $\mathrm{O}\big(r \times n\big)$. Assuming that the Thread-Functor processes p itemsets, we know that for each one we will invoke the Query function exactly once and the Is-Subset-Up-To-Perm function at most r times then we can deduce that the worst case runtime complexity of Thread-Functor is $\mathrm{O}\big(p \times n! \times ||F||\big)$. Which means that if we have a degree of parallelism $P = 1$ then $p = r$ and the worst case time complexity of our algorithm is equal to that of the naive algorithm described in section 2. The space required by the function Thread-Functor is equal to the sum of the space required by the Query and Is-Subset-Up-Perm. Since the function Is-Subset-Up-Perm requires the sets $C_i(S)$, $C_i(C_j(S))$ to be precomputed for every $S \in F$ we deduce the space required by this function is $r \times n^2$. Hence the worst case space required by Thread-Functor is equal to $\mathrm{O}\big(||F||\big) + \mathrm{O}\big(r \times n^2\big) = \mathrm{O}\big(||F|| + r \times n^2\big)$.

Having analysed the Thread-Functor function, we now focus on the worst case time complexity of the Find-Min-Rep-Perm function for the input dataset $F = \{F_0, F_1, \ldots, F_{r-1}\}$ using exactly P threads where $1 \leq P \leq r$. Since we already know that the maximum time required by a Thread-Functor to process p itemsets is $\mathrm{O}\big(p \times n! \times ||F||\big)$ we can distribute the work load to the P threads evenly, meaning that each one should process $p = \mathrm{O}\big(r/P\big)$ itemsets. Hence the worst case time complexity for finding the minimal itemsets of F using P threads is $\mathrm{O}\big(\frac{r \times n! \times ||F||}{P}\big)$. The worst case space required by the Find-Min-Rep-Perm function is equal to the sum of the space required to initialize the binary tree data structure T and the space required by the Thread-Functor function. That is Find-Min-Rep-Per requires $\mathrm{O}\big(||F|| + r \times n^2\big)$ space in the worst case.

$r = \|F\|$	$\|Min_\pi(F)\|$	$SubTests$	$TotPerms$	$PosPerms$	$NegPerms$	$RunTime$	n	d
113	78	199	74	74	0	0.2 seconds	11	2
103 168	180	190 114	111 093	111 093	0	6 seconds	13	2
570 758	104 667	13 873 784	489 828	487 278	2 550	60 seconds	11	3
10 566 574	3 450 474	12 474 224 514	7 741 099	7 597 114	143 985	29 hours	13	3

Fig. 1. Experimental evaluation summary on real world datasets obtained from the domain of sorting networks. The dataset F for n-input comparator network of depth d is generated by applying network levels with maximal number of comparators to the set of minimal itemsets up to permutation of depth $d-1$. The column $SubTests$ presents the total number of calls to the function Is-Subset-Up-To-Perm. The column $TotPerms$ describes the total number of permutations and the columns $PosPerms$ and $NegPerms$ present the breakdown of permutations performed when the outcome of the function Is-Subset-Up-To-Perm was positive('true') and negative('false') respectively.

6. EXPERIMENTAL EVALUATION

6.1. Environment Setup

In all of the conducted experiments we used a computer with four Intel Xeon CPU E7- 4820 processors. Each CPU has 8 cores clocked at 2.00GHz, equipped with 8MB of third level cache and 128GB of main memory. Note that our experiments investigate the case when the entire data structure fits in main memory.

6.2. Real World: Sorting Networks

The inspiration for developing the algorithm presented in this paper is derived from the problem of finding optimal sorting networks. Bundala's method needs to find the minimal representative up to permutation itemsets within a dataset. Hence, we need to compare our method to the existing state of the art approach for solving the same problem [Bundala and Zavodny 2014].

Comparison to Bundala's approach, $n = 13, d = 2, r = 1\,155$. Bundala only reports the runtime of his implementation for finding the minimal up to permutation saturated layers of a thirteen input network. In his work, it is not mentioned whether the implementation of their program is parallel or not, so we assume that it is not parallel and compare its reported runtime to a single threaded implementation of our algorithm. We executed our program against the same input of $1\,155$ itemsets (refereed to as outputs of saturated layers in [Bundala and Zavodny 2014]). Our program computes the 212 representative up to permutation itemsets in under 2 seconds using a degree of parallelism $P = 1$, whereas Bundala's program takes 32 minutes [Bundala and Zavodny 2014]. Hence, for this input, our non-parallel implementation is about $1\,000$ times faster than Bundala's reported runtime over the same input whilst producing the same output.

Test Data Generation. Since our algorithm outperforms the current state of the art, we evaluate our program on much bigger datasets containing more than $1\,155$ itemsets by generating real world inputs derived from n-input sorting networks, for $n = 11$ and 13. In order to generate the input datasets, we first compute the set of maximal levels for an n-input comparator network, i.e. the set of levels which have exactly $\lfloor n/2 \rfloor$ comparators. An *output* of a comparator network is defined as an itemset over the domain $D = \{1, 2, \ldots, n\}$. Our input datasets are generated by computing the outputs of all n-input comparator networks with exactly d maximal levels, for some positive d. Summary of all conducted experiments is presented in Figure 1, where every row is uniquely identified by n and d which follows the data generation process described in this paragraph.

Evaluation. In Figure 1 we have presented a summary of the conducted experiments. Note that after generating all maximal levels for n-input comparator network

of depth d using the minimal up to permutation ones of depth $d-1$, we first apply Bayardo and Panda's [Bayardo and Panda 2011] algorithm for identifying the minimal itemsets (not up to permutation) to arrive at the dataset F. The execution time of Bayardo and Panda's program is insignificant in comparison to the $RunTime$ of our algorithm, as well as usage of memory and other computer resources. This step reduces the size of the input dataset by about 17% on average in all test cases.

The counters presented in Figure 1 demonstrate the 'goodness' factor of the search space pruning techniques presented in this paper. Recall, that the necessary conditions described in section 3 are logically grouped into two categories. The first logical group is aimed at the existence of a permutation π that could satisfy $\pi(A) \subseteq B$ for some itemsets A and B; i.e. if one of these conditions is not satisfied then A can not be a subset of B up to permutation. Column $SubTests$ in Figure 1 gives the total number of itemset pairs A and B that meet all of the necessary conditions from section 3.1. Comparing the total number of subset tests performed $SubTests$ to that of the naive approach is a good indicator of the speedup achieved by our approach; i.e. looking at row $n = 13, d = 2$ we see that the dataset contains $r = 103\,168$ itemsets and out of the possible $\frac{r \times (r-1)}{2} = 5\,321\,766\,528$ subset up to permutation tests, our algorithm performed only $190\,114$, meaning that this group of search space reduction techniques give us a speedup factor of approximately $27\,992$ in comparison to the naive approach. Similarly, for row $n = 13, d = 3$ we can deduce that this set of necessary conditions are expected to give our algorithm a speedup factor of around $4\,475$ times compared to the naive method (described in section 2).

The second logical group of search space reduction techniques presented in section 3.2 is aimed at reducing the number of permutation candidates π for testing $\pi(A) \subseteq B$. The naive algorithm for checking if A is a subset of B up to permutation would check all $n!$ permutations for every pair of itemsets A and B. Given that our algorithm needs to check $SubTests$ number of itemset pairs, then we deduce that the naive approach would check a total of $n! \times SubTests$ permutations. The column $TotPerms$ in Figure 1 gives the total number of permutations that meet all of the necessary conditions described in section 3.2, i.e. that is an execution counter at line 6 of Algorithm 3. Note that in all conducted experiments the total number of permutations tested $TotPerms$ is less than $SubTests$, let alone $n! \times SubTests$ which is the case for the naive approach. Hence, the expected speedup factor that this group of search space pruning techniques gives in comparison to the naive method is approximately $n!$. The columns $PosPerms$ and $NegPerms$ represent the breakdown of the number of permutations that are tested which satisfy the subset up to permutation property and the ones which did not satisfy the property, respectively. Another important observation on the 'goodness' factor of our techniques is that the number $NegPerms$ is negligibly small in comparison to the total permutations performed $TotPerms$ in all experiments conducted. Which means that almost every permutation that we test does satisfy the subset up to permutation property.

Since our program is multi-threaded and from two distinct threads we can find that A is a subset of C up to permutation and that B is subset of C up permutation at the 'same' time, and mark it as non-minimal we note that $PosPerms > r - |Min_{\pi}(F)|$. Hence, our multi-threaded algorithm performs a bit more work than the single threaded one, but it compensates by producing the correct result much faster in terms of wall-clock time. When executed on a machine with 32 physical cores using a degree of parallelism of $P = 32$ our program is on average 23 times faster than the single threaded version (no parallelism); not achieving a speedup factor of P is expected due to synchronization of the executing threads (implemented using locks), the operating system as well as other factors. We also performed a small set of experiments on a

machine with 4 physical cores to get a speedup factor of 3 when compared to the single threaded version. Hence, our program is about $0.7 * P$ times faster in terms of wall-clock time when compared to the single threaded version, where $P > 1$ denotes the degree of parallelism.

Summary. To summarize the 'goodness' of both groups of search space pruning techniques we present a detailed analysis on the dataset $n = 13, d = 3$ which is easily adoptable to the any other dataset from Figure 1. The first set of search space reduction techniques has an expected speedup factor of $\frac{r \times (r-1)}{2 \times SubTests} \approx 4\,475$ over the naive method. The second set of techniques has an expected speedup of $\frac{n! \times SubTests}{TotPerms} \approx 10^{13}$ over the naive algorithm. Our algorithm combines the two techniques, hence the expect speedup factor is close to $4\,475 * 10^{13} \approx 4 \times 10^{16}$ over the naive method. For this dataset our algorithm performed $PosPerms - (r - |Min_{\pi}(F)|) \approx 481\,000$ more subset up to permutation tests than necessary due to multi-threading.

7. CONCLUSION

A new parallel algorithm for identifying minimal itemsets within a dataset up to permutation is presented in this paper which applies various novel search space reduction techniques. The presented properties are aimed at detecting when an itemset cannot be a subset of another up to permutation of the domain D and in reducing the number of permutations of D that are tested. We present a binary tree data structure that allows us to efficiently check one of the necessary conditions that is easily extendible to capture all properties if required. A runtime speedup factor of around $1\,000$ is observed in comparison to the current state of the art method for solving the same problem on the same $216KB$ input dataset. The algorithm is further evaluated against real world data from the domain of sorting networks demonstrating that our approach can solve multi-gigabyte input datasets in less than 30 hours on a 32-core machine. The experimental evaluation presents analysis of execution counters measuring the significance of the search space reduction techniques described.

ACKNOWLEDGMENTS

The author would like to thank Andrew Anderson.

REFERENCES

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

Daniel Bundala and Jakub Zavodny. 2014. Optimal Sorting Networks. In *Language and Automata Theory and Applications - 8th International Conference, LATA 2014, Madrid, Spain, March 10-14, 2014. Proceedings (Lecture Notes in Computer Science)*, Adrian Horia Dediu, Carlos Martín-Vide, José Luis Sierra-Rodríguez, and Bianca Truthe (Eds.), Vol. 8370. Springer, 236–247.

Michael Codish, Luís Cruz-Filipe, and Peter Schneider-Kamp. 2014. The Quest for Optimal Sorting Networks: Efficient Generation of Two-Layer Prefixes. *CoRR* abs/1404.0948 (2014). http://arxiv.org/abs/1404.0948

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

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

ALGORITHM 1: Pseudo code of the function $Initialize(F)$ for initializing the binary tree data structure T described in section 4.

Input: Dataset $F = \{F_0, F_1, \ldots, F_{r-1}\}$ over the domain $D = \{d_1, d_2, \ldots, d_n\}$.
Output: The function $Initialize(F)$ returns a binary tree T which is used to efficiently query the data structure and find for a fixed F_i all F_j such that $j > i$ and F_i can be a subset of F_j up to permutation according to the necessary condition given by Lemma 3.2.

Function `Initialize`(**dataset** $F = \{F_0, F_1, \ldots, F_{r-1}\}$)

1 $leaves\_count \longleftarrow$ Least-Power-of-Two-Not-Smaller-Then(r);
2 $nodes\_count \longleftarrow leaves\_count * 2$;
`/* we represent the binary tree data structure as an array */`
3 $T.nodes \longleftarrow Node[nodes\_count]$;
`/*` $Node$ `is a vector of` n `elements, initialized with vectors of zeros */`
4 $T.root \longleftarrow 1$;
`/* The root is stored in` $Node[1]$ `*/`
5 $i \longleftarrow 0$;
6 **repeat**
7 $T.nodes[leaves\_count + i] \longleftarrow C(F_i)$;
8 `/* as described in Lemma 3.2 */`
9 $i \longleftarrow i + 1$;
until $i < r$;
10 $i = leaves\_count - 1$;
11 **repeat**
12 $T.Nodes[i] \longleftarrow max(T.nodes[i * 2], T.nodes[i * 2 + 1])$;
13 `/* the function` max `is defined in section 1.1 */`
14 $i \longleftarrow i - 1$;
until $T.root \leq i$;
15 **return** T;

ALGORITHM 2: Pseudo code for the function $Query(T, F_i)$ for querying the binary tree data structure T as described in section 4.

Input: An binary tree data structure T that is initialized using Algorithm 1 over the dataset F and an itemset $F_i \in F$, where $F = \{F_0, F_1, \ldots, F_{r-1}\}$.

Output: The function $Query(T, F_i)$ efficiently retrieves all F_j such that $i < j < r$ and F_i can be a subset of F_j up to permutation of D using the necessary condition from Lemma 3.2.

```
1  Function Query(binary tree T, itemset F_i)
2      return Query-Rec((T, F_i, T.root, 0, r));
3  Function Query-Rec(binary tree T, itemset F_i, integer node_index, integer l, integer r)
4      if r ≤ i then
5          return ∅;
       end
6      if T.nodes[node_index] < C(F_i) then
           /* the necesarry condition from Lemma 3.2 is not met for F_i and any of the
              leaves ( F_l, F_{l+1}, ..., F_{r-1} ) that are children of the node
              T.nodes[node_index]. */
7          return ∅;
       end
8      if l == r then
9          return {F_l};
       end
10     left_child ⟵ node_index * 2;
11     right_child ⟵ node_index * 2 + 1;
12     mid ⟵ ⌊(l + r)/2⌋;
13     return Query-Rec((T, F_i, left_child, l, mid)) ⋃ Query-Rec((T, F_i, right_child, mid, r));
       /* recursively traverse the left and right children of the current node. */
```

ALGORITHM 3: Pseudo code of the function Is-Subset-Up-To-Perm(A, B) for checking if the itemset A is subset of B up to permutation by recursively generating all permutation candidates π that meet the necessary conditions described in section 3.2.

Input: The itemsets A and B.
Output: The function Is-Subset-Up-To-Perm-Rec(A, B) returns $true$ iff there exists a permutation $\pi \in \Pi_n$ such that $\pi(A) \subseteq B$.

```
Function Is-Subset-Up-To-Perm(itemset A, itemset B)
1     return Is-Subset-Up-To-Perm-Rec (A, B, 1, π);
Function Is-Subset-Up-To-Perm-Rec(itemset A, itemset B, integer index, permutation π)
2     i_1 ⟵ index − 1;
3     if ∃i ∈ {1, ..., i_1 − 1} : |C_i(C_{i_1}(A))| < |C_i(C_{i_1}(B))| then
          /* Section 3.2                                                    */
4         return false;
      end
5     if index == n + 1 then
6         if π(A) ⊆ B then
7             return true;
          end
8         return false;
      end
      for i ⟵ 1 to n do
9         if ∄j ∈ {1, ..., index − 1} : π(j) == i then
              /* π must be bijective                                        */
10            if C_index(A) ≤ C_i(B) then
                  /* Section 3.2                                          */
11                π(index) ⟵ i;
                  if Is-Subset-Up-To-Perm-Rec(A, B, index + 1, π) then
12                    return true;
                  end
              end
          end
      end
13    return false;
```

ALGORITHM 4: Pseudo code for finding the minimal up to permutation itemsets M of the input dataset $F = \{F_0, F_1, \ldots, F_{r-1}\}$ using T threads, where every $F_i \in F$ is an itemset over the domain $D = \{d_1, d_2, \ldots, d_n\}$. We present a subroutine Find-Min-Rep-Perm which identifies the minimal representative itemsets of F using T parallel threads. It is important to note that in the Thread-Functor subroutine the variables $index$ and is_min are passed to the by reference, meaning that they are shared between threads.

Input: Dataset $F = \{F_0, F_1, \ldots, F_{r-1}\}$ over the domain $D = \{d_1, d_2, \ldots, d_n\}$ and the degree of parallelism P
Output: The minimal itemsets within the dataset F up to permutation of D. i.e. $Min_\pi(F)$
Function `Find-Min-Rep-Perm`(**dataset** $F = \{F_0, F_1, \ldots, F_{r-1}\}$, **integer** P)
`/* note that` F `is ordered in non-decreasing cardinality order s.t. if` $i < j$ `then` $|F_i| \leq |F_j|$ `*/`
1 $T \longleftarrow$ Initialize(F);
`/* binary tree data structure, as per Algorithm 1 */`
2 $atomic < bool > is\_min[r] \longleftarrow \{true, true, \ldots, true\}$;
`/* atomic boolean variables */`
3 $atomic < int > index \longleftarrow 0$;
`/* the index that is to be processed next */`
4 start T parallel instances of Thread-Functor($F, index, is\_min$);
5 wait for all T instances to finish working;
6 **return** $\{F_i \in F \mid is\_min[i] == true\}$;
Function `Thread-Functor`(**dataset** $F = \{F_0, F_1, \ldots, F_{r-1}\}$, **atomic** $<$ **integer** $>$ $index$, **atomic** $<$ **bool** $>$ $is\_min[r]$)
7 $i \longleftarrow$ fetch-and-increment($index$) `/* an atomic operation */`
8 **repeat**
9 $C \longleftarrow$ Query(T, F_i);
`/* get all candidate supersets of` F_i `as per Algorithm 2 */`
10 $C \longleftarrow \{F_j \in C \mid F_i$ and F_j meet the necessary conditions from Lemmas 3.5 and 3.4 $\}$;
11 **forall the** $F_j \in C$ **do**
12 **if** $is\_min[i]$ **then**
`/* atomically check if` F_i `is still minimal */`
13 **if** $is\_min[j]$ **then**
`/* atomically fetching the` F_j`-th boolean value */`
14 **if** *Is-Subset-Up-To-Perm*(F_i, F_j) **then**
`/* Algorithm 3 */`
15 $is\_min[j] \longleftarrow false$;
`/* atomically setting the` j`-th boolean value */`
end
end
end
end
16 $i \longleftarrow$ fetch-and-increment($index$);
until $i < r$;