

The Subset Partial Order: Computing and Combinatorics

Amr Elmasry *

Max-Planck Institut für Informatik
Saarbrücken, Germany
elmasry@mpi-inf.mpg.de

Abstract

Given a family \mathcal{F} of k sets with cardinalities s_1, s_2, \dots, s_k and $N = \sum_{i=1}^k s_i$, we show that the size of the partial order graph induced by the subset relation (called the subset graph) is $O(\sum_{s_i \leq B} 2^{s_i} + N/\log N \cdot \sum_{s_i > B} \log(2s_i/B))$, where $B = \log(N/\log^2 N)$. This implies a simpler proof to the $O(N^2/\log^2 N)$ bound concluded in [2].

We also give an algorithm that computes the subset graph for any family of sets \mathcal{F} . Our algorithm requires $O(nk^2/\log k)$ time and space on a pointer machine, where n is the number of domain elements. When \mathcal{F} is dense, i.e. $N = \Theta(nk)$, the algorithm requires $O(N^2/\log^2 N)$ time and space. We give a construction for a dense family whose subset graph is of size $\Theta(N^2/\log^2 N)$, indicating the optimality of our algorithm for dense families. The subset graph can be dynamically maintained when \mathcal{F} undergoes set insertion and deletion in $O(nk/\log k)$ time per update (that is sub-linear in N for the case of dense families). If we assume words of $b \leq k$ bits, allow bits to be packed in words, and use bitwise operations, the above running time and space requirements can be reduced by a factor of $b \log(k/b+1)/\log k$ and $b^2 \log(k/b+1)/\log k$ respectively.

1 Introduction

The graph induced by the subset relation on a family of sets is called the subset partial order or the subset graph for this family. The problem of computing the subset graph of a family of sets is a practical problem that arises in many applications. One such application is in propositional logic: given a formula in restricted Conjunctive-Normal-Form, we need to simplify the formula by removing each conjunct that has a strict superset of the propositional variables of another conjunct. See for example [5].

A natural lower bound for computing the subset graph is its size. It is also theoretically interesting to find upper and lower bounds on the size of the subset graphs. Yellin and Jutla [8] gave a construction of a family whose subset graph has $\Theta(N^2/\log^2 N)$ size. Later, Pritchard [2] proved that the size of the subset graph is $O(N^2/\log^2 N)$. In this paper, we give a simpler proof for this fact. We also give other bounds that

illustrate the structure of the subset graphs, and bound the size of the subset graphs for some special families.

Several algorithms for computing the subset graphs were introduced. An algorithm that runs in $O(Nd)$ time is given in [9] and refined in [4], where d is the maximum degree (number of sets) of an element in \mathcal{F} . Pritchard [3, 4] gave a couple of algorithms that run in $O(N^2/\log N)$ time. He also introduced yet another $O(N^2/\log N)$ -time algorithm [2] by modifying the algorithm in [8]. However, it is still open whether it is possible or not to compute the subset graph in $O(N^2/\log^2 N)$ comparisons.

Lacking for a general $O(N^2/\log^2 N)$ algorithm, the problem is tackled for some special cases as well. When there is a constant that bounds the cardinality of each set, an algorithm for computing the $O(N)$ subset graph in $O(N \log N)$ time was given in [5]. This algorithm can be shown to be optimal following a reduction from the Element-Uniqueness problem. Caveat, such algorithm requires random access to arrays of size k .

A dense family is defined as a family with $N = \Theta(nk)$. Dense families are natural in practice; consider, for example, the case when every element is expected to show up in each set with some constant probability. In this paper, we give an algorithm to compute the subset graph for a dense family that requires $O(N^2/\log^2 N)$ time and space. We also give a construction for a dense family whose subset graph has $\Theta(N^2/\log^2 N)$ size (the construction in [8] is not for a dense family), implying the optimality of our algorithm.

Allowing for stronger models of computation (where even the $N^2/\log^2 N$ barrier could be broken), better bounds are possible. On a RAM permitting bitwise operations on words of $\Theta(\log N)$ bits, an algorithm using $O(N^2 \log \log N/\log^2 N)$ operations was given in [1]. On such model, considering normal families (dense families with the additional property that a positive proportion of sets each has at least $\log N$ elements), Shen and Evans [7] gave an algorithm that uses $O(N^2/\log^2 N)$ bitwise operations and $O(N^2/\log^3 N)$ words of space.

*Supported by Alexander von Humboldt Fellowship.

Once performing our algorithm on such model, it uses $O(N^2/\log^3 N)$ bitwise operations and $O(N^2/\log^4 N)$ words of space when applied to a dense family, improving over [7] and breaking the $N^2/\log^2 N$ bound.

The problem of maintaining the extremal sets of \mathcal{F} under dynamic updates (set insertion, set deletion, and set-content update) is also of interest. Shen [6] gave an algorithm that uses $O(N + nk/\log N + k^2)$ words of space, processes in $O(1)$ time a query on whether a set of \mathcal{F} is extremal, and maintains all extremal sets of \mathcal{F} in $O(N)$ time per update. On the other hand, our algorithm is extended to maintain the subset graph and allow dynamic updates in $O(nk/\log k)$ time per update; for the case of dense families, our algorithm then requires $O(N/\log k)$ time per update. Allowing for bitwise operations as above, with words of $\Theta(\log N)$ bits, an extra $\log N$ factor can even be saved.

In Section 2 we give a proof that the size of the subset graph is $O(N^2/\log^2 N)$, and introduce a family of dense sets whose subset graph is of size $\Theta(N^2/\log^2 N)$, indicating that this is a lower bound for computing the subset graph for dense families in the comparison-based model. In Section 3 we introduce the algorithm, analyze it on a pointer machine as well as in a stronger computational model, and show how to extend it to consider dynamic set updates.

2 The combinatorics of the subset graph

We warm up with the following lemma, which implies an $O(N^2/\log N)$ bound on the size of the subset graph. Although this is a weaker bound than the tight bound we establish later in Lemma 2.2, the next lemma is interesting in its own right.

LEMMA 2.1. *Except for $O(N^\epsilon)$ sets, where ϵ is any constant $0 < \epsilon < 1$, the number of supersets of any other set is $O(N/\log N)$.*

Proof. The degree of an element is the number of sets to which it belongs. Consider the $\nu = \lfloor \epsilon \cdot \log N \rfloor$ elements with the largest degrees (we call these elements the *high-degree* elements, and call the others the *low-degree* elements). Let $\mathcal{F}' \subset \mathcal{F}$ be the subfamily having none of the low-degree elements in any of its sets. Then, $|\mathcal{F}'| \leq 2^\nu = O(N^\epsilon)$. Any other set $S \in \mathcal{F} - \mathcal{F}'$ must contain at least one of the low-degree elements; call this element e . All the supersets of S must as well contain e , and hence their count is bounded by e 's degree d_e . Since d_e is less or equal to the degree of any of the high-degree elements, then $d_e < N/\nu = O(N/\log N)$.

The next lemma contains a simpler proof, than that in [2], to the fact that the size of the subset graph is $O(N^2/\log^2 N)$.

LEMMA 2.2. *The size of the subset graph for a family of sets, as induced by the subset relation, is $O(\sum_{s_i \leq B} 2^{s_i} + N/\log N \cdot \sum_{s_i > B} \log(2s_i/B)) = O(N^2/\log^2 N)$, where $B = \log(N/\log^2 N)$.*

Proof. We group the sets into two categories according to whether their sizes are larger than B or not, and bound the number of subsets for each category. Let w be the number of subsets of a set S whose size is s .

If $s \leq B$, we use the fact that $w \leq 2^s$ to bound the number of subsets of S . Hence, the number of possible subsets of the sets of this category is $O(\sum_{s_i \leq B} 2^{s_i}) = O(\sum_{s_i \leq B} N/\log^2 N) = O(N^2/\log^2 N)$, because there are at most N sets.

If $s > B$, we claim that $w = O(N/\log N \cdot \log(2s/B))$. Hence, the number of subsets of the sets of this category is $O(\sum_{s_i > B} N/\log N \cdot \log(2s_i/B)) = O(N/\log N \cdot \sum_{s_i > B} s_i/B)$. This also implies a bound of $O(N^2/\log^2 N)$, because $\sum_{s_i > B} s_i \leq N$.

For the remainder of the proof we will show that $w = O(N/\log N \cdot \log(2s/B))$ if $s > B$. To maximize w , the sizes of the subsets of S should be as small as possible since they add up to at most N . Precisely, these will be all the possible sets of sizes $1, 2, \dots, h$, where h is the integer satisfying

$$(2.1) \quad \sum_{j=1}^{h-1} j \binom{s}{j} < N \leq \sum_{j=1}^h j \binom{s}{j}.$$

And w is bounded as

$$(2.2) \quad w \leq \sum_{j=1}^h \binom{s}{j}.$$

Consider the case when the value of h resulting from the solution of (2.1) is $h = \Theta(s)$. The number of sets whose sizes is $\Theta(s)$ is $O(N/s)$, which is $O(N/B)$ when $s > B$. In such case, these subsets will be dominating the value of w , as indicated by (2.2). As a consequence, $w = O(N/\log N)$, and the claim follows. We are only left with the case when $h = o(s)$, which is treated next.

Using (2.1) and the inequality $\binom{s}{j} \leq s^j/j!$, then

$$N \leq \sum_{j=1}^h j \binom{s}{j} \leq s \sum_{j=0}^{h-1} \frac{s^j}{j!}.$$

When $h = o(s)$, we obtain

$$N < \frac{2s^h}{(h-1)!}.$$

Using Stirling's formula, then

$$N < \frac{2s^h}{((h-1)/e)^h} \sqrt{h-1}.$$

Taking the base-2 logarithms, we get

$$(2.3) \quad \lg N < h \lg(s \cdot e/(h-1)) + \frac{1}{2} \lg(h-1) + 1.$$

Consequently, h is bounded from below as

$$(2.4) \quad h = \Omega\left(\frac{\log N}{\log(s/h)}\right).$$

From (2.2), when $h = o(s)$, the number of subsets of S whose size is h dominates w . Hence, we can alternatively bound w as

$$w = O(N/h).$$

Substituting for h from (2.4), then

$$w = O\left(\frac{N}{\log N} \log(s/h)\right).$$

To establish the claim, we show that $\log(s/h) = O(\log(2s/B))$ when $s > B$. We consider two cases:

- If $s = \Omega(B^2)$, then

$$\log(s/h) < \log s = \Theta(\log(s/B)).$$

- If $B < s = o(B^2)$, we show that $h > \lfloor B^2/6s \rfloor$, which implies $\log(s/h) = O(\log(2s/B))$.

The proof is by contradiction. Assume

$h \leq \lfloor B^2/6s \rfloor$, and substitute with $h = \lfloor B^2/6s \rfloor$ in the right-hand side of (2.3). Thus,

$$\begin{aligned} h \lg(s \cdot e/h) &\leq \frac{B^2}{6s} (2 \lg(s/B) + \lg(6e)) \\ &< \frac{B^2}{6s} \left(\frac{s}{B} \lg e + \lg(6e) \right) \\ &< \frac{B}{6} \lg(6e^2) = (1 - \epsilon)B, \end{aligned}$$

where $0 < \epsilon < 1$ is a fixed constant. The other terms of the right-hand side of (2.3) evaluate to $o(B)$. As a consequence, (2.3) would imply

$$\lg N < (1 - \epsilon)B + o(B).$$

But, this is not true when N is large enough.

Now, we give a matching lower bound for dense sets, by giving a construction for a dense family whose subset graph is of size $\Theta(N^2/\log^2 N)$.

LEMMA 2.3. *There exist families of dense sets whose subset graph is of size $\Theta(N^2/\log^2 N)$.*

Proof. Let e_1, e_2, \dots, e_n be the domain elements, and assume that n is divisible by 4. Our construction has two groups of sets. The first group has all the sets with the elements $e_1, e_2, \dots, e_{n/2}$ and $n/4$ among the elements $e_{n/2+1}, \dots, e_n$. The second group has all the sets with $n/4$ elements among the elements $e_1, e_2, \dots, e_{n/2}$. The number of sets in each group is

$$\binom{n/2}{n/4} = \Theta(2^{n/2}/\sqrt{n}).$$

It follows that $N = \Theta(2^{n/2} \cdot \sqrt{n})$, which ensures that $n = \Theta(\log N)$. Each set of the second group is a subset of all the sets of the first group, indicating that the size of this complete bipartite subset graph is $\Theta(2^n/n) = \Theta(N^2/\log^2 N)$.

Next, we show that the bound derived in Lemma 2.2 is superior to $O(N^2/\log^2 N)$ when considering the number of supersets and subsets for large sets.

LEMMA 2.4. *The sum of supersets and subsets for all sets of size $s = \omega(\log N)$ is $o(N^2/\log^2 N)$.*

Proof. Let $s_i = f_i(N) \cdot \log N$ be the size of the i -th such set, where $f_i(N)$ is a monotonically increasing function in N . Let $f(N)$ be the average of these functions. Then, the number of such sets $k' = O(N/(\log N \cdot f(N)))$. This implies that the total number of the supersets of such sets, which must be large sets as well, is

$$O(k'^2) = o(N^2/\log^2 N).$$

Using the $O(N/\log N \cdot \sum_i \log(s_i/\log N))$ bound from Lemma 2.2, and applying Markov's inequality, the number of subsets for such sets is

$$\begin{aligned} O(N/\log N \cdot \sum_i \log f_i(N)) \\ &= O(N/\log N \cdot k' \cdot \log f(N)) \\ &= O(N^2/\log^2 N \cdot \frac{\log f(N)}{f(N)}) \\ &= o(N^2/\log^2 N). \end{aligned}$$

Finally, we consider families with bounded intersections, where any two sets intersect in at most t elements. Obviously, for a set in this family to be a subset of another, its size should be at most t . The next lemma bounds the size of the subset graphs for such families and demonstrates a realization achieving such bound.

LEMMA 2.5. *Given a family of sets where any two sets intersect in at most t elements, for some constant t , the size of the subset graph is $O(N^{(2t-1)/t})$. Moreover, subset graphs with such bounds are realizable.*

Proof. We say that a set is *bounded* if its size is at most t , otherwise it is *unbounded*. Because a bounded set only has a constant number of subsets, we only consider the subset graph restricted to bounded sets being subsets of unbounded sets. Such restriction is obviously a bipartite graph, where one group has bounded sets and the other group has unbounded sets. We define the *effect* of a set to be the in-degree (out-degree) of the corresponding node within the subset graph divided by the set size. To maximize the size of the subset graph, all sets must have the same effect f ; otherwise we could have replaced a set with low effect with another with a higher effect and get a larger subset graph. Consider an unbounded set whose size is s . Since the in-degree of a node resembles the number of subsets of the corresponding set, thus $f \cdot s = O(s^t)$ implying $f = O(s^{t-1}) = O(n^{t-1})$, where $n > s$ is the number of domain elements. To maximize the size of the graph, the number of small sets should as well be maximized, pushing this number to $\Theta(n^t)$. Therefore, the size of the graph is $O(n^t \cdot n^{t-1}) = O(n^{2t-1})$. Since the sum of the in-degrees equals the sum of the out-degrees, thus $N = \Theta(n^t)$ and the lemma follows.

One realization of such graphs is a family having all the sets of size t and all the sets of size $n - t + 1$. To verify that this works, note that

$$\binom{n}{n-t+1} \cdot (n-t+1) = \Theta\left(\binom{n}{t}\right) = \Theta(n^t).$$

3 Computing the subset graph

We start by introducing two related problems used as subroutines for our algorithm, and show how to efficiently handle them. The intuition of the algorithm and a non-efficient solution are then stated. Later, we give the main algorithm and analyze its time and space requirements. Finally, the bounds are improved for a stronger model of computation.

Two related problems

Problem 1: Given a list of r entities (r is a power of 2) representing the integers from 0 to $r - 1$ in order, and a sequence x of $\log r$ bits. We need to find, in $O(r)$ bit operations, the entity corresponding to x .

Initialize a pointer to the first entity. As long as all the bits of x are not 0's, decrement x and advance the pointer to the next entity. When all the bits of x are 0's, the pointer is pointing to the desired entity.

To decrement x , traverse the bits of x from right to left and flip every 0 (if any) until reaching the first 1 and flip it. In other words, starting from the least-significant bit, a subsequence $\langle \dots 1, 0, \dots, 0 \rangle$ is converted to $\langle \dots 0, 1, \dots, 1 \rangle$.

It is easy to show that the total number of flips done

while decrementing x is $O(r)$. For completeness, we give a proof for this fact next.

LEMMA 3.1. *Given any sequence x of $\log r$ bits, the sum of bit flips done while decrementing x until all its bits are 0's is at most $2r - \log r - 2$.*

Proof. Consider the case when all the bits of x are initially 1's; this case involves the largest number of bit flips. For every decrement, exactly one 1 is flipped to a 0. Then, the total number of 1's flipped to 0's is $r - 1$. Since all the bits of x are finally 0's, the total number of 0's flipped to 1's is less than the total number of 1's flipped to 0's by $\log r$. It follows that the total number of 0's flipped to 1's is $r - \log r - 1$. Therefore, the total number of bit flips is at most $2r - \log r - 2$.

Problem 2: Given a list of r entities (r is a power of 2) representing the integers from 0 to $r - 1$ in order, and a sequence x of $\log r$ bits. We need to mark, in $O(r)$ bit operations, the entities corresponding to the binary numbers that are dominated by x . (m_1 dominates m_2 if and only if every 1-bit in m_2 has a 1-bit at the corresponding position in m_1 .)

We use another sequence c of $\log r$ bits to serve as a binary counter. Initialize all the bits of c to 0 and initialize a pointer to point to the first item in the list of entities. For the iterative step, we increment c and advance the pointer to the next entity.

To increment c , traverse the bits of c from right to left and flip every 1 (if any) until reaching the first 0 and flip it. In other words, starting from the least-significant bit, a subsequence $\langle \dots 0, 1, \dots, 1 \rangle$ is converted to $\langle \dots 1, 0, \dots, 0 \rangle$. Similar to Lemma 3.1, the total number of bit flips for incrementing c is $O(r)$.

To efficiently decide which entities correspond to an integer that is dominated by x , another pointer f is maintained to decide whether x dominates c or not. If x dominates c , f is set to *null*. Otherwise, f points to the most-significant position that has a 0-bit in x and a 1-bit in c . Dealing with f as an index to a bit position, we write $x_f = 0$ and $c_f = 1$. When c is incremented resulting in c' , the following are the possible cases:

- i. If f is not *null* (x is not dominating c) and $c_f = c'_f$ (the f -th bit of c is not flipped), then x is not dominating c' . Accordingly, f does not change.
- ii. Otherwise, let g be the most-significant flipped position of c , i.e. $c_g = 0$ and $c'_g = 1$.
 - (a) If $x_g = 1$, then x dominates c' . Accordingly, f is set to *null*.
 - (b) If $x_g = 0$, then x is not dominating c' . Accordingly, f is set to g .

In addition to the $O(r)$ checks and bit flips done while incrementing c in total, each case requires a constant number of bit checks per increment.

The solutions of the above two problems can be efficiently realized on a pointer machine, with both x and c implemented as linked lists of bits whose headers point to the least-significant bits.

Insight

We can construct the subset graph using the solutions of the above two problems as follows. The domain elements are arbitrarily identified with integers from 1 to n . Every set is viewed as an integer formed by a sequence of n bits; a 1-bit indicates that the element corresponding to this position is in this set, and a 0-bit indicates otherwise. It is easy to convert the sets to this format if they are not. We allocate a list of entities representing the integers from 0 to $2^n - 1$ in order. Each entity points to a string of set identifiers.

Using the solution to Problem 2, we handle the sets S_1, S_2, \dots, S_k in arbitrary order. For every set S_i , we append the identifier of S_i to an entity's string if the entity represents an integer that is dominated by the integer corresponding to S_i . This requires $O(2^n)$ operations per set. Later, to find the supersets of a set S_i , we search for the entity with the integer corresponding to S_i using the solution to Problem 1, and scan the string of set identifiers attached to this entity. This requires $O(2^n + k)$ operations per set.

As this solution is expensive, we show in the next subsection how to improve it for an efficient algorithm.

The algorithm

We partition the sequence of n domain elements into $\lceil n/p \rceil$ consecutive subsequences $U_1, U_2, \dots, U_{\lceil n/p \rceil}$ of size p each (the optimal value of p is determined in Lemma 3.2 as $\lceil \log k \rceil$). For each U_j , we associate a list of 2^p entities representing integers from 0 to $2^p - 1$. Each of these entities will be pointing to a string of k bits, one bit for each set. In the sequel, we call each list of entities associated with a subsequence a *lookup buffer* and call all the entities together with the strings they are pointing to the *auxiliary structure*.

Using the algorithm for the solution of Problem 2, we proceed building the auxiliary structure as follows. The sets are arbitrarily identified as S_1, S_2, \dots, S_k . For each set S_i , using the n -bit sequence representing S_i , we independently consider each U_j . For each such subsequence x of S_i , every string pointed to by an entity of the associated lookup buffer is appended by a bit at the i -th position. This bit is set to 1 if the integer corresponding to this entity is dominated by x , otherwise it is set to 0.

In the pseudo-code of Algorithm 1, $buffer.j$ represents the lookup buffer associated with U_j and $buffer.j.c$ represents the string of k bits corresponding to the entity of $buffer.j$ representing the integer c . At line 5, c is set to a sequence of p bits all of them are 0's. At line 6, the condition is checked using bitwise operations. At lines 7 and 12, the solution to Problem 2 is applied.

Algorithm 1 *build-structure*

```

1: for  $i = 1$  to  $k$  do
2:   let  $\langle x_1, x_2, \dots, x_n \rangle$  be the binary sequence representing the set  $S_i$ 
3:   for  $j = 0$  to  $\lceil n/p \rceil - 1$  do
4:      $x \leftarrow \langle x_{j*p+1}, x_{j*p+2}, \dots, x_{j*p+p} \rangle$ 
5:      $c \leftarrow 0$ 
6:     while  $(c < 2^p)$  do
7:       if  $(x$  dominates  $c)$  then
8:         append 1 to the string  $buffer.j.c$ 
9:       else
10:        append 0 to the string  $buffer.j.c$ 
11:      end if
12:      increment  $c$ 
13:    end while
14:  end for
15: end for

```

After building the auxiliary structure, the sets are re-traversed to build the subset graph. For each set S_i , the entity corresponding to each subsequence U_j within $buffer.j$ is identified, using the solution to Problem 1. This ends up with $\lceil n/p \rceil$ entities each pointing to a string of k bits. These $\lceil n/p \rceil$ strings are simultaneously scanned bit by bit. A 1-bit at position u , for all such strings, indicates that the set S_u corresponding to this position is a superset of S_i . Otherwise, if there is at least one 0-bit at position u of one of the strings, then S_u is not a superset of S_i . The subset graph is completely built once all the sets are considered.

In the pseudo-code of Algorithm 2, a list l temporarily holds pointers to the entities corresponding to S_i , where $l.j$ is a pointer to the corresponding entity in $buffer.j$. At line 5, to locate $buffer.j.x$, the solution to Problem 1 is applied.

It is straightforward to efficiently realize the algorithm on a pointer machine, by having all the sequences and strings implemented as linked lists, maintaining a pointer to the tail of every list to be able to efficiently append a bit, and implementing the solutions to Problem 1 and Problem 2 as stated earlier.

Analysis

LEMMA 3.2. *The running time and space required by the algorithm are $O(nk^2/\log k)$.*

Algorithm 2 *build-graph*

```
1: for all  $i = 1$  to  $k$  do
2:   let  $\langle x_1, x_2, \dots, x_n \rangle$  be the binary sequence representing the set  $S_i$ 
3:   for  $j = 0$  to  $\lceil n/p \rceil - 1$  do
4:      $x \leftarrow \langle x_{j*p+1}, x_{j*p+2}, \dots, x_{j*p+p} \rangle$ 
5:      $l.j \leftarrow \text{buffer}.j.x$ 
6:   end for
7:   for  $u = 1$  to  $k$  do
8:      $super \leftarrow 1$ 
9:     for  $j = 0$  to  $\lceil n/p \rceil - 1$  do
10:       $super \leftarrow super \wedge l.j.u$ 
11:    end for
12:    if ( $super == 1$ ) then
13:      declare  $S_u$  as a superset of  $S_i$ 
14:    end if
15:  end for
16: end for
```

Proof. The extra space used by the algorithm is basically that of the auxiliary structure. There are $\lceil n/p \rceil$ lookup buffers, each having 2^p entities, each pointing to a string of k bits. This sums up to $O(n/p \cdot k \cdot 2^p)$ bits. As for the running time, building the auxiliary structure requires $O(2^p)$ per set per subsequence (using the solution to Problem 2). This sums up to a total of $O(n/p \cdot k \cdot 2^p)$. Looking up the auxiliary structure to find the supersets of a set is done in two steps. Locating the correct entity requires $O(2^p)$ per set per subsequence (using the solution to Problem 1), while scanning the corresponding lists requires $O(n/p \cdot k)$ per set. This sums up to a total of $O(n/p \cdot k \cdot (2^p + k))$. To balance the work, we set $p = \lceil \log k \rceil$. The lemma follows.

LEMMA 3.3. *For a dense family of sets, the running time and space required by the algorithm are $O(N^2/\log^2 N)$.*

Proof. We show that for a dense family of sets $nk^2/\log k = O(N^2/\log^2 N)$.

Since $nk = \Theta(N)$ for a dense family of sets, and using the fact that $n \geq \log k$, then $k \log k = O(N)$ implying that $k = O(N/\log N)$. Fix a constant ϵ , $0 < \epsilon < 1$.

- i. If $k < N^{(1-\epsilon)/2}$, then $nk^2/\log k = O(nN^{1-\epsilon}) = O(N^{2-\epsilon}) = O(N^2/\log^2 N)$.
- ii. If $k \geq N^{(1-\epsilon)/2}$, then $\log k = \Theta(\log N)$. Plugging in the bounds: $nk = \Theta(N)$, $k = O(N/\log N)$ and $\log k = \Theta(\log N)$, then $nk^2/\log k = O(N^2/\log^2 N)$.

A stronger model of computation

Assuming words of size $b \leq k$ bits, allowing bits to be packed in words, and using bitwise operations, the running time and space requirements can be reduced. For such case, instead of allocating one bit per item in the strings pointed to by entities, we pack b bits representing b sets per word that is viewed as one item. The size of each list is henceforth $\lceil k/b \rceil$ words. To be able to check if there is a 1 in the same position for several strings, *bitwise-and* operations are performed on the corresponding items and the result is checked for the positions of 1's. For every set S_i , the output is then reported as $\lceil k/b \rceil$ words of b bits each; a 1-bit in the u -th position indicates that S_u is a superset of S_i . The following lemma would then replace Lemma 3.2.

LEMMA 3.4. *Using words of size $b \leq k$ bits and allowing bitwise operations, the running time required by the algorithm is $O(\frac{nk^2}{b \log(k/b+1)})$ and the space requirement is $O(\frac{nk^2}{b^2 \log(k/b+1)})$ words.*

Proof. As in the proof of Lemma 3.2, the storage requirement is $O(n/p \cdot k \cdot 2^p)$ bits, which is $O(n/p \cdot k/b \cdot 2^p)$ words. Building the auxiliary structure as well as locating the correct entities are similarly still $O(n/p \cdot k \cdot 2^p)$. However, scanning the lists and checking for positions that have 1's in the looked-up strings can now be done faster in $O(n/p \cdot k/b \cdot k)$, saving a factor of b . This sums up to a total running time of $O(n/p \cdot k \cdot (2^p + k/b))$. To balance the work, we set $p = \lceil \log(k/b + 1) \rceil$. The lemma follows.

The case when $b = \Theta(\log N)$ is of special interest.

COROLLARY 3.1. *For dense families of sets, using words of $\Theta(\log N)$ bits and allowing bitwise operations, the running time and space required by the algorithm are $O(N^2/\log^3 N)$ and $O(N^2/\log^4 N)$ respectively.*

Dynamic updates

Given a family of k sets and the corresponding subset graph, it may be required to modify the subset graph following a new set *insertion* or an existing set *deletion*. We show how to do that in $O(nk/\log k)$ time per update, which is $O(N/\log k) = o(N)$ for dense families.

As before, we maintain an auxiliary structure with $\lceil n/p \rceil$ lookup buffers. When a new set is inserted, a bit representing this set is appended to every string pointed to by an entity. This bit is set to 1 if the associated subsequence of the new set is a superset of the integer corresponding to the entity, otherwise it is set to 0. Similar to our construction algorithm, the supersets of the new set are found and reported. To

be able to identify the subsets of the new set, we need to augment the auxiliary structure with lists that mark the subsets, in addition to the supersets, of every integer corresponding to an entity. This introduces a problem that is similar to Problem 2, which could in a similar manner be solved in linear time with respect to the number of entities. Updating the auxiliary structure and reporting the subsets and supersets of the new set will then require $O(n/p \cdot (2^p + k))$ time, which is $O(nk/\log k)$ by setting $p = \lceil \log k \rceil$. The deletion task is even simpler. Only the bits corresponding to the set to be deleted are removed from the auxiliary structure, and the node as well as the incident arcs that represent this set in the subset graph are also removed. This also requires $O(nk/\log k)$ time when $p = \lceil \log k \rceil$.

However, we are not done yet. The problem is that the value of k is now changing, and the size of the auxiliary structure and accordingly our established bounds rely on the value of p which depends on k . A solution that achieves the bound in the amortized sense is to reconstruct the auxiliary structure whenever the number of sets k is doubled or is reduced by half. Then, we can use the construction algorithm to construct the new auxiliary structure in $O(nk^2/\log k)$ time. Since the number of updates that will be performed before a subsequent construction is at least $k/2$, the construction time is amortized as $O(nk/\log k)$ cost per update.

An efficient worst-case solution is as well possible. The idea is to maintain three auxiliary structures, where one of them is always under construction. For a given power of two k' , we maintain and use an auxiliary structure corresponding to $k = k'$, plus two more structures: one corresponding to $2k'$ and the other to $k'/2$. We will be always busy building one of these latter two auxiliary structures. Every new update is performed on the three structures (except for a deletion of a set that was not integrated to the incomplete structure). Accompanying a new update, two sets from the current structure, which are not yet in the incomplete structure, are integrated to it. We keep using the auxiliary structure corresponding to k' until the number of sets becomes either $2k'$ or $k'/2$. In such case, we start using the appropriate auxiliary structure, dismiss the other one, and start building a new structure (by initializing it and incrementally integrating sets to it with the upcoming updates). More specifically, if the number of sets becomes $2k'$, use the corresponding structure, dismiss the structure corresponding to $k'/2$, and start building a structure corresponding to $4k'$. If the number of sets becomes $k'/2$, use the corresponding structure, dismiss the structure corresponding to $2k'$, and start building a structure corresponding to $k'/4$. In either case, spending $O(nk/\log k)$ time per update

building the incomplete structure is enough to finish the job just on time.

Acknowledgment Thanks to Khaled Elbassioni for introducing the problem, for several rounds of reviews to the proof of Lemma 2.2, and for suggesting handling families with bounded intersections.

References

- [1] P. Pritchard, *A fast bit-parallel algorithm for computing the subset partial order*, *Algorithmica*, 24 (1999), pp. 76–86.
- [2] P. Pritchard, *On computing the subset graph of a collection of sets*, *Journal of Algorithms*, 33 (1999), pp. 187–203.
- [3] P. Pritchard, *An old sub-quadratic algorithm for finding extremal sets*, *Information Processing Letters*, 62 (1997), pp. 329–334.
- [4] P. Pritchard, *A simple sub-quadratic algorithm for computing the subset partial order*, *Information Processing Letters*, 56 (1995), pp. 337–341.
- [5] P. Pritchard, *Opportunistic algorithms for eliminating supersets*, *Acta Informatica*, 28 (1991), pp. 733–754.
- [6] H. Shen, *Fully dynamic algorithms for maintaining extremal sets in a family of sets*, *International Journal of Computer Mathematics*, 69 (1998), pp. 203–215.
- [7] H. Shen and D. Evans, *Fast sequential and parallel algorithms for finding extremal sets*, *International Journal of Computer Mathematics*, 61(3-4) (1996), pp. 195–211.
- [8] D. Yellin and C. Jutla. *Finding extremal sets in less than quadratic time*, *Information Processing Letters*, 48 (1993), pp. 29–34.
- [9] D. Yellin. *Algorithms for subset testing and finding maximal sets*, 3rd ACM-SIAM Symposium on Discrete Algorithms, (1992), pp. 386–392.