On Computing the Subset Graph of a Collection of Sets

Paul Pritchard*

Abstract

The extremal sets in a collection of sets are those that either have no subset or have no superset in the collection. Yellin and Jutla presented an algorithm for determining the extremal sets by constructing the partial order induced by the subset relation, which they called the "subset graph". This paper presents alternative implementations of their algorithm that have several advantages, including lower worst-case complexities. Also, an upper bound is established on the maximal number of edges in a subset graph of given size (measured by the sum of the cardinalities of the sets at the vertices). This matches (up to a constant factor) the lower bound constructed by Yellin and Jutla.

1 Introduction

Yellin and Jutla [7] tackled the following problem, for applications of which see [5].

> Given is a collection $\mathcal{F} = \{S_1, \ldots, S_k\}$, where each S_i is a set over the same domain D. A set is a *minimal* (resp. *maximal*) set of $\mathcal{F}$ iff it has no strict subset (resp. superset) in $\mathcal{F}$. Find the *extremal* sets of $\mathcal{F}$, i.e., those that are minimal or maximal.

With the problem size chosen as $N = \sum_i S_i$, Yellin and Jutla presented an abstract algorithm together with two implementations, one with worst-case running-time $O(N^2/\sqrt{\log N})$, and the other taking randomized expected time $O(N^2/\log N)$. Their algorithm constructs the partial order on $\mathcal{F}$ induced by the subset relation, the graph of which they call the *subset graph*.

We show that alternative implementations may be given that use only simple data-structures and that have lower worst-case complexities.

*On leave from School of Computing and Information Technology, Griffith University, Queensland, Australia 4111

Our first implementation achieves a worst-case running time of $O(N^2 \log\log N/\log N)$ in $O(N^2/\log^2 N)$ words of space. Our second implementation achieves a worst-case running time of $O(N^2/\log N)$ operations on $\Theta(\log N)$-bit words, matching that of our algorithm given in [6], but can require considerably more space ($O(N^2/\log N)$ words).

We also establish an upper bound of $O(N^2/\log^2 N)$ on the maximal number of edges in a subset graph of size N, i.e., on $|\{(x,y) : x, y \in \mathcal{F},\ y \subset x\}|$. This matches (up to a constant factor) the lower bound constructed in [7].

2 The Algorithm of Yellin and Jutla

We recall the key definitions from [7]. A *subset graph* G for $\mathcal{F}$ is a DAG such that each vertex is labeled by one or more identical sets in $\mathcal{F}$, each set in $\mathcal{F}$ labels exactly one vertex, and there is a path (of length at least 1) from a vertex labeled by S to a vertex labeled by S' iff $S' \subset S$. G is *closed* if an edge connects the endpoints of every path. The closed subset graph for $\mathcal{F}$ is unique: it represents the poset induced by the subset relation. For each d in the domain D, weight(d) denotes the number of sets in $\mathcal{F}$ that contain d; we assume weight(d) > 0. For $D' \subseteq D$ and $S \in \mathcal{F}$, we define $S|D' = S \cap D'$ and $\mathcal{F}|D' = \{S_1|D', \ldots, S_k|D'\}$.

Yellin and Jutla's algorithm maintains the closed subset graph for $\mathcal{F}|D'$. It starts with $D' = \emptyset$, and incrementally increases D', by adding one element at a time in order of decreasing (by which is meant non-increasing) weight, until $D' = D$. The vertices of the graph are labeled with the full sets of $\mathcal{F}$ rather than their projections onto D'.

The code for updating the graph prior to adding d to D' is as follows:

> $\{\ G = \mathcal{F}|D'\ \}$
> Mark each vertex labeled by a set containing d;
> **forall** marked vertices v **do**
>   Create a new vertex v';
>   Transfer each S labeling v s.t. $d \in S$ from v to v'
> **end forall**;
> **forall** marked vertices v **do**
>   **if** v is unlabeled **then**
>   Replace each edge $v \to z$ by edge $v' \to z$
>   **else**
>   Add edge $v' \to z$ for each edge $v \to z$;
>   Add edge $v' \to v$
>   **end if**;
>   Add edge $v' \to w'$ for each edge $v \to w$ with w marked
> **end forall**;
> Remove each unlabeled vertex and all edges to it;
> Unmark each marked vertex

$\{\ G = \mathcal{F}|(D' \cup \{d\})\ \}$

(We have given slightly higher-level code than in [7], and changed the order of statements slightly to better reflect the intended implementation.)

The abstract implementation in [7] uses a dictionary at each vertex to represent the endpoints of the edges from that vertex. It is observed that the statement

$$\text{Replace each edge } v \to z \text{ by edge } v' \to z$$

can be implemented in $O(1)$ time by simply transferring the dictionary at v to v'.

In the introductory section of [7], it is claimed that

> Using hashing, the graph can be constructed in expected time proportional to the size of the graph. Using the trie data structure of Willard ... adds a multiplicative factor of $\sqrt{\log N}$.

Here size means number of edges. But this is false. If

$$\mathcal{F} = \{1\},\ \{2, \ldots, m\},\ \{1, \ldots, m\},$$

then the complete subset graph has only 2 edges, but if the elements are processed in numeric order (which is by decreasing weights), then m edges will be deleted.

However, the complexity analysis in [7] does show that the complete subset graph is constructed in $O(N^2/\log N)$ dictionary operations, each of which costs $O(1)$ randomized amortized expected time using dynamic perfect hashing or $O(\sqrt{\log k})$ worst-case time using q-tries. (Recall that k is the number of sets in $\mathcal{F}$.) Both implementations use space proportional to the number of edges in the (current) subset graph.

3 An Alternative Implementation Using Ordered Arrays

There is a simpler alternative to the deterministic implementation using q-tries, which retains the attractive features of giving a worst-case complexity (and therefore a guaranteed time-bound), requiring $O(1)$ space per edge in the current subset graph, and using a RAM model of modest power (none of which apply to dynamic perfect hashing).

In essence, the idea is simply to store, for each vertex, the endpoints of the edges from that vertex in contiguous elements of an array, and in increasing order. We proceed to outline the main features of the implementation. Minor housekeeping requirements are omitted.

Let vertices be numbered by consecutive natural numbers in order of their creation. A set of $\mathcal{F}$ is represented by an array or linked list of its elements arranged in order of processing by the algorithm (by decreasing weight). Initially, all sets are attached to vertex 0 (in a linked list). Vertices will be marked by following an element list linking all sets of $\mathcal{F}$ that contain d. On initialization, the *first element only* of each set determines the element list to which the set is linked. When a set containing d is transferred from v to v', if d has successor d', then the set is appended to the end of the element list for d'. This strategy ensures that when the element list for d is used, it links sets by increasing order of the vertices that they label. The lists of sets labeling vertices will need to be doubly-linked to permit constant-time deletion.

To implement the statement

$$\text{Replace each edge } v \to z \text{ by edge } v' \to z$$

vertex v' is made to point to the array-section of vertices pointed to by v, requiring $O(1)$ time. To implement the statement

$$\text{Add edge } v' \to z \text{ for each edge } v \to z$$

the array-section of vertices pointed to by v is copied to become the initial array-section for v'. This requires $O(1)$ time for each edge added. Since (as may readily be checked) a vertex z at the endpoint of an edge from a vertex v satisfies $z < v$, the next statement

$$\text{Add edge } v' \to v$$

preserves the order for vertex v'.

Now consider the statement

$$\text{Add edge } v' \to w' \text{ for each edge } v \to w \text{ with } w \text{ marked.} \qquad (1)$$

To implement it, the marked vertices are gathered in an array in increasing order during execution of the first statement in the algorithm. Then all vertices w that are in both this array and the array-section pointed to by v are found in increasing order, and the corresponding numbers w' appended to the array-section for v'.

Let the number of marked vertices and the number of vertices at the endpoint of an edge from vertex v be n, m (not necessarily respectively), with $m \leq n$. Then the "opportunistic" algorithm exploited in [5, 6] will find the common vertices in order using $O(m \log(1 + n/m)) = O(n)$ comparisons of elements. This is optimal up a constant factor for this problem. Absolutely optimal algorithms are discussed in detail in [4].

The last statement in need of implementation is

$$\text{Remove each unlabeled vertex and all edges to it.}$$

Although it is possible to arrange to find each occurrence of an unlabeled (and therefore defunct) vertex in $O(1)$ time, removing these occurrences would be too time-consuming, because the array-sections that contain them would have to be compacted. So these occurrences of defunct vertices are temporarily retained.

When an array-section is copied, all occurrences of defunct vertices are removed, and the array-section is compacted. Note that for the previous analyses to apply, the defunct vertices at endpoints of edges from v must be counted.

The analysis in [7] shows that the total time required by all statements other than (1) is $O(N^2/\log N)$. In the first $\log_2(N/\log^2 N)$ iterations of the algorithm, the number of vertices is $O(N/\log^2 N)$, since it starts at 1 and at most doubles in each iteration. Therefore the parameters n, m determining the cost of each execution of (1) are $O(N/\log^2 N)$, and therefore so is the cost. In the remaining iterations, each element d has weight $O(N/\log N)$, since $\Theta(\log N)$ elements have already been processed, and elements are taken in decreasing order by weight. Now the number of marked vertices is $O(\text{weight}(d))$, and the number of vertices in the array-section is $O(k) = O(N)$, defunct vertices included. So the cost of (1) is

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

Since over the entire algorithm, the total number of executions of (1) is $O(N)$, their total cost is $O(N^2\log\log N/\log N)$, and this is the worst-case complexity of our implementation.

The space-complexity is dominated by the array-sections of endpoints of edges from vertices. They can all be stored in a single array. Rather than allocating the maximum possible space — $O(k)$ — for each vertex, a modest initial allocation can be made, and whenever a section is exceeded, the whole section can be copied to a new section of twice the size. The time-complexity is increased by a factor of at most 2, and at most double the minimum amount of space needed is used. The vacated section can be noted and used for vertices created later. Discounting the defunct vertices that are retained, the space used is of the same order as that needed for the implementation using q-tries. To ensure that the defunct vertices do not increase the order of space required, we can keep counts of the numbers of occurrences in array-sections of defunct and active vertices (overall, or for each vertex), and compact when the former reaches the latter. By charging the cost to the creation of the edge that is deleted, we see that only the constant factor of the time-complexity can be altered. The space-complexity is at most doubled.

The ordered arrays implementation has several advantages over its deterministic rival using q-tries, one being that it uses much less complicated algorithmic machinery. Both run in the same order of space, but the former has a lower worst-case running time. On the family of sets from [7], which was designed to have as large a subset graph as possible, and therefore is presumably a difficult input for the algorithm, the array-based implementation can be seen to run in time $O(N^2/\log N)$, since $k = O(N/\log N)$.

However, the new implementation cannot be said to be uniformly “at least as good”, because when there are sufficiently few marked vertices, the cost of executing statement (1) is higher. The new implementation can take as much as $\Theta(\log N)$ time per new edge added, whereas the q-tries implementation always takes $O(\sqrt{\log N})$ time.

4 An Alternative Implementation Using an Adjacency Matrix

An adjacency-matrix representation of the subset graph results from implementing each dictionary of endpoints of edges from a vertex as a bit-vector. A surprisingly attractive implementation of Yellin and Jutla's algorithm can be built on this foundation.

Notionally, the adjacency matrix will have k rows, each comprising $\sim k/\log_2 N$ words of $\sim \log_2 N$ bits. The sets of $\mathcal{F}$ that label vertices, and the element lists that link sets having the same first member not less than d (and the vertices that they label), are managed as before.

As usual, the statement

$$\text{Replace each edge } v \to z \text{ by edge } v' \to z$$

is implemented by making v' point to the row for v, in $O(1)$ time. Statement

$$\text{Add edge } v' \to z \text{ for each edge } v \to z$$

is implemented by copying the row for v to the new row for v'. In order for it to take no more than $O(1)$ time per new edge, the well-known technique of avoiding initialization is used — see [1]. We adapt it slightly to maintain validating back-pointers to the words containing 1-bits rather than those initialized. All that is necessary is to remove the corresponding back-pointer when an entire word becomes zero when the sole edge that it records is removed, replacing it by the highest-numbered back-pointer, and decrementing the count of non-zero words.

Implementing statement

$$\text{Add edge } v' \to v$$

amounts to setting the appropriate bit, after initializing the corresponding word (and recording this fact) if necessary, all of which takes $O(1)$ time.

Now consider the statement

$$\text{Add edge } v' \to w' \text{ for each edge } v \to w \text{ with } w \text{ marked.}$$

The bit in the bit-vector for v that corresponds to each marked vertex w is tested, and if it is a 1-bit, the corresponding bit w' in the bit-vector for v' is made a 1-bit.

The last statement in need of implementation is

$$\text{Remove each unlabeled vertex and all edges to it.}$$

In order to limit the rows to k rather than N bits (solely to save space), when a vertex v becomes unlabeled, its column in the adjacency matrix is taken over to represent the starting vertices of the edges to v'. Because it is too expensive to zero the column, the non-zero bits in it are individually zeroed. To find each one in $O(1)$ time, a list of them is maintained, and updated whenever an edge to v is created. Note that when copying (the active part of) a bit-vector to add a new row to the matrix, well-known bit-manipulation techniques enable each 1-bit to be extracted in $O(1)$ time if a 2's complement integer representation is used. Alternatively, lookup in a table indexed by each possible pattern of bits in a word can be used. Also note that care must be taken when a column is zeroed to retain the starting points of new edges created by the statement previously examined.

Each statement has been implemented in worst-case time no more than the randomized amortized expected time required by the implementation using dynamic perfect hashing, so the worst-case complexity of this new implementation is $O(N^2/\log N)$. The space required is $\Theta(k^2/\log N)$ words for the adjacency matrix, plus $O(1)$ words for each edge of the current subset graph. In contrast, the dynamic perfect hashing implementation only needs space proportional to the number of edges in the (current) subset graph. This is shown to be $O(N^2/\log^2 N)$ in the next section.

Given the much greater complications of the dynamic perfect hashing implementation, the adjacency matrix implementation is expected to be much faster in practice. (The performance of the first optimized implementation of dynamic perfect hashing, as briefly discussed in [2], supports this view.) Furthermore, the extra space for the adjacency matrix may be tolerable in many practical circumstances (such as when k is small).

5 An Upper Bound on the Size of a Subset Graph

Note: where not specified, all logarithms in this section are natural logarithms.

In [7] a family of sets of size N is constructed whose unique subset graph has $\Theta(N^2/\log^2 N)$ edges, thereby establishing a lower bound on the maximum size of a subset graph for a collection of sets of size N.

An upper bound of $O(N^2/\log N)$ is implicit in the analysis of Yellin and Jutla's algorithm. We proceed to establish a tighter upper bound by adapting and tightening the complexity analysis in [6].

Let $\mathcal{F}_n$ denote $\{S : S \in \mathcal{F},\ |S| = n\}$, $\mathcal{F}_m^n$ denote $\cup_{m\le i\le n}\mathcal{F}_i$, and $\mathcal{F}\#\mathcal{G}$ denote $|\{(x,y) : x \in \mathcal{F},\ y \in \mathcal{G},\ y \subset x\}|$. Partition $\mathcal{F}$ into the collection $\mathcal{F}_0^D$ of "smaller" sets and $\mathcal{F}_{D+1}^K$ of "larger" sets, where the cutoff D is to be determined later.

The total number of subsets of smaller sets is

$$\mathcal{F}_0^D\#\mathcal{F} = \mathcal{F}_0^D\#\mathcal{F}_0^D = O(2^D N), \tag{2}$$

since there are $O(N)$ smaller sets, and each has $O(2^D)$ subsets.

The larger sets may have subsets in both partitions. The total number of subsets *from* the larger sets is

$$\mathcal{F}_{D+1}^K\#\mathcal{F}_{D+1}^K = O(N^2/D^2), \tag{3}$$

since there are $O(N/D)$ larger sets.

It remains to bound the total number of subsets of the larger sets from the smaller sets. We first partition the larger sets at cardinality D^2. Then

$$\mathcal{F}^K_{D^2+1} \# \mathcal{F}^D_0 = O(N^2/D^2), \tag{4}$$

since there are $O(N/D^2)$ largest sets and each has $O(N)$ subsets.

To bound the total number of subsets of the remaining larger sets from the smaller sets, we partition the latter at d, $d < D$, to be determined later. We have

$$\mathcal{F}^{D^2}_{D+1} \# \mathcal{F}^D_{d+1} = O\left(\frac{N^2}{Dd}\right), \tag{5}$$

since the numbers of sets in each collection in the l.h.s. of (5) are $O(N/D)$ and $O(N/d)$ respectively.

Finally, we have

$$\begin{aligned}
\mathcal{F}^{D^2}_{D+1} \# \mathcal{F}^d_0 &\leq \sum_{n=D+1}^{D^2} |\mathcal{F}_n| \sum_{m=0}^{d} \binom{n}{m} \\
&\leq \sum_{n=D+1}^{D^2} |\mathcal{F}_n| \sum_{m=0}^{d} \binom{D^2}{m} \\
&< \frac{N}{D} \sum_{m=0}^{d} \frac{(D^2)^m}{m!} \\
&= O(ND^{2d-1}).
\end{aligned} \tag{6}$$

Now put

$$D = \log_2(N/\log^2 N),$$

so that $D = \Theta(\log N)$ and $D < \log_2 N$. Terms (2), (3), (4) become $O(N^2/\log^2 N)$. For term (6), since $N = \log_2 N^{\log N/\log\log_2 N}$, we choose

$$d = \frac{1}{2}\left(\frac{\log N}{\log\log_2 N} - 1\right),$$

so that ND^{2d-1} is also $O(N^2/\log^2 N)$. Lastly, the remaining term (5) is now seen to be $O(N^2 \log\log N/\log^2 N)$, which therefore bounds the number of edges in a subset graph.

To bring this closer to the known lower bound of $\Omega(N^2/\log^2 N)$, we need only consider term (5).

Now that d is known, we partition $\mathcal{F}^{D^2}_{D+1}$ at $D\log D$, since

$$\mathcal{F}^{D^2}_{D\log D+1} \# \mathcal{F}^D_d = O\left(\frac{N^2}{dD\log D}\right) = O\left(\frac{N^2}{\log^2 N}\right).$$

It follows that the number of edges in the subset graph is

$$\mathcal{F}\#\mathcal{F} = O(N^2/\log^2 N) + \mathcal{F}_{D+1}^{D\log D}\#\mathcal{F}_{d+1}^{D}.$$

Consider the latter term. Let $x \in \mathcal{F}_{D+1}^{D\log D}$ and $|x| = rD$ where $1 < r \leq \log D$. The maximum number of subsets of x in $\mathcal{F}_{d+1}^{D}$ is obtained by taking all subsets of x of cardinality d, then all of cardinality $d+1$, etc. Let h be the maximum cardinality used to give the biggest possible collection. Then $h \leq D$, and the collection has size (not cardinality) bounded by N. So

$$\begin{aligned}
N &\leq \sum_{m=d+1}^{h} \binom{rD}{m} m \\
&= \sum_{m=d+1}^{h} rD \binom{rD-1}{m-1} \\
&< rD \sum_{m=d}^{h-1} \frac{(rD-1)^m}{m!} \\
&< \frac{(rD)^h}{(h-1)!} \sum_{i=0}^{\infty} \left(\frac{h-1}{rD}\right)^i \\
&= \frac{(rD)^h}{(h-1)!} \left(1 - \frac{h-1}{rD}\right)^{-1} \\
&\leq 2(rD)^h/(h-1)! \quad \text{if } r \geq 2,
\end{aligned}$$

where we have used $\binom{n}{m}m = n\binom{n-1}{m-1}$. Taking logs, we have

$$\log N \leq h\log(rD) - h\log h + h + O(\log h)$$

using Stirling's approximation for $h!$ (see [3]). Since $D = \log_2(N/\log^2 N)$, for N sufficiently large,

$$\frac{D}{\log_2 e} \leq h\left(\log\frac{rD}{h} + 2\right). \tag{7}$$

For given r, $2 \leq r \leq \log D$, let

$$H_r(h) = h\left(\log\frac{rD}{h} + 2\right) - \frac{D}{\log_2 e}. \tag{8}$$

As may be readily verified, $H_r(d) < 0$ for N sufficiently large, $H_r(D) > 0$, and the derivative of $H_r(h)$ is $\log(rD/h) + 1$, which is positive in the interval $(d, D]$. Therefore this interval contains a unique root, say $h^*(r)$. We find that

$$h^* = \begin{cases} \Theta(D), \text{ if } r \text{ is constant}, \\ \Theta(D/\log\log\log D), \text{ if } r = \log\log D, \\ \Theta(D/(\delta\log\log D)), \text{ if } r = \log^\delta D,\ 0 < \delta \leq 1. \end{cases} \tag{9}$$

In particular, we find that

$$h^*(2) = \frac{D}{c\log_2 e} \approx \frac{D}{3.25} ,$$

where $c \approx 2.25$ is the solution to

$$c - \log c - (2 + \log 2 + \log\log_2 e) = 0.$$

Since $\binom{n}{m} = \frac{n-m+1}{m}\binom{n}{m-1}$, the average cardinality of a set, in the biggest possible collection of subsets of x from $\mathcal{F}^D_{d+1}$, will be $\Omega(h^*(r))$, where $|x| = rD$. I.e., the cardinality of this maximal collection is $O(N/h^*)$. In each of the three cases of (9), we find that

$$\frac{N}{|x|}\cdot\frac{N}{h^*} = O\left(\frac{N^2}{\log^2 N}\right),$$

with the l.h.s. monotonically decreasing with $|x|$.

So let $h^*(2) = h_2$ and $r > 2$. Then

$$\begin{aligned}
H_r\left(\frac{2h_2}{r}\right) &= \frac{2h_2}{r}\left(\log\frac{r^2 D}{2h_2} + 2\right) - \frac{D}{\log_2 e} \\
&= \frac{2h_2}{r}\left(\log\frac{2D}{h_2} + 2\right) + \frac{2h_2}{r}\log\frac{r^2}{4} - \frac{D}{\log_2 e} \\
&= \frac{2}{r}\left(\frac{D}{\log_2 e}\right) + \frac{2}{r}\left(\frac{D}{c\log_2 e}\right)\log\frac{r^2}{4} - \frac{D}{\log_2 e} \\
&= \frac{D}{\log_2 e}\left(\frac{2}{r}\left(1 + \frac{2\log(r/2)}{c}\right) - 1\right).
\end{aligned}$$

Now

$$\frac{2}{r}\left(1 + \frac{2\log(r/2)}{c}\right) < \frac{2}{r}\left(1 + \log\frac{r}{2}\right) < \frac{2}{r}e^{\log(r/2)} = 1,$$

so $H_r(2h_2/r) < 0$. Therefore $h^*(r) > (2/r)h^*(2)$ for $r > 2$. Of course, when $D < |x| \leq 2D$, the biggest possible collection of subsets of set x from $\mathcal{F}^D_{d+1}$ has cardinality $O(N/h_2) = O(N/\log N)$.

Accordingly,

$$\begin{aligned}
\mathcal{F}^{D\log D}_{D+1} \# \mathcal{F}^D_{d+1} &= O\left(\frac{N}{D}\cdot\frac{N}{h_2}\right) \\
&= O(N^2/\log^2 N),
\end{aligned}$$

establishing the following result.

Theorem *The number of edges in the subset graph of a collection of sets of size N is $O(N^2/\log^2 N)$.*

This matches the lower bound constructed in [7] up to a constant factor.

6 Final Remarks

In [6] we analyzed some algorithms that we previously proposed for computing the minimal sets in a family of sets. After noting that they could easily be adapted to compute all extremal sets, and, indeed, the subset graph itself, we showed that one of them, based on an opportunistic subset test, has worst-case complexity $O(N^2/\log N)$, and that this bound is tight. We had previously showed [5] that the time-complexity is $O(N \log N)$ if each set in $\mathcal{F}$ has cardinality bounded by a constant. The extra space required by the algorithm is only $O(N)$ words to compute the extremal sets, and, following the upper bound established in this paper, $O(N^2/\log^2 N)$ space to store the subset graph. We wrote [6]:

> Nevertheless, the approach in [7] is conceptually cleaner than ours, being founded on a natural mathematical structure, and perhaps offers better prospects for improvement.

We have begun to investigate those prospects in this paper, and have found two relatively simple implementations with good worst-case performance. The analysis leading to the upper bound on the size of a subset graph may be a useful starting point for a more careful analysis of Yellin and Jutla's algorithm, which may in turn point the way to better implementations or useful variations of the algorithm itself.

Acknowledgments

I wish to thank the Department of Computer Science at the University of Edinburgh for kindly hosting my sabbatical visit.

References

[1] A. V. Aho, J. E. Hopcroft, and J. D. Ullman, *The Design and Analysis of Computer Algorithms*. Addison-Wesley, Reading, Mass. (1974).

[2] M. Dietzfelbinger, A. Karlin, K. Mehlhorn, F. Meyer auf der Heide, H. Rohnert, and R. E. Tarjan, Dynamic perfect hashing: upper and lower bounds. *SIAM J. Comput.* ***23*** (1994) 738–761.

[3] R. L. Graham, D. E. Knuth and O. Patashnik, *Concrete Mathematics*. Addison-Wesley, Reading, Mass. (1989).

[4] R. M. Karp, A generalization of binary search. In F. Dehne, J.-R. Sack, N. Santoro and S. Whitesides (eds), "Algorithms and Data Structures", *Lect. Notes in Comp. Sci.* ***709*** (1993) 27–34.

[5] P. Pritchard, Opportunistic algorithms for eliminating supersets. *Acta Informatica* ***28*** (1991) 733–754.

[6] P. Pritchard, An old sub-quadratic algorithm for finding extremal sets. Technical Report ECS-LFCS-94-301, Dept of Computer Science, University of Edinburgh, September 1994.

[7] D. M. Yellin and C. S. Jutla, Finding extremal sets in less than quadratic time. *Inform. Process. Lett.* ***48*** (1993) 29–34.