OPPORTUNISTIC ALGORITHMS FOR COVERING WITH SUBSETS

Paul Pritchard*

Department of Computer Sciences
The University of Texas at Austin
Austin, Texas 78712-1188

TR-89-12 May 1989

* On leave from the Department of Computer Science, University of Queensland, Australia.

Opportunistic Algorithms for Covering with Subsets

Paul Pritchard*
Department of Computer Sciences
The University of Texas at Austin

Abstract

The main problem tackled in this paper is that of finding each set in a given collection that has no proper subset in the collection. Starting with a solution that uses a quadratic (in the size of the collection) number of subset tests, solutions are developed which are opportunistic in the sense of running significantly faster for certain classes of input (such as when most sets are small). They are based on an opportunistic algorithm for the fundamental problem of finding an element common to two ordered sequences. Methodological issues are emphasized throughout.

1 The Problem

(Readers who are unfamiliar with propositional logic may go directly to Problem 1.) Suppose we are given a formula in a restricted conjunctive normal form: it is the conjunction of one or more subformulae, where each subformula is the disjunction of one or more propositional variables. (There are no other connectives or any quantifiers.) Then the formula may be maximally simplified (whilst retaining the normal form) by perfoming the following operations.

1. Remove each repeated occurrence of a propositional variable in each conjunct (using the associativity, commutativity, and idempotence of $\vee$).
2. Remove each conjunct that has a repeated set of propositional variables (using the associativity and commutativity of $\vee$, and the idempotence of $\wedge$).
3. Remove each conjuct that has a strict superset of the propositional variables in another (using the associativity and commutativity of $\vee$, and $P \wedge (P \vee Q) \equiv P$).

The first two operations may be dispensed with by modeling the formula as a set S of sets. A set x belongs to S if and only if it is the set of all propositional variables in some conjunct of the formula. However, in order to obtain the representation of the simplest formula of the same normal form that is equivalent to the given one, we must solve Problem 1, which models the third operation above, and which is the subject of the rest of the paper. (We encountered the propositional form of the problem in the context of constraint-satisfaction—see [5]. A propositional variable here corresponds to the assertion that a particular variable does not have a particular value.)

Problem 1 *Given a set S of subsets of a finite set U, find* $\mathrm{cover}(S)$ *such that* $\mathrm{cover}(S) \subseteq S$, $(\forall x : x \in S : (\exists y : y \in \mathrm{cover}(S) : y \subseteq x))$, *and* $\mathrm{cover}(S)$ *has minimal cardinality.*

(Note: $x \subseteq y$ means every member of x is a member of y; $x \subset y$ means $x \subseteq y \wedge x \neq y$.) We think of each set in S as being *covered* by a subset in $\mathrm{cover}(S)$.

The *mathematical* version of Problem 1 is easily solved—$\mathrm{cover}(S)$ is uniquely given by

*On leave from the Department of Computer Science, University of Queensland, Australia

$$\text{cover}(S) = (\textbf{Setof } x : x \in S \wedge (\forall y : y \in S : y \not\subset x)). \tag{1}$$

(We employ a consistent notation, due to E. W. Dijkstra, wherever bound variables are involved. The general form for a set-constructor is

$$(\textbf{Setof } x : D.x : t.x)$$

which denotes the range of the term $t.x$ when the domain of x is characterized by $D.x$. However, $t.x$ is commonly x, and in such cases we omit the second colon and term, giving a form very close to traditional mathematical notation; this abbreviation is also used with **Max** and **Min**.)

The *algorithmic* version of this problem—computing cover(S)—is not so easily disposed of. In order to simplify matters, we can index the universe and work with sets of indices, for which we introduce the following notation.

$$[a,b] = (\textbf{Setof } i : a \leq i \wedge i \leq b),\ [a,b) = [a,b] - \{b\},\ (a,b] = [a,b] - \{a\}.$$

We henceforth assume that $U = [0, M)$, $(0 < M)$. The cardinality of a set s is denoted $|s|$; hence $M = |U|$. We employ the following convention: a, b, c, d are integers (in U, unless stated otherwise); w, x, y, z are sets of integers (subsets of U, unless stated otherwise); n, m are cardinalities of subsets of U; S, X, Y are sets of subsets of U; M, N are cardinalities of sets of subsets of U, or lengths of sequences. The conventions also apply to primed and subscripted variables.

Problem 1 as stated does not exclude the cases $S = \emptyset$ (for which cover$(S) = \emptyset$) and $\emptyset \in S$ (for which cover$(S) = \{\emptyset\}$). Although these cases do not occur when modelling formulae (they correspond to the empty formula and the conjunct *false* respectively), they are legitimate cases in Problem 1, and should be handled correctly by our algorithms.

2 A High-Level Solution

Our prototypical high-level algorithm exhibits the computational content of equation (1). Our algorithms are expressed in the language of guarded commands used in [4], extended with **if** and **forall** commands. The command

if b **then** S **fi**

is an abbreviation of

if $b \rightarrow S \,[]\, \neg b \rightarrow skip$ **fi**

The **forall** command was introduced in [7] to denote iteration over a fixed finite set in unspecified order. We briefly recapitulate. Term **taken** denotes the set of values of the loop-variable for which the loop's command has been executed. Initially, **taken** $= \emptyset$; on termination, **taken** is the specified set. Our convention for specifying the result of a function is by pseudo-assignment to the function-name, as in Pascal.

Algorithm 1

$C := filter(S, S)$
$\{C = \text{cover}(S)\}$
where
function $filter(X, Y)$ **is**
$\{$**post** $filter = (\textbf{Setof } x : x \in X \wedge (\forall y : y \in Y : y \not\subset x))\}$
$F := X;$
$\{$**invariant** $F = (\textbf{Setof } x : x \in X \wedge (\forall y : (x,y) \in \textbf{taken} : y \not\subset x))\}$
forall (x, y) **in** $X \times Y$ **do**
if $y \subset x$ **then** $F := F - \{x\}$ **fi**
od;
$filter := F$

We say that $filter(X, Y)$ is the result of filtering X by Y.

Algorithm 1 minimally constrains the order of evaluation of the expressions $y \subset x$, which is why we prefer it to the alternative algorithm which determines for each x in turn the truth-value of $(\forall y : y \in Y : y \not\subset x)$.

3 Filtering With Smaller Sets

It is apparent from Algorithm 1 that there is a solution which uses a number of subset tests which is quadratic in N, $N = |S|$.

The most fundamental way to speed-up Algorithm 1 is to have it avoid considering certain pairs (x, y). A simple way to do so is to exploit the obvious fact that

$$y \subset x \Rightarrow |y| < |x|.$$

To do so, S is first partitioned according to the cardinality of its members:

$$K = (\textbf{Max } s : s \in S : |s|) \tag{2}$$

$$(\forall n : n \in [0, K] : L_n = (\textbf{Setof } s : s \in S \wedge |s| = n)) \tag{3}$$

A non-member of cover(S) in L_n must have a subset in L_m, for some m less than n. I.e., L_n need only be filtered by L_m if $m \in [0, n)$.

To avoid multiple consideration of members of L_m which are eventually deleted, i.e., not in cover(S), L_m should be completely filtered before filtering *by* it. Furthermore, it is preferable to filter a given L_n by taking the sets L_m in increasing order of m, on the presumption that the cost of evaluating $y \subset x$ is a non-decreasing function of $|y|$.

There are two natural algorithms consistent with the partial order on filterings thus defined. The first, given below as Algorithm 2, takes each L_m in increasing order of m, $m \in [0, K)$, and does all filtering *by* it. In the interest of Dirty Realism, S, which is represented implicitly as the union of the L_n's, is modified, progressively shedding members until only cover(S) remains. I.e., since the result of function *filter* is always assigned to its first argument, *filter* is reformulated as a procedure.

S_0 is a constant needed to formulate the postcondition in terms of the original value of S. The notation $P\langle x := t\rangle$ means P with all free occurrences of x replaced by t, with renaming of bound variables if necessary to avoid capture of free variables in t. The $\langle\rangle$ operator has highest precedence.

Algorithm 2

$\{S = S_0\}$
"Partition S by cardinality";
$\{((2) \wedge (3))\langle S := S_0\rangle\}$
$m := 0$;
{**invariant** $(\forall n : n \in [0, K] : L_n = (\textbf{Setof } s : s \in S_0 \wedge |s| = n \wedge$
$(\forall y : y \in (\textbf{Union } m' : m' \in [0, m) : L_{m'}) : y \not\subset s)))\}$
do $m < K \rightarrow$
{**invariant** $(\forall n : n \in \textbf{taken} : L_n = (\textbf{Setof } s : s \in S_0 \wedge |s| = n \wedge$
$(\forall y : y \in (\textbf{Union } m' : m' \in [0, m] : L_{m'}) : y \not\subset s)))\}$
forall n **in** $(m, K]$ **do** $filter(L_n, L_m)$ **od**;
$m := m + 1$
od
$\{(\textbf{Union } n : n \in [0, K] : L_n) = \text{cover}(S_0)\}$
where
procedure $filter(X, Y)$ **is**
$\{\textbf{pre } X = X_0\}$
$\{\textbf{post } X = (\textbf{Setof } x : x \in X_0 \wedge (\forall y : y \in Y : y \not\subset x))\}$

Note that in presenting an invariant for a loop—which obligation is always met—we sometimes omit conjuncts which are implied by the invariant of an enclosing loop or a preceding assertion by virtue of the fact that the variables concerned are unchanged. When buttressed with such missing conjuncts, the invariant supplied should be strong enough to support the postcondition.

The second algorithm takes each L_n in increasing order of n, $n \in (0, K]$, and does all filtering *of* it. Because this algorithm imposes a total order on the filterings (to be consistent with our partial-order), whereas Algorithm 2 doesn't, we proceed in our development with the latter, because the extra freedom it affords may prove useful.

4 Algorithms for Filtering

It remains to implement procedure $filter(X, Y)$. Just as before, there are two sweetly reasonable approaches.

In the first approach—given as Algorithm 3 below—for each member x of X, a search is performed for a member y in Y such that $y \subset x$, and x is removed from X if the search is successful.

Function "oneof" returns an arbitrary member of its argument, which must be a non-empty finite set. In conjunction with Y', which accumulates the values over which y ranges, it is used above to implement a search over Y in unspecified order.

Algorithm 3 (implementation of $filter(X, Y)$)

$\{X = X_0\}$
if $Y \neq \emptyset$ **then**
  $X' := \emptyset$;
  {**invariant** $Y \neq \emptyset \wedge X' \subseteq X_0 \wedge X = (\textbf{Setof}\ x : x \in X' \wedge (\forall y : y \in Y : y \not\subset x)) \cup (X_0 - X')$}
  do $X \not\subseteq X' \rightarrow$
    $x, y :=$ oneof$(X - X')$, oneof(Y);
    $Y' := \{y\}$;
    {**invariant** $(\forall y' : y' \in Y' - \{y\} : y' \not\subset x)$}
    do $y \not\subset x \wedge Y' \neq Y \rightarrow$
      $y :=$ oneof$(Y - Y')$;
      $Y' := Y' \cup \{y\}$
    od;
    if $y \subset x$ **then** $X := X - \{x\}$ **fi**;
    $X' := X' \cup \{x\}$
  od
fi
$\{X = (\textbf{Setof}\ x : x \in X_0 \wedge (\forall y : y \in Y : y \not\subset x))\}$

On termination, $X \subseteq X'$, so (from the invariant) $X' = X_0$, and the postcondition holds.

In the second (and dual) approach, for each member y of Y, all members x of X are removed for which $y \subset x$. At this stage of the development, there seems little reason to prefer this to Algorithm 3, or vice-versa. Therefore, we arbitrarily proceed with Algorithm 3 in the starting lineup, keeping the dual approach on the bench in case the former gets into foul trouble.

5 Refining the Filtering Algorithm

In seeking to improve the efficiency of Algorithm 3, we now exploit our freedom to impose a total order on set S (because it is finite), and thereby on its subsets X and Y. We choose to investigate lexicographic order, which presupposes that the members of S can be regarded as sequences rather than sets. We do so by (notionally) arranging the members of each set in S in numeric order.

We use $\sqsubset$ (read as "is a (strict) prefix of") for the irreflexive prefix relation, and $\prec$ (read as "precedes") for lexicographic order, restricted in each case to finite sets of integers. We first define two operators (having higher precedence than all logical operators and relations): for all sets z of integers and integers c, z **before** c (resp. z **after** c) is the set of all elements in z that are less than (resp. greater than) c. Then for all finite sets x, y of integers,

$$y \sqsubset x \equiv (\exists b : b \in x : y = x \textbf{ before } b)$$

$$y \prec x \equiv y \sqsubset x \vee (\exists a, b : a \in y \wedge b \in x : y \textbf{ before } a = x \textbf{ before } b \wedge a < b)$$

We leave it to the reader to check that these definitions accord with the familiar ones for sequences, and in particular that $\prec$ is a total order.

There is a neat relationship between the three relations $\subset$, $\sqsubset$ and $\prec$:

Theorem 1 *For all finite sets x, y of integers,*

$$y \prec x \wedge y \subset x \equiv y \sqsubset x.$$

Proof[1]: Let x, y be finite sets of integers. Then

$$\begin{array}{ll} & y \prec x \wedge y \subset x \\ = & \{\text{definition of } \prec\} \\ & (y \sqsubset x \vee (\exists a, b : a \in y \wedge b \in x : y \textbf{ before } a = x \textbf{ before } b \wedge a < b)) \wedge y \subset x \\ = & \{y \sqsubset x \Rightarrow y \subset x;\ \text{predicate calculus}\} \\ & y \sqsubset x \vee ((\exists a, b : a \in y \wedge b \in x : y \textbf{ before } a = x \textbf{ before } b \wedge a < b) \wedge y \subset x) \\ = & \{(y \textbf{ before } a = x \textbf{ before } b \wedge a < b) \Rightarrow a \notin x\} \\ & y \sqsubset x \vee ((\exists a, b : a \in y \wedge b \in x : y \textbf{ before } a = x \textbf{ before } b \wedge a < b \wedge a \notin x) \wedge y \subset x) \\ = & \{(\exists a : a \in y : a \notin x) \Rightarrow y \not\subset x;\ \text{predicate calculus}\} \\ & y \sqsubset x \vee \mathit{false} \\ = & y \sqsubset x. \ \square \end{array}$$

Suppose that $y \subset x$. Then $y \neq x$, and, from Theorem 1, $y \prec x \equiv y \sqsubset x$. So $y \not\sqsubset x \equiv x \prec y$, because $\prec$ is a total order. We have established a corollary: if $y \subset x$, then either $y \sqsubset x$ or $x \prec y$. We exploit this fact in two stages. In the first, before any filtering is performed as in Algorithm 2 (or its alternative if that is under later consideration), S is pre-filtered by Algorithm 2.init, which removes all members of S that have a prefix in S.

The algorithm employs functions "min", "max" and "next", which may be applied to a totally ordered set V of any type. Along with function "prev", which is used later, they are defined as follows. The implicitly associated total order is denoted $\lhd$. To make the functions total, top and bottom elements (written $\top$ and $\bot$ respectively) are introduced, and the total order extended appropriately. When V is a subset of U, we choose $\top$ and $\bot$ to be M and -1 respectively; when V is a set of subsets of U, we choose $\top$ and $\bot$ to be $\{M\}$ and $\{-1\}$ respectively.

$$\min(V) = (\textbf{Min } v : v \in V \cup \{\top\})$$

$$\max(V) = (\textbf{Max } v : v \in V \cup \{\bot\})$$

$$\text{prev}(V, v) = (\textbf{Max } v' : (v' \in V \wedge v' \prec v) \vee v' = \bot)$$

$$\text{next}(V, v) = (\textbf{Min } v' : (v' \in V \wedge v \prec v') \vee v' = \top)$$

[1] In this proof format, which we learned from E. W. Dijkstra, hints are enclosed in curly braces.

Algorithm 2.init

$\{S = S'\}$
"Remove all members of S which have a prefix in S":
$y := \min(S)$;
{**invariant** $S = S' - (\textbf{Setof}\ x : x \in S' \wedge x \prec \text{next}(S, y) \wedge$
$(\exists y' : y' \in S' : y' \preceq y \wedge y' \sqsubset x))$}
do $y \prec \max(S) \rightarrow$
$x := \text{next}(S, y)$;
if $y \subset x \rightarrow S := S - \{x\}$
[] $y \not\subset x \rightarrow y := x$
fi
od
$\{S = S_0 \wedge S_0 = S' - (\textbf{Setof}\ x : x \in S' \wedge (\exists y : y \in S' : y \sqsubset x))\}$

Since no member is ever added to a set L_n, it is henceforth assumed that the following assertion is invariantly and globally true:

$$(\forall n, m, x, y : \{n, m\} \subseteq [0, K] \wedge x \in L_n \wedge y \in L_m : y \not\sqsubset x).$$

When the composition of Algorithm 2.init and Algorithm 2 terminates, therefore, the desired set cover(S') has been computed, because

$$(\textbf{Union}\ n : n \in [0, K] : L_n) = \text{cover}(S_0) \wedge \text{cover}(S_0) = \text{cover}(S').$$

The second stage of exploiting (the corollary of) Theorem 1 involves only procedure *filter*: in searching for a member y of Y for which $y \subset x$, attention is restricted to values y which x (lexicographically) precedes.

Algorithm 4 (implementation of $filter(X, Y)$)
$\{X = X_0 \wedge (\forall x, y : x \in X \wedge y \in Y : y \not\sqsubset x)\}$
$x := \emptyset$;
{**invariant** $X = (\textbf{Setof}\ x' : x' \in X_0 \wedge x' \preceq x \wedge (\forall y : y \in Y : y \not\subset x')) \cup$
$(\textbf{Setof}\ x' : x' \in X_0 \wedge x \prec x')$}
do $x \neq \max(X) \wedge \text{next}(X, x) \prec \max(Y) \rightarrow$
$x := \text{next}(X, x)$;
if *covered*(x, Y) **then** $X := X - \{x\}$ **fi**
od
$\{X = (\textbf{Setof}\ x : x \in X_0 \wedge (\forall y : y \in Y : y \not\subset x))\}$
where
function *covered*(x, Y) **is**
{**pre** $x \prec \max(Y) \wedge (\forall y : y \in Y : y \not\sqsubset x)$}
{**post** *covered* $\equiv (\exists y : y \in Y : y \subset x)$}
$y := \text{next}(Y, x)$;
{**invariant** $(\forall y' : y' \in Y \wedge y' \prec y : y' \not\subset x)$}
do $y \not\subset x \wedge y \neq \max(Y) \rightarrow y := \text{next}(S, y)$ **od**;
covered $:= y \subset x$

Advantage may be taken of the fact that the sequence of values assigned to y by the first command in function *covered* is non-decreasing. A variable y' is introduced in *filter* which obeys the invariant

$$y' \in Y \cup \{\top\} \wedge \text{prev}(Y, y') \prec x.$$

It is initialized to $\min(Y)$, and updated with

do $y' \prec x \rightarrow y' := \text{next}(Y, y')$ **od**

before being passed to *covered* as the value with which to initialize y.

With X and Y implemented as (singly-) linked lists of their members, the worst-case complexity of Algorithm 4 is $O(|X| \times |Y|)$ set-inclusion tests. (It can be expected to make about half as many tests as does Algorithm 3.) With the members of each L_n implemented as arrays or linked lists of *their* members in ascending order, a straightforward set-inclusion test takes time linear in the cardinality of the larger set. (Unless stated otherwise, our measure of complexity is number of operations on a RAM with cells capable of containing integers of size $O(N \textbf{ max } M)$.)

The cost (in RAM operations) of Algorithm 2 (using Algorithm 4 for *filter*) is therefore linear in

$$\begin{array}{ll} & (\textbf{Sum } m : m \in [0, K) : (\textbf{Sum } n : n \in (m, K] : |L_m| \times |L_n| \times n)) \\ \leq & (\textbf{Sum } m : m \in [0, K) : |L_m| \times (\textbf{Sum } n : n \in [0, K] : |L_n| \times n)) \\ = & (\textbf{Sum } m : m \in [0, K) : |L_m| \times (\textbf{Sum } x : x \in S : |x|)) \\ \leq & N \times (\textbf{Sum } x : x \in S : |x|) \\ = & K' \times N^2 \end{array}$$

where K' is the average cardinality of a member of S (which, in general, is a function of N).

A set S of subsets of U can be put in lexicographic order in $O((\textbf{Sum } x : x \in S : |x|) + M)$ RAM operations (see [1])—knowing this was an incentive to investigate this choice of order. This dominates the cost of Algorithm 2.init. Finally, the command "Partition S by cardinality" can be implemented using a bucket-sort in time $O(N + K)$, so the total cost of our solution to Problem 1 is therefore $O(K' \times N^2 + M)$.

In the applied problem from [5] which motivated the abstract Problem 1, the members of S are small sets; specifically, their cardinalities are bounded by a small constant. Our solution does not exploit such input; we would like to transform it to do so.

Consider function *covered*. It potentially examines each y in Y such that $x \prec y$, and on average there are $\Theta(|Y|)$ such sets. Let each member of X have cardinality n, $n > 0$, and of Y cardinality m, $m \in [0, n)$—a fact which is not exploited by function *covered*. Let $x \circ m$ denote the set of m-element subsets of x: for all m, $m \in [0, |x|]$,

$$x \circ m = (\textbf{Setof } z : z \subseteq x \wedge |z| = m).$$

Now there is a subset of x in Y if and only if Y contains a member of $x \circ m$, which has cardinality C_m^n.[2] If this number is sufficiently small, it may be more efficient to explicitly search for an element common to $x \circ m$ and Y.

Although we shall later exploit the special nature of $x \circ m$, it is instructive to regard the problem of implementing this search as an instance of the problem tackled in the next section.

6 Digression: Finding an Element Common to Two Ordered Sequences

Consider the following problem:

Problem 2 *Given two finite non-decreasing sequences of elements from a totally ordered universe, determine whether they contain a common element, and if so report two respective positions that witness that fact.*

Problem 2 is both simple and natural, and we suppose that it surfaces in many contexts. Gries sets it as exercise 8 in [4, §16.3], and proceeds to generalize it—by having three sequences—to

[2] The least member of $x \circ m$ is a prefix of x, and thus cannot be in Y. But its exclusion would be unnatural.

a problem (due to Feijin) that he gives a William F. Buckleyesquesetting as the Welfare Crook problem.

Gries presents a neat solution to the Welfare Crook problem that is linear in the sum of the sizes of its (three) sequences, and we presume he had a similar solution in mind to our Problem 2. But note that if the smaller sequence has sufficiently few members, repeatedly using Binary Search for each of these over the larger sequence is faster. In the extreme case where the sequences have lengths 1 and N, the latter solution is $O(\log N)$, while the former is $\Theta(N)$ in the worst-case. Yet the linear solution is certainly preferable when the two sequences have lengths of the same order.

There is a market for a solution with a complexity that varies smoothly from logarithmic to linear as the length of the smaller sequence varies from 1 to that of the larger. We characterize such an algorithm as "opportunistic", because it exploits special properties of its input without compromising good worst-case performance. Perhaps the best-known opportunistic algorithm is the formidable Smoothsort ([2]), which may explain our feeling that opportunism has been somewhat neglected (in the realm of algorithms).

A solution is given below as Algorithm 5. It enjoys the property of being completely symmetric in its two sequences, viz. $A.i$, $i \in [0, N)$, and $B.j$, $j \in [0, M)$, and their associated variables. Note that for a sequence C of length L, we define $C.L$ to be $\top$ and $C.(-1)$ to be $\bot$.

The idea is to use Doubling Search followed by Binary Search—incorporated in function *loc*—to locate an element in a sequence, alternating in the main loop between the two sequences. Operator **cand** is non-strict left-to-right conjunction; it guards against its right hand argument being undefined.

Algorithm 5

$\{N \geq 0 \wedge M \geq 0 \wedge (\forall i : i \in [-1, N) : A.i \leq A.(i+1)) \wedge (\forall j : j \in [-1, M) : B.j \leq B.(j+1))\}$
$i, j := 0, 0;$
$\{$**invariant** $((\exists i', j' : i' \in [0, i] \wedge j' \in [0, j] : A.i' = B.j') \Rightarrow A.i = B.j) \wedge i \in [0, N] \wedge$
  $j \in [0, M] \wedge (B.j < A.i \Rightarrow A.(i-1) < B.j) \wedge (A.i < B.j \Rightarrow B.(j-1) < A.i)\}$
do $j < M \wedge A.i < B.j \rightarrow i := loc(B.j, i, A, N)$
 $[]$ $i < N \wedge B.j < A.i \rightarrow j := loc(A.i, j, B, M)$
od
$\{(i < N \wedge j < M \wedge A.i = B.j) \equiv (\exists i, j : i \in [0, N) \wedge j \in [0, M) : A.i = B.j)\}$
where
  function $loc(v, k, C, L)$ **is**
  $\{$**pre** $k = k_0 \wedge C.k < v \wedge v \leq C.L \wedge (\forall i : i \in [k, L) : C.i \leq C.(i+1))\}$
  $\{$**post** $loc \in (k_0, L] \wedge C.(loc-1) < v \wedge v \leq C.loc\}$
    $d := 1;$
    $\{$**invariant** $d \div 2 \in [0, L-k) \wedge C.(k + (d \div 2)) < v\}$
    do $d < L - k$ **cand** $C.(k+d) < v \rightarrow d := 2 \times d$ **od**;
    $k, h := k + (d \div 2), (k+d)$ **min** L;
    $\{$**invariant** $C.k < v \wedge v \leq C.h \wedge k \in [k_0, L) \wedge h \in (k_0, L]\}$
    do $k + 1 \neq h \rightarrow$
      $mid := (k+h) \div 2;$ $\{k < mid \wedge mid < h\}$
      if $C.m < v \rightarrow k := mid$
      $[]$ $v \leq C.m \rightarrow h := mid$
      fi
    od;
    $loc := k$

We measure the complexity of Algorithm 5 as the number T of all comparisons involving a member of each sequence. Suppose without loss of generality that $M \leq N$. Since variable j is (strictly) increasing, it is incremented at most M times. Since variables i and j are incremented alternately, variable i is also incremented at most M times. Hence the number of comparisons

exclusive of those occurring in evaluations of *loc* is $O(M)$. The cost of evaluating *loc* is $O(1+\log\delta)$, where δ is the result minus the value of the second argument. Let ΔJ (resp. ΔI) be the bag (i.e., multiset) of all *increments* of j (resp. i). Then since there are $O(M)$ evaluations,

$$T = O(M) + (\mathbf{Sum}\ \delta : \delta \in \Delta J : \log\delta) + (\mathbf{Sum}\ \delta : \delta \in \Delta I : \log\delta). \tag{4}$$

To bound the second term of (4), we can use the appealing theorem that for $M > 1$, the product of a bag of natural numbers summing to M is maximized when all members are either 2 or 3 (and there are at most two 2's). A proof of this may be found in [3]. It immediately follows that the second term is $O(M)$. Since there are at most M members of ΔI, and they sum to at most N, the third term is $O(M\log(3\ \mathbf{max}\ (N/M)))$, i.e., $O(M\log(1+N/M))$, which dominates the complexity of Algorithm 5.

Although Doubling Search is crucial to Algorithm 5, so also is Binary Search. The Binary Search can be eliminated by tail-recursively applying Doubling Search with new starting index k. There are two reasons for not doing so. The first is that the guard of the loop would be more expensive. The second is that whereas the given method has complexity $\Theta(\log d)$, where the given value is located d positions along the sequence, the contemplated one has complexity $O(\log^2 d)$!

When $M = O(1)$, $T = O(\log N)$; when $M = O(N)$, $T = O(N)$. The complexity of Algorithm 5 varies smoothly between the complexities of the best algorithms for the extreme cases. But Algorithm 5 exploits its input in another way: whenever one sequence has a group of elements that fall between successive elements of the other, the whole group will be skipped at logarithmic cost, in contrast to the linear cost of the approach in [4].

Algorithm 5 is truly opportunistic. It is made from fast, simple components with low overheads. Furthermore, it can be parameterized so that *all* pairs of respective positions with matching values can be obtained one at a time by making repeated calls, and without increasing the worst-case complexity. (In this extended problem, it is natural to assume that each sequence is strictly increasing, to avoid the possibility of $\Theta(N \times M)$ output pairs.) While not lobbying for its induction into the Algorithms Hall Of Fame, we do think it worthy of a place in the Useful Box.[3]

7 Opportunistic Filtering

Returning now to Algorithm 4, we transform function *filter* so as to exploit Algorithm 5. Because the latter requires random-access to its input, Y is viewed as a sequence $Y.i$, $i \in [0, |Y|)$, by making the following definition:

$$Y.i = \begin{cases} \min(Y), & \text{if } i = 0 \wedge 0 < |Y|, \\ \text{next}(Y, Y.(i-1)), & \text{if } i \in [1, |Y|). \end{cases} \tag{5}$$

The transformation of code and assertions is as follows. The version of Algorithm 4 used incorporates the code to maintain variable y' (as $Y.k$); the only other change is to strengthen the precondition to include information about the lengths of the members of X and Y. Function *covered* is now implemented with Algorithm 5: in the latter, $A.i$, $i \in [0, N)$ becomes $Y.i$, $i \in [k, |Y|)$, and $B.j$, $j \in [0, M)$ becomes $x \circ |Y.i|$.

Of course, $x \circ |Y.i|$ must not be explicitly generated—doing so would destroy the point of Algorithm 5. Fortunately, neither is it necessary to be able to efficiently generate an element given its index in the lexicographic ordering. (That is a challenge which we are willing to forego here.) It suffices to be able to locate $Y.i$ in $x \circ |Y.i|$. This capability is provided by function *lub*, which given y such that $|y| \leq |x|$, returns the minimum w in $x \circ |y| \cup \{\top\}$ such that $y \preceq w$, i.e., the least upper bound of y in $x \circ |y| \cup \{\top\}$, which *mathematical* function is written $\text{lub}(y, x)$.

Variable z plays the role of $B.j$ in Algorithm 5. Note that we have unfolded the loop of function *covered* in order to initialize z, exploiting the fact that

[3] An aptly-named resource featured in the A(ustralian)BC's children's TV program *Play School*.

$x \prec Y.k \Rightarrow \min(x \circ m) \prec Y.k.$

Algorithm 6 (implementation of $filter(X, Y)$)
$\{X = X_0 \wedge (\forall x, x', y, y' : \{x, x'\} \subseteq X \wedge \{y, y'\} \subseteq Y : |x| = |x'| \wedge |y| = |y'| \wedge |y| < |x| \wedge y \not\sqsubset x)\}$
$x, k := \emptyset, 0;$
$\{$**invariant** $X =$ (**Setof** $x' : x' \in X_0 \wedge x' \preceq x \wedge (\forall y : y \in Y : y \not\subset x')) \cup$
(**Setof** $x' : x' \in X_0 \wedge x \prec x') \wedge k \in [0, |Y|) \wedge Y.(k-1) \prec x\}$
do $x \neq \max(X) \wedge \text{next}(X, x) \prec \max(Y) \rightarrow$
$x := \text{next}(X, x);$
do $Y.k \prec x \rightarrow k := k + 1$ **od**;
if $covered(x, Y, k)$ **then** $X := X - \{x\}$ **fi**
od
$\{X =$ (**Setof** $x : x \in X_0 \wedge (\forall y : y \in Y : y \not\subset x))\}$
where
function $covered(x, Y, k)$ **is**
$\{$**pre** $x \prec \max(Y) \wedge (\forall y, y' : \{y, y'\} \subseteq Y : |y| = |y'| \wedge |y| < |x| \wedge y \not\sqsubset x) \wedge$
$k \in [0, |Y|) \wedge Y.(k-1) \prec x \wedge x \prec Y.k)\}$
$\{$**post** $covered \equiv (\exists y : y \in Y : y \subset x)\}$
$N, m, i, z := |Y|, |\text{oneof}(Y)|, k, lub(Y.k, x);$
$\{$**invariant** $((\exists i', z' : i' \in [0, i] \wedge z' \in x \circ m : z' \preceq z \wedge Y.i' = z) \Rightarrow Y.i = z) \wedge i \in [k, N] \wedge$
$z \in x \circ m \cup \{\top\} \wedge (z \prec Y.i \Rightarrow Y.(i+1) \prec z) \wedge (Y.i \prec z \Rightarrow \text{prev}(z, x \circ m) \prec Y.(i))\}$
do $z \prec \top \wedge Y.i \prec z \rightarrow i := loc(z, i, Y, N)$
$[]\ i < N \wedge z \prec Y.i \rightarrow z := glb(Y.i, x)$
od;
$\{(i < N \wedge z \prec \top \wedge Y.i = z) \equiv (\exists i, z : i \in [k, N) \wedge z \in x \circ m : Y.i = z)\}$
$covered := Y.i = z$
where
function $lub(y, x)$ **is**
$\{$**pre** $|y| \leq |x|\}$
$\{$**post** $lub = \text{lub}(y, x)\}$
function $loc(v, k, C, L)$ **is** *as in Algorithm 5*

Computing $\text{lub}(y, x)$ is a pretty problem. We prepare for a solution by looking for a formal characterization. Suppose $|y| \leq |x|$ and $l = \text{lub}(y, x)$. Then $y \preceq l$, and applying the definition of $\prec$ gives

$$y \sqsubseteq l \vee (\exists a, b : a \in y \wedge b \in l : y \textbf{ before } a = l \textbf{ before } b \wedge a < b).$$

Since $l = \top \vee l \in x \circ |y|$, and $|l| = |y| \Rightarrow y \not\sqsubset l$,

$$l = \top \vee l = y \vee (\exists a, b : a \in y \wedge b \in x : y \textbf{ before } a = l \textbf{ before } b \wedge a < b \wedge$$
$$y \textbf{ before } a \subset x \wedge |y \textbf{ from } a| \leq |x \textbf{ from } b|).$$

Suppose that $l \neq \top$ and $l \neq y$, and let a be a witness to the existentially quantified formula above. Then there is an upper bound l' with

$$l' \textbf{ after } a = (x \textbf{ after } a) \textbf{ first } |y \textbf{ from } a|.$$

But $l \preceq l'$, so $b = \text{next}(x, a)$, and

$$(\exists a : a \in y : l = (y \textbf{ before } a) \cup ((x \textbf{ after } a) \textbf{ first } |y \textbf{ from } a|) \wedge$$
$$y \textbf{ before } a \subset x \wedge |y \textbf{ from } a| \leq |x \textbf{ after } a|). \tag{6}$$

Finally, for l to be the *least* upper bound, a must be maximal, which completes the characterization of l in this case.

Suppose we weaken the conditions on a by only requiring $a \in y \cup \{-1, M\}$ and y **before** $a \subseteq x$. Then when $l = \top$, $a = -1$; and when $l = y$, $a = M$. Otherwise, $a \neq M$, and $x \neq \emptyset$ because $|x| = 0$ implies $|y| = 0$ implies $l = y$, and therefore a satisfies the conditions in (6) if and only if it satisfies the weaker ones, because

$$(a \in y \wedge y \textbf{ before } a \subseteq x \wedge |y| = |x|) \Rightarrow y \textbf{ before } a \subset x.$$

We have now established our characterizing theorem.

Theorem 2 *For all x, y such that $|y| \leq |x|$,*

$$\text{lub}(y, x) = \begin{cases} (y \textbf{ before } a) \cup ((x \textbf{ after } a) \textbf{ first } |y \textbf{ from } a|), & \text{if } a \in y \cup \{M\}, \\ \top, & \text{if } a = -1, \end{cases} \tag{7}$$

where

$$a = (\textbf{Max } a' : a' \in y \cup \{-1, M\} \wedge y \textbf{ before } a' \subseteq x \wedge |y \textbf{ from } a'| \leq |x \textbf{ after } a'|). \tag{8}$$

□

The case $lub(y, x) = y$ if $a = M$ has been subsumed in the first clause of (7).

Equation (7) reduces the problem of computing $lub(y, x)$ to that of computing a. Let (8) have the form $a = (\textbf{Max } a' : Q(a'))$. The second conjunct of Q suggests that a should be found, if possible, with a left-to-right scan of y, allowing y **before** $a' \subseteq x$ to be computed incrementally. This would be consistent with computing a maximum if

$$Q(a) \equiv (\forall a' : a' \in [-1, a] : Q(a')),$$

because the search would then be for the *minimum* a'' (if any) such that $\neg Q(a'')$. Unfortunately, this equivalence does not hold. E.g., when $x = \{0, 1, 3\}$ and $y = \{1, 2\}$, $a = 2$ but $\neg Q(1)$.

We therefore decide to compute a by first computing

$$(\textbf{Max } a' : a' \in y \cup \{-1, M\} \wedge y \textbf{ before } a' \subseteq x)$$

with a left-to-right scan, and then completing the computation with a right-to-left scan. Our high-level solution is

$a := \min(y)$;
{**invariant** $a \in y \cup \{M\} \wedge y$ **before** $a \subseteq x$}
do $a \in x \rightarrow a := \text{next}(y, a)$ **od**;
{**invariant** $a \in y \cup \{-1, M\} \wedge y$ **before** $a \subseteq x \wedge (\textbf{Max } a' : Q(a')) \leq a$}
do $|y \textbf{ from } a| > |x \textbf{ after } a| \rightarrow a := \text{prev}(y, a)$ **od**
{(8)}

It is straightforward to refine and extend this algorithm to obtain an algorithm that computes $\text{lub}(y, x)$ in $O(|x|)$ RAM-operations, which is optimal in the worst-case. Nevertheless, it can be needlessly inefficient. E.g., when $x = [0, n)$ and $y = (0, n]$, $(n > 0)$, the first loop examines every member of y, yet it is apparent at the start that $a = -1$, because

$$(a \in y \wedge y \textbf{ before } a \subseteq x \wedge |x \textbf{ after } a| < |y \textbf{ from } a| - 1) \Rightarrow (\textbf{Max } a' : Q(a')) < a.$$

(To see this, observe that if $a' \in y \cup \{M\}$ and $a' > a$, then

$$|x \textbf{ after } a'| - |y \textbf{ from } a'| \leq |x \textbf{ after } a| - |y \textbf{ from } a| + 1,$$

with equality only if $a' \notin x$, because every member of y between a and a' exclusive must also be in x.)

The upshot is that an opportunistic algorithm may be obtained by strengthening the guard of the first loop by adding the conjunct

$|x \textbf{ after } a| \geq |y \textbf{ from } a| - 1.$

Furthermore, by maintaining the maximum a' such that $a' < a \wedge Q(a')$, the need for the second loop disappears.

This completes the high-level design of our algorithm for computing $\text{lub}(y, x)$. An implementation is given below as Algorithm 7. It should be merely tedious to verify its correctness from the supplied assertions.

Algorithm 7 (implementation of $lub(y, x)$)

$\{|y| \leq |x|\}$
$a, b, i, j, a' := \min(y), \min(x), |y|, |x|, -1;$
$\{\textbf{invariant } a \in y \cup \{M\} \wedge b \in x \cup \{M\} \wedge i = |y \textbf{ from } a| \wedge j = |x \textbf{ from } b| \wedge i \leq j \wedge$
$\quad y \textbf{ before } a \subseteq x \textbf{ before } b \wedge \text{prev}(x, b) < a \wedge a' = (\textbf{Max } a' : a' < a \wedge Q(a'))\}$
do $b < a \wedge 0 < i \wedge i < j \rightarrow b, j := \text{next}(x, b), j - 1$
$[]\ b = a \wedge 0 < i \rightarrow\ \{Q(a) \equiv i < j\}$
$\quad$ **if** $i < j$ **then** $a' := a$ **fi**;
$\quad a, b, i, j := \text{next}(y, a), \text{next}(x, b), i - 1, j - 1$
od;
$\{(i > 0 \wedge b < a \wedge (8)\langle a := a'\rangle) \vee ((i = 0 \vee (i > 0 \wedge b > a)) \wedge (8))\}$
if $i > 0 \wedge b < a$ **then** $a := a'$ **fi**;
"Establish lub = right hand side of first clause of (7)";
$\{lub = \text{lub}(y, x)\}$

It is difficult to see how the problem of computing $\text{lub}(y, x)$ could be successfully tackled without a formal postcondition like that afforded by Theorem 2 (and a methodology of algorithm design capable of exploiting it).

In determining bounds on the complexity of our new solution, we shall assume that each L_n is implemented as an array of arrays or doubly-linked lists, so that most of the low-level operations cost $O(1)$ RAM operations (as does $z \prec \top$). A set comparison $v \prec w$ costs $O(|v| \textbf{ min } |w|)$, evaluation of $lub(y, x)$ costs $O(|x|)$, and evaluation of $loc(z, i, \ldots)$ costs $O((1 + \delta) \times |z|)$, where δ is the difference between the result and the given value of i.

It follows that the cost of executing $filter(X, Y)$ is dominated by the cost of the $|X|$ evaluations of $covered(x, Y, k)$. Let $x \in L_n$ and $Y = L_m$. The analysis of the cost of evaluating *covered* is similar to that used for Algorithm 5. But there are now two cases, corresponding to which of $x \circ m$ and $Y.i$, $i \in [k, |Y|)$ is larger, i.e., to which of C_m^n and $|Y| - k - 1$ is larger. Suppose first that $|Y| \leq C_m^n$. Then there are $O(|Y|)$ iterations of the loop, at a total cost of $O(n \times |Y|)$. Now suppose that $|Y| \geq C_m^n$. Then there are $O(C_m^n)$ iterations of the loop, at a total cost of $O(C_m^n \times (n + m \times \log(|Y|/C_m^n)))$, which is also $O(n \times |Y|)$, because $x \log(B/x)$, $B > 0$, is an increasing function of x, $x > 0$.

Summing over all executions of $filter(X, Y)$ gives a total cost for Algorithm 6 of order

$$(\textbf{Sum } m : m \in [0, K) : (\textbf{Sum } n : n \in (m, K] : |L_n| \times (|L_m| \textbf{ min } C_m^n) \times (n + m \log \frac{|L_m|}{|L_m| \textbf{ min } C_m^n}))), \tag{9}$$

which is also $O(K' \times N^2)$. Algorithm 6 is never slower than Algorithm 2 (using Algorithm 4 for *filter*) by more than a constant factor. But when most members of S are small, Algorithm 6 will be considerably faster.

More precisely, consider expression (9). It is bounded by

$$(\textbf{Sum } m : m \in [0, K) : (\textbf{Sum } n : n \in (m, K] : |L_n| \times C_m^n \times n)) +$$

$$(\textbf{Sum } m : m \in [0, K) : (\textbf{Sum } n : n \in (m, K] : |L_n| \times C_m^n \times m \times \log |L_m|)).$$

Now the first summand

$$\begin{aligned}
&= (\textbf{Sum } n : n \in [1, K] : (\textbf{Sum } m : m \in [0, n) : |L_n| \times C_m^n \times n)) \\
&= (\textbf{Sum } n : n \in [1, K] : |L_n| \times n \times (\textbf{Sum } m : m \in [0, n) : C_m^n)) \\
&= \{(\textbf{Sum } m : m \in [0, n] : C_m^n) = 2^n;\ C_n^n = 1\} \\
&\quad (\textbf{Sum } n : n \in [1, K] : |L_n| \times n \times (2^n - 1)) \\
&\leq K \times 2^K \times N.
\end{aligned}$$

And the second summand

$$\begin{aligned}
&= (\textbf{Sum } n : n \in [1, K] : (\textbf{Sum } m : m \in [0, n) : |L_n| \times C_m^n \times m \times \log N)) \\
&= \log N \times (\textbf{Sum } n : n \in [1, K] : |L_n| \times (\textbf{Sum } m : m \in [0, n) : m \times C_m^n)) \\
&= \{C_m^n = C_{n-m}^n\} \\
&\quad \log N \times (\textbf{Sum } n : n \in [1, K] : |L_n| \times (\textbf{Sum } m : m \in [0, n) : n \times C_m^n))/2 \\
&= \log N \times (\textbf{Sum } n : n \in [1, K] : |L_n| \times n \times (2^{n-1} - 1/2)) \\
&\leq K \times 2^{K-1} \times N \log N.
\end{aligned}$$

In the problem that motivated this paper, each member of S has cardinality bounded by a small constant K. Therefore in this case the complexity of the solution using Algorithm 6 is $O(N \log N + M)$, whereas Algorithm 2's is $O(N^2 + M)$. But the crudeness of the preceding estimates don't do justice to Algorithm 6: as shown by its complexity $O((9) + M)$, it can take advantage of its input in many sub-computations. For instance, it will also exploit input where almost all members of S have cardinalities in a narrow range, because then almost all of the binomial co-efficients will be small. Algorithm 6 is another opportunistic algorithm—one which we think is a satisfactory solution to Problem 1.

8 Tinkering

Let us briefly explore some possibilities for speeding up our solution. On the micro-level, the reader with an appreciation for the pervasiveness of Problem 2 will have realized that Algorithm 5 can be used to implement an opportunistic subset test. But such a test is no longer used, having been subsumed in function *lub*. However, the loop in the body of that function can be replaced by repeated calls of Algorithm 5 (and a little housekeeping), provided set x is implemented as an array. The upshot is that there is an opportunistic algorithm for computing $\text{lub}(y, x)$ with complexity $O(|y| \times \log(1 + |x|/|y|))$. This is significant when $|y| \ll |x|$, but such cases are in the minority, and furthermore, the opportunism of Algorithm 6 diminishes their importance, because the corresponding binomial coefficients are relatively small. Although there are infinite classes of input for which the resulting solution would have a lower order of complexity, they are unlikely to occur in practice, and we are content to class this as a "theoretical" improvement.

An opportunity for a higher-level program transformation is provided by the fact that in Algorithm 2, the order in which a set L_m filters the sets L_n, $n \in (m, K]$, is not fixed. We can attempt to exploit this by carrying out all these filterings together, interleaving execution so as to maximize the sharing of computations. One way of doing so is to filter the k'th smallest elements of the L_n's together, maintaining the bag of corresponding values of z in Algorithm 6 as a priority queue. The hope is that the cost of maintaining the queue will pay dividends in the form of a decreased total cost of stepping through L_m in all filterings (because steps will be smaller). However, when $|L_m| \leq C_m^n$, the worst-case cost of filtering a member of L_n blows up to

$$O(|L_m| \times (n + m \log(K - m))),$$

so the resulting algorithm may be a factor of $O(\log K)$ slower than our basic "quadratic" solution. Since the decreased cost of stepping through L_m will be manifested in a larger (than C_m^n) denominator in the log-term of (9), and has to balance the multiplier $\log(K-m)$, precise analysis is tricky and depends on constant factors. Given all of this, this transformation is not justified.

Although Algorithm 6 filters the members of X in increasing order, it need not. (It does so for the minor gain of maintaining $Y.k$.) So let us investigate filtering all the members of X in parallel and interleaving the operations for our convenience. Let $X = L_n$ and $Y = L_m$. When $|L_m| \geq C_m^n$, each m-element subset of each member of L_n may be computed. A convenient arrangement is for all of the j'th smallest m-element subsets to be processed together, for each $j \in [1, C_m^n]$, because they can be quickly bin-sorted and then batch-processed with Algorithm 5 (because Problem 2 emerges yet again).

The total cost of filtering L_n with L_m in this manner is

$$\begin{aligned} & O(C_m^n \times (m \times |L_n| + M + (|L_m| \ \mathbf{min}\ |L_n|) \times m \times \log(1 + \frac{|L_m| \ \mathbf{max}\ |L_n|}{|L_m| \ \mathbf{min}\ |L_n|}))) \\ = \ & O(|L_n| \times C_m^n \times m \times (1 + \log(1 + |L_m|/|L_n|))) + O(C_m^n \times M). \end{aligned} \tag{10}$$

Let us compare this new approach with Algorithm 6 by calculating the ratio of their respective complexities; for the moment, we ignore the second term of (10), and divide its first term by the innermost term of (9). When $|L_m| \leq C_m^n$, the ratio is $O((m \times C_m^n)/(n \times |L_m|))$, showing that the new approach is inferior when L_m is very small; conversely, when $C_m^n \leq |L_m|$, the ratio is

$$O\left(\frac{1 + \log(1 + |L_m|/|L_n|)}{n/m + \log(|L_m|/C_m^n)}\right),$$

showing that the new approach is superior in some circumstances, such as when m is small, n is large, and L_n is sufficiently large.

In isolation, the transformed algorithm is not opportunistic, because the worst-case complexity is increased. But since its cost can be estimated accurately, it is possible to obtain a theoretically superior opportunistic algorithm by simply having it choose the faster of the two competing approaches for each filtering step. However, our analysis ignored the second term of (10); if it is prohibitively large, a factor of $\log |L_n|$ must be introduced for the sorting, and this will effectively rule out the contemplated transformation.

9 Closing Remarks

Two benefits of a parsimonious programming methodology are that the programmer can afford to thoroughly understand his or her tools, and that he or she has maximum freedom at each stage of refinement. We point to three examples in the text. The first is the use of a loop (the **forall** loop) that iterates over a set in unspecified order—see Algorithms 1 and 2. The second is the use of function "oneof", together with a set which accumulates its values, to implement a search (which may terminate early) over a set in unspecified order—see Algorithm 3. The third is the use of function "next" in Algorithm 7, rather than using indices and thereby imposing a random-access structure on the sets to which it is applied.

The freedom to exploit different time-orderings of operations is used in several places. Although no outstanding gains are made as a result, this is not always the case—see [6] and, especially, [7]. When an essentially arbitrary choice of time-ordering is made, as in §§3,4, it is good practice to at least document the alternative(s), so that they need not be rediscovered in later attempts to obtain different solutions.

The problem tackled in this paper was new to us. Especially in such situations, it is incumbent on the programmer to pay special attention to notation. We have tried to make it clear, natural, and conducive to manipulation, and point to Theorem 2 as evidence of some success.

Finally, we point out that the theorem from which Algorithm 7 was developed was not a mathematical rabbit pulled out of a hat, but rather was itself developed from the relevant definition, letting the symbols do the work. A fully mature and convincing methodology of program construction must include one of mathematical discovery, lest one mystery be explained away with another. This is a challenging task which must be taken up by the Computer Science educator, both because of the special nature of the mathematics, and because Mathematicians seem to have displayed little enthusiasm for the task, for all sorts of reasons which we leave it to the reader to ponder.

Acknowledgements

In non-decreasing order of gratitude, I wish to thank the University of Texas at Austin, its Department of Computer Sciences, the members of the Austin Tuesday Afternoon Club other than myself (who ATACed part of an early version of this paper), Jay Misra, and Edsger Dijkstra, for contributing to an enjoyable and educational stay.

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] E. W. Dijkstra. Smoothsort, an alternative for sorting in situ. *Science of Computer Programming*, 1:223–233, 1982.

[3] E. W. Dijkstra. To hell with "meaningful identifiers"! Epistle EWD 1044, Dept of Computer Sciences, The University of Texas at Austin, February 1989.

[4] D. Gries. *The Science of Programming.* Springer-Verlag, New York, 1981.

[5] P. Pritchard. Algorithms for finding matrix models of propositional calculi. Submitted to *Journal of Automated Reasoning.*

[6] P. Pritchard. Another look at the "longest ascending subsequence" problem. *Acta Informatica*, 16:87–91, 1981.

[7] P. Pritchard. Linear prime-number sieves: A family tree. *Science of Computer Programming*, 9:17–35, 1987.