SIAM J. COMPUT.
Vol. 16, No. 6, December 1987

© 1987 Society for Industrial and Applied Mathematics

Downloaded 12/25/12 to 152.3.102.242. Redistribution subject to SIAM license or copyright; see http://www.siam.org/journals/ojsa.php

THREE PARTITION REFINEMENT ALGORITHMS*

ROBERT PAIGE† AND ROBERT E. TARJAN‡

Abstract. We present improved partition refinement algorithms for three problems: lexicographic sorting, relational coarsest partition, and double lexical ordering. Our double lexical ordering algorithm uses a new, efficient method for unmerging two sorted sets.

Key words. partition, refinement, lexicographic sorting, coarsest partition, double lexical ordering, unmerging, sorted set, data structure

AMS(MOS) subject classifications. 68P05, 68P10, 68Q25, 68R05

1. Introduction.

The theme of this paper is partition refinement as an algorithmic paradigm. We consider three problems that can be solved efficiently using a repeated partition refinement strategy. For each problem, we obtain a more efficient algorithm than those previously known. Our computational model is a uniform-cost sequential random access machine [1]. All our resource bounds are worst-case. We shall use the following terminology throughout the paper. Let U be a finite set. We denote the size of U by $|U|$. A *partition* P of U is a set of pairwise disjoint subsets of U whose union is all of U. The elements of P are called its *blocks*. If P and Q are partitions of U, Q is a *refinement* of P if every block of Q is contained in a block of P. As a special case, every partition is a refinement of itself.

The problems we consider and our results are as follows:

(i) In § 2, we consider the *lexicographic sorting problem*: given a multiset U of n strings over a totally ordered alphabet Σ of size k, sort the strings in lexicographic order. Aho, Hopcroft and Ullman [1] presented an $O(m+k)$ time and space algorithm for this problem, where m is the total length of all the strings. We present an algorithm running in $O(m'+k)$ time and $O(n+k)$ auxiliary space (not counting space for the input strings), where m' is the total length of the *distinguishing prefixes* of the strings, the smallest prefixes distinguishing the strings from each other.

(ii) In § 3, we consider the *relational coarsest partition problem*: given a partition P of a set U and a binary relation E on U, find the coarsest refinement Q of P such that for each pair of blocks B_1, B_2 of Q, either $B_1 \subseteq E^{-1}(B_2)$ or $B_1 \cap E^{-1}(B_2) = \emptyset$. Here $E^{-1}(B_2)$ is the preimage set, $E^{-1}(B_2) = \{x \mid \exists y \in B_2 \text{ such that } xEy\}$. Kanellakis and Smolka [9] studied this problem in connection with equivalence testing of CCS expressions [14]. They gave an algorithm running in $O(mn)$ time and $O(m+n)$ space, where m is the size of E (number of related pairs) and n is the size of U. We propose an algorithm running in $O(m \log n)$ time and $O(m+n)$ space.

* Received by the editors May 19, 1986; accepted for publication (in revised form) March 25, 1987.

† Computer Science Department, Rutgers University, New Brunswick, New Jersey 08903 and Computer Science Department, New York University/Courant Institute of Mathematical Science, New York, New York 10012. Part of this work was done while this author was a summer visitor at IBM Research, Yorktown Heights, New York 10598. The research of this author was partially supported by National Science Foundation grant MCS-82-12936 and Office of Naval Research contract N00014-84-K-0444.

‡ Computer Science Department, Princeton University, Princeton, New Jersey 08544 and AT&T Bell Laboratories, Murray Hill, New Jersey 07974. The research of this author was partially supported by National Science Foundation grants MCS-82-12936 and DCR-86-05962 and Office of Naval Research contract N00014-84-K-0444.

Downloaded 12/25/12 to 152.3.102.242. Redistribution subject to SIAM license or copyright; see http://www.siam.org/journals/ojsa.php

(iii) In § 4, we consider the *double lexical ordering problem*: given an $n \times k$ nonnegative matrix M containing m nonzero entries, independently permute the rows and columns of M so that both the rows and the columns are in nonincreasing lexicographic order. Lubiw [11] defined this problem and gave an $O(m(\log(n+k))^2+n+k)$ time and $O(m+n+k)$ space algorithm. We develop an implementation of her algorithm running in $O(m \log(n+k)+n+k)$ time and $O(m+n+k)$ space. Our method uses a new, efficient algorithm for unmerging two sorted sets.

We conclude in § 5 with some miscellaneous remarks. Although our algorithms use different refinement strategies tuned for efficiency, the similarities among them are compelling and suggest further investigation of the underlying principles.

2. Lexicographic sorting. Let Σ be an alphabet of $k \geqq 1$ symbols and let Σ^* be the set of finite length strings over Σ. The empty string is denoted by λ. For $x \in \Sigma^*$ we denote the ith symbol of x by $x(i)$ and the length of x (number of symbols) by $|x|$. Symbol $x(i)$ is said to be in *position* i of string x. For any $x, y \in \Sigma^*$, the *concatenation* of x and y, denoted by xy, is a new string z of length $|x|+|y|$, in which $z(i)=x(i)$ for $1 \leqq i \leqq |x|$ and $z(|x|+i)=y(i)$ for $1 \leqq i \leqq |y|$. For any $x, y \in \Sigma^*$, x is a *prefix* of y iff $y=xz$ for some string $z \in \Sigma^*$. Note that every string is a prefix of itself. If x is a prefix of y but $x \neq y$, then x is a *proper prefix* of y. Let $<$ be a total ordering of Σ. We extend $<$ to a total ordering of Σ^*, called *lexicographic order*, by defining $x \leqq y$ iff either (i) there exists j such that $x(j)<y(j)$ and $x(i)=y(i)$ for $1 \leqq i<j$, or (ii) x is a prefix of y.

Let $\Sigma=\{1, 2, \cdots, k\}$, let $<$ be numeric order on $\Sigma \cup \{0\}$, and let Σ^*0 denote the set of strings formed by appending 0 to the end of each string in Σ^*. Our formulation of the *lexicographic sorting problem* is that of arranging a multiset of strings $U \subseteq \Sigma^*0$ in increasing lexicographic order. Note that a lexicographic arrangement of strings is the same whether all or none have 0 end markers.

In considering this problem we shall denote the strings in U by $x_1, x_2, \cdots, x_n$ and their total length $\sum_{i=1}^{n} |x_i|$ by m. For each string $x_i \in U$, the *distinguishing prefix* of x_i in U, denoted by x_i', is either

(i) the shortest prefix of x_i that is not a prefix of any other string in U if x_i occurs uniquely in U; or

(ii) x_i if x_i has multiple occurrences in U.

Because the end marker 0 can only occur in the last position of a string, no string in U can be a proper prefix of any other string in U. Thus, if $x, y \in U$ and $x \neq y$, we know that $x' \neq y'$. Consequently, if $x, y \in U$, then $x \leqq y$ iff $x' \leqq y'$, so that sorting the distinguishing prefixes is both necessary and sufficient to sort U. If we denote the total length $\sum_{i=1}^{n} |x_i'|$ of the distinguishing prefixes by m', then lexicographic sorting requires $\Omega(m')$ time. Note that $|x_i'| \geqq 1$ by the definition of x_i', which implies $m' \geqq n$.

Aho, Hopcroft and Ullman [1] presented a lexicographic sorting algorithm running in $O(m+k)$ time and space. Their algorithm uses a multipass radix sort, processing the strings from last position to first and using appropriate preprocessing to determine the set of buckets occupied during each pass of the radix sort. Mehlhorn [13] studied the special case of lexicographic sorting in which all strings have the same length. He concluded that a straightforward algorithm scanning strings from first position to last must use $\Omega(km)$ time (see [13, exercise 18, p. 100]).

We exhibit a lexicographic sorting algorithm that scans strings in the direction from first to last position and runs in $O(m'+k)$ time, thereby improving the Aho, Hopcroft and Ullman algorithm. The auxiliary space required by our algorithm (not counting space for the input strings) is $O(n+k)$, also an improvement. Our algorithm

Downloaded 12/25/12 to 152.3.102.242. Redistribution subject to SIAM license or copyright; see http://www.siam.org/journals/ojsa.php

requires fewer passes than the Aho, Hopcroft and Ullman algorithm and may be superior in practice as well as in theory. For an appropriately defined probability model, m' is on the order of $n \log_k n$; thus in situations where m is much larger than $n \log_k n$, an improvement from $O(m+k)$ to $O(m'+k)$ is significant.

The first and most important step in obtaining an efficient sorting algorithm is to separate the concern of finding the distinguishing prefixes from that of sorting the prefixes. Thus our algorithm consists of two steps:

(1) Determine the distinguishing prefixes of all the strings.

(2) Lexicographically sort these prefixes.

To describe the first step, we use the following definitions. We partition U into labeled blocks B_α. Block B_α is the multiset of all strings in U with prefix α; e.g., $B_\lambda = U$. We call α the *associated prefix* of B_α. If α and β are strings and α is a prefix of β, then $B_\alpha \supseteq B_\beta$. Block B_α is *finished* iff $\alpha = x'$ for some string $x \in U$. Thus we can determine the distinguishing prefixes for the strings in U by finding all of the finished blocks. A block that is not finished is said to be *unfinished.*

Let P be a set of labeled blocks that partition U. We say that P is *finished* if it consists of all the finished blocks; otherwise, it is *unfinished.* If P contains an unfinished block B_α, then $split(B_\alpha, P)$ denotes the refinement of P that results from replacing block B_α by the blocks in the set $\{B_{\alpha u} \mid u \in \Sigma \cup \{0\}$ *such that* $\exists x \in B_\alpha$ *for which* $x(|\alpha|+1) = u\}$.

To determine all the finished blocks, we start with an initial partition $P = \{B_\lambda\}$ and successively refine P by executing the following step until P is finished:

Refine. Find an unfinished block $B_\alpha \in P$; replace P by $split(B_\alpha, P)$.

LEMMA 1. *The refinement algorithm maintains the invariant that the partition F of finished blocks is a refinement of P, and for all* $B_\beta \in F$ *there exists* $B_\alpha \in P$ *such that* α *is a prefix of* β.

Proof. The proof is by induction on the number of refinement steps. The lemma is true for B_λ. Assume it is true after $k \geqq 0$ steps. Let B_β be a finished block. By hypothesis, just after the kth step there is a block $B_\alpha \in P$ such that α is a prefix of β, and so $B_\beta \subseteq B_\alpha$. If B_α is finished, so that $\alpha = \beta$, then the invariant holds after the next step. If B_α is unfinished, then α is a proper prefix of β. Hence, $split(B_\alpha, P)$ is defined and contains a block $B_{\alpha u}$ such that αu is a prefix of β, which implies $B_\beta \subseteq B_{\alpha u}$. □

THEOREM 1. *The algorithm terminates and is correct.*

Proof. Lemma 1 implies that the algorithm terminates and is correct. Also by Lemma 1, the sum of the lengths of all the associated prefixes of blocks in P can be no greater than m'. Since each refinement step increases this sum, the algorithm terminates after at most m' refinement steps. □

We make several observations that facilitate an efficient implementation of the preceding algorithm. By Lemma 1 and the definition of distinguishing prefixes, the algorithm maintains the invariant that a block $B_\alpha \in P$ is unfinished iff $|B_\alpha| \neq 1$ and $\alpha(|\alpha|) \neq 0$. Thus we can test whether a block is finished or not in constant time, and we can also maintain the set S of unfinished blocks in P efficiently. Although we could implement each block B_α by an index $I = |\alpha|$ and a list of pointers to strings, by implementing S as a queue and using a breadth-first strategy to refine the unfinished blocks of S, we can make I a global variable.

That is, we use a repeated partition refinement strategy that scans the strings starting from the first position and partitions the strings into blocks on the basis of their scanned prefixes. The algorithm maintains three objects: a partition P of the strings, a set S containing the unfinished blocks of P, and an integer I indicating the

Downloaded 12/25/12 to 152.3.102.242. Redistribution subject to SIAM license or copyright; see http://www.siam.org/journals/ojsa.php

current position being examined in the strings. Initially, $P=\{U\}$, $S=\{U\}$, and $I=0$. The algorithm consists of repeating the following step until S is empty:

Scan Blocks. Replace I by $I+1$. Let $S'=S$ and replace S by ϕ. Repeat the following step for each block B_α in S'.

Refine. Remove B_α from S'. Replace P by $split(B_\alpha, P)$. Add each new unfinished block to S. The distinguishing prefix of any string belonging to each new finished block $B_{\alpha u}$ is αu, which is of length I.

The preceding algorithm maintains the invariant $|\alpha|=I-1$ on entry to the *refine* step. The effect of one iteration of the step *scan blocks* is to refine the partition P based on the Ith symbol in each string. The correctness of the algorithm is obvious. We can implement the algorithm to run in $O(m')$ time and $O(n+k)$ auxiliary space as follows. Let B be a block selected for refinement. To partition B, we use an auxiliary array of buckets numbered 0 through k. We insert the strings in B into the buckets corresponding to their Ith symbols, simultaneously making a list of all nonempty buckets. (When a string is added to an empty bucket, we add that bucket to the list of nonempty buckets.) Once the strings are distributed among the buckets, we examine each nonempty bucket, forming a new block from the contents and emptying the bucket.

When moving strings from place to place, we do not move the strings themselves but rather pointers to them; thus each string movement takes $O(1)$ time. We can avoid initializing each bucket in the array of buckets to empty by using the solution to exercise 2.12 of Aho, Hopcroft and Ullman's book [1]. The total time for the partitioning algorithm is then $O(m')$, since each symbol of each distinguishing prefix is scanned once. If we explicitly initialize the array of buckets before the first refinement step, the time for partitioning, including initialization, is $O(m'+k)$. The auxiliary space needed by the partitioning algorithm is $O(1)$ words per string plus $k+1$ words for the array of buckets, for a total of $O(n+k)$.

To carry out step two, the sorting of the prefixes, one option is to use the lexicographic sorting algorithm of Aho, Hopcroft and Ullman. This method, however, uses $O(m'+k)$ auxiliary space. If we compute some extra information during step one, we can reduce the space needed and considerably simplify the sorting task.

We distinguish between two kinds of refinements. Let us call a refinement a *proper refinement* if the refined block B is actually split into two or more smaller blocks and an *improper refinement* if B is not split but is returned to P intact. We modify our basic algorithm for step one in order to construct a *refinement tree* T whose nodes correspond to the distinct blocks produced by the algorithm. Initially T contains only one node B_λ. As T is constructed, its leaves are the blocks of P. After a proper refinement in which a block B_α is split, we make B_α the parent of each new leaf block $B_{\alpha u}$ and say that u is the *sorting symbol* for $B_{\alpha u}$. An improper refinement to a leaf block B does not change T, since only the associated prefix and not the value of B is modified. The sorting symbol, if any, remains unchanged. Note that only the root block lacks a sorting symbol.

The procedure for constructing a refinement tree T preserves the following two invariants: (i) siblings (i.e., nodes with the same parent) in the tree have distinct sorting symbols; and (ii) if B_α and B_β are siblings with sorting symbols u and v, respectively, and $u<v$, then all strings $x\in B_\alpha$ are lexicographically less than all strings $y\in B_\beta$. Thus, when T is fully constructed, if we arrange each set of siblings within T in ascending order by sorting symbol, then the blocks forming the frontier of T contain the strings of U in lexicographic order.

We represent each block created during step one by a record. For each block B we store with the block its parent $p(B)$ in the refinement tree. If block B belongs to

Downloaded 12/25/12 to 152.3.102.242. Redistribution subject to SIAM license or copyright; see http://www.siam.org/journals/ojsa.php

P (i.e., is either finished or is unfinished and belongs to S), we also store with the block a list of pointers to the strings it contains. For each symbol $i = 0, \cdots, k$, we construct a set L_i of blocks in the refinement tree that have sorting symbol i. When a new block $B_{\alpha u}$ is created during a proper refinement of block B_α, we add $B_{\alpha u}$ to L_u and remove from B_α the list of pointers to the strings it contains. By implementing $L_0, \cdots, L_k$ as an array of lists of blocks, these extra computations increase the time and auxiliary space used in step one by only a constant factor.

In step two, the process of ordering the tree consists of constructing for each block an ordered list of its children. We begin by initializing these lists of children to be empty. Then, for $i = 0, \cdots, k$ and for each block $B \in L_i$, we remove B from L_i and add it to the end of the list of children of block $p(B)$. Thus, when this computation is completed, the children of each block will be in increasing order by sorting symbol.

Since each node of the refinement tree T has either no children or at least two children, T contains at most $2n-1$ nodes. Ordering the tree takes $O(n+k)$ time and auxiliary space. The preorder tree traversal that actually produces the sorted list of strings takes $O(n)$ time. The total resources needed for lexicographic sorting are thus $O(m'+k)$ time and $O(n+k)$ auxiliary space. The space consists of $O(1)$ words per string and two auxiliary arrays of $k+1$ words each. The two auxiliary arrays of buckets can be combined into one array by storing strings belonging to a leaf block of the refinement tree at the front of each bucket and storing blocks internal to the refinement tree at the back. This saves $k+1$ words of space.

We conclude this section with two observations. First, the algorithm can be greatly simplified if the alphabet is of constant size, i.e., $k = O(1)$. In this case we can combine step two with step one, because we can afford to scan empty buckets of the array when forming new blocks during a refinement step. When partitioning a block B during step one, after distributing the strings of B into buckets, we scan all the buckets in increasing order, forming a new block for each nonempty bucket. This allows us to modify the basic algorithm for step one so that both the partition P and the queue of unfinished blocks S are kept in sorted order by the associated prefixes of their blocks. Each block has a pointer to its successor in both S and P. After P is finished, we can output all the strings in lexicographic order by a single linear pass through its blocks.

Alternatively, we can perform a single left-to-right scan of the ordered blocks of P as they are refined. In this method, we need to store the position index locally in each block. Initially, P consists of the single block B_λ with index $I = 0$, which is the first block processed. To process a block B_α we increment its index and test whether it is finished. If it is, then we process the next block to the right or stop if P is finished. If B_α is unfinished, we perform a refinement and replace B_α with the new blocks in sorted order. The index of each new block is I. The leftmost of the new blocks added to P is processed next. This algorithm has the advantage that the strings in a block B can be printed as soon as we determine that B is finished.

Our second observation is that if we can process several symbols of a string at a time then we can obtain a time-space trade-off for the algorithm: if we process c symbols at a time, the running time is $O(m'/c + k^c)$, and the space used is $O(n+k^c)$. In the case of a binary alphabet, this means that we can sort in $O(m'/\log n)$ time and $O(n)$ auxiliary space if we are allowed to operate on $\log n$ bits at a time. An interesting open problem is whether the ideas of Kirkpatrick and Reisch [10] can somehow be applied to the variable-length lexicographic sorting problem we have considered here.

3. Relational coarsest partition. Let E be a binary relation over a finite set U, i.e., a subset of $U \times U$. We abbreviate $(x, y) \in E$ by xEy. For any subset $S \subseteq U$, $E(S) = \{y \mid \exists x \in S \text{ such that } xEy\}$ and $E^{-1}(S) = \{x \mid \exists y \in S \text{ such that } xEy\}$. If $B \subseteq U$ and $S \subseteq U$,

Downloaded 12/25/12 to 152.3.102.242. Redistribution subject to SIAM license or copyright; see http://www.siam.org/journals/ojsa.php

B is *stable* with respect to S if either $B \subseteq E^{-1}(S)$ or $B \cap E^{-1}(S) = \emptyset$. If P is a partition of U, P is *stable* with respect to S if all of the blocks belonging to P are stable with respect to S. P is *stable* if it is stable with respect to each of its own blocks.

The *relational coarsest partition problem* is that of finding, for a given relation E and initial partition P over a set U, the coarsest stable refinement of P, i.e., the partition such that every other stable partition is a refinement of it, so that it has the fewest blocks. (In Theorem 2 we prove that the coarsest stable refinement is unique.) In discussing time bounds for this problem, we let n denote the size of U and m the size of E.

This problem was studied by Kanellakis and Smolka [9] in connection with testing congruence (a kind of equivalence) of finite state processes in the calculus of communicating systems (CCS) [14]. They gave an algorithm requiring $O(mn)$ time and $O(m+n)$ space. For the special case in which every element $x \in U$ has an image set $E(\{x\})$ of size at most a constant c, they gave an algorithm running in $O(c^2 n \log n)$ time. They conjectured the existence of an $O(m \log n)$ time algorithm for the general problem.

We develop an $O(m \log n)$ time algorithm, thereby verifying their conjecture. Our algorithm combines Hopcroft's "process the smaller half" strategy [1], [7] with a new variant of partition refinement. Hopcroft used the "process the smaller half" idea to solve a deceptively similar problem, that of computing the coarsest partition stable with respect to one or more given functions. A solution to this problem can be used to minimize the number of states of a deterministic finite automaton [1], [5], [7]. As Kanellakis and Smolka noted, the relational coarsest partition problem is sufficiently different from the functional problem that a nontrivial generalization of Hopcroft's algorithm is needed to solve it.

Our algorithm uses a primitive refinement operation that generalizes the one used in Hopcroft's algorithm. For any partition Q and subset $S \subseteq U$, let $split(S, Q)$ be the refinement of Q obtained by replacing each block $B \in Q$ such that $B \cap E^{-1}(S) \neq \emptyset$ and $B - E^{-1}(S) \neq \emptyset$ by the two blocks $B' = B \cap E^{-1}(S)$ and $B'' = B - E^{-1}(S)$. We call S a *splitter* of Q if $split(S, Q) \neq Q$. Note that Q is unstable with respect to S if and only if S is a splitter of Q.

Our algorithm exploits several properties of *split* and consequences of stability. Let S and Q be two subsets of U, and let P and R be two partitions of U. The following elementary properties are stated without proof.

(1) Stability is *inherited* under refinement; that is, if R is a refinement of P and P is stable with respect to a set S, then so is R.

(2) Stability is *inherited* under union; that is, a partition that is stable with respect to two sets is also stable with respect to their union.

(3) Function *split* is *monotone* in its second argument; that is, if $S \subseteq U$ and P is a refinement of Q, then $split(S, P)$ is a refinement of $split(S, Q)$.

(4) Function *split* is *commutative*. The coarsest partition of P stable with respect to both S and Q is $split(S, split(Q, P)) = split(Q, split(S, P))$.

We begin by describing a naïve algorithm for the problem. The algorithm maintains a partition Q that is initially P and is refined until it is the coarsest stable refinement. The algorithm consists of repeating the following step until Q is stable with respect to each of its blocks:

Refine. Find a set S that is a union of some of the blocks of Q and is a splitter of Q; replace Q by $split(S, Q)$.

Downloaded 12/25/12 to 152.3.102.242. Redistribution subject to SIAM license or copyright; see http://www.siam.org/journals/ojsa.php

Before proving the correctness of this algorithm, let us make a few observations. The effect of a refinement step is to replace a partition unstable with respect to a set S by a refinement stable with respect to S. Since stability is inherited under refinement, a given set S can be used as a splitter in the algorithm only once. Since stability is inherited under the union of splitters, after sets are used as splitters their unions cannot be used as splitters. In particular, a stable partition is stable with respect to the union of any subset of its blocks.

LEMMA 2. *The algorithm maintains the invariant that any coarsest stable refinement of the initial partition P is also a refinement of the current partition Q.*

Proof. By induction on the number of refinement steps. The lemma is true initially by definition. Suppose it is true before a refinement step that refines partition Q using a splitter S. Let R be any coarsest stable refinement of P. Since S is a union of blocks of Q and R is a refinement of Q by the induction hypothesis, S is a union of blocks of R. Hence, R is stable with respect to S. Since *split* is monotone, $R = split(S, R)$ is a refinement of $split(S, Q)$. □

THEOREM 2. *The refinement algorithm is correct and terminates after at most $n-1$ refinement steps, having computed the unique coarsest stable partition.*

Proof. Since the number of blocks in Q must be between one and n, and since this number increases after each refinement step, the algorithm terminates after at most $n-1$ refinement steps. Once no more refinement steps are possible, Q is stable, and by Lemma 2 any stable refinement is a refinement of Q. It follows that Q is the unique coarsest stable refinement. □

The refinement algorithm is more general than is necessary to solve the problem: there is no need to use unions of blocks of the current partition Q as splitters; restricting the splitters to blocks of Q will do. However, the freedom to split using unions of blocks is one of the crucial ideas needed in developing a fast version of algorithm. We shall see how this freedom is used later in the section.

In an efficient implementation of the algorithm, it is useful to reduce the problem instance to one in which $|E(\{x\})| \geqq 1$ for all $x \in U$. To do this we preprocess the partition P by splitting each block $B \in P$ into $B' = B \cap E^{-1}(U)$ and $B'' = B - E^{-1}(U)$. The blocks B'' will never be split by the refinement algorithm; thus we can run the refinement algorithm on the partition P' consisting of the set of blocks B'. P' is a partition of the set $U' = E^{-1}(U)$, of size at most m. The coarsest stable refinement of P' together with the blocks B'' is the coarsest stable refinement of P. The preprocessing and postprocessing take $O(m+n)$ time if we have available for each element $x \in U$ its preimage set $E^{-1}(\{x\})$. Henceforth we shall assume $|E(\{x\})| \geqq 1$ for all $x \in U$. This implies $m \geqq n$.

We can implement the refinement algorithm to run in $O(nm)$ time by storing for each element $x \in U$ its preimage set $E^{-1}(\{x\})$. Finding a block of Q that is a splitter and performing the appropriate splitting takes $O(m)$ time. (Obtaining this bound is an easy exercise in list processing.) An $O(mn)$ time bound for the entire algorithm follows from the bound on the number of splitting steps.

To obtain a faster version of the algorithm, we need a good way to find splitters. In addition to the current partition Q, we maintain another partition X such that Q is a refinement of X and Q is stable with respect to every block of X. Initially $Q = P$ and X is the partition containing U as its single block. The improved algorithm consists of repeating the following step until $Q = X$:

> *Refine.* Find a block $S \in X$ that is not a block of Q. Find a block $B \in Q$ such that $B \subseteq S$ and $|B| \leqq |S|/2$. Replace S within X by the two sets B and $S - B$; replace Q by $split(S - B, split(B, Q))$.

Downloaded 12/25/12 to 152.3.102.242. Redistribution subject to SIAM license or copyright; see http://www.siam.org/journals/ojsa.php

The correctness of this improved algorithm follows from the correctness of the original algorithm and from the two ways given previously in which a partition can inherit stability with respect to a set.

Before discussing this algorithm in general, let us consider the special case in which E is a function, i.e., $|E(\{x\})| = 1$ for all $x \in U$. In this case, if Q is a partition stable with respect to a set S that is a union of some of the blocks of Q, and $B \subseteq S$ is a block of Q, then *split*(B, Q) is stable with respect to $S - B$, since if B_1 is a block of *split*(B, Q), $B_1 \subseteq E^{-1}(B)$ implies $B_1 \cap E^{-1}(S-B) = \emptyset$, and $B_1 \subseteq E^{-1}(S) - E^{-1}(B)$ implies $B_1 \subseteq E^{-1}(S-B)$. This means that in each refinement step it suffices to replace Q by *split*(B, Q), since *split*(B, Q) = *split*$(S-B,$ *split*$(B, Q))$. This is the idea underlying Hopcroft's "process the smaller half" algorithm for the functional coarsest partition problem; the refining set B is at most half the size of the stable set S containing it.

In the more general relational coarsest partition problem, stability with respect to both S and B does *not* imply stability with respect to $S - B$, and Hopcroft's algorithm cannot be used. Nevertheless, we are still able to exploit Hopcroft's "process the smaller half" idea, by refining with respect to both B and $S - B$ using a method that explicitly scans only B.

Consider a general step of the improved refinement algorithm.

LEMMA 3. *Suppose that partition Q is stable with respect to a set S that is a union of some of the blocks of Q. Suppose also that partition Q is refined first with respect to a block $B \subseteq S$ and then with respect to $S - B$. Then the following conditions hold*:

(1) *Refining Q with respect to B splits a block $D \in Q$ into two blocks $D_1 = D \cap E^{-1}(B)$ and $D_2 = D - D_1$ iff $D \cap E^{-1}(B) \neq \emptyset$ and $D - E^{-1}(B) \neq \emptyset$.*

(2) *Refining split(B, Q) with respect to $S - B$ splits D_1 into two blocks $D_{11} = D_1 \cap E^{-1}(S-B)$ and $D_{12} = D_1 - D_{11}$ iff $D_1 \cap E^{-1}(S-B) \neq \emptyset$ and $D_1 - E^{-1}(S-B) \neq \emptyset$.*

(3) *Refining split(B, Q) with respect to $S - B$ does not split D_2.*

(4) $D_{12} = D_1 \cap (E^{-1}(B) - E^{-1}(S-B))$.

Proof. (1, 2) These follow from the definition of *split*.

(3) By (1), if D is split, then $D \cap E^{-1}(B) \neq \emptyset$. Since D is stable with respect to S, and since $B \subseteq S$, then $D_2 \subseteq D \subseteq E^{-1}(S)$. Since by (1) $D_2 \cap E^{-1}(B) = \emptyset$, it follows that $D_2 \subseteq E^{-1}(S-B)$.

(4) This follows from the fact that $D_1 \subseteq E^{-1}(B)$ and $D_{12} = D_1 - E^{-1}(S-B)$. □

Performing the three-way splitting of a block D into D_{11}, D_{12}, and D_2 as described in Lemma 3 is the hard part of the algorithm. Identity (4) of Lemma 3 is the crucial observation that we shall use in our implementation.

A given element $x \in U$ is in at most $\log_2 n + 1$ different blocks B used as refining sets, since each successive such set is at most half the size of the previous one. We shall describe an implementation of the algorithm in which a refinement step with respect to a block B takes $O(|B| + \Sigma_{y \in B} |E^{-1}(\{y\})|)$ time. From this an $O(m \log n)$ overall time bound on the algorithm follows by summing over all blocks B used for refinement and over all elements in such blocks.

Efficient implementation of the algorithm requires several data structures. We represent each element x by a record, which we shall not distinguish from the element itself. We represent each pair x, y such that xEy by a record that we shall call an *edge* and denote by xEy. We represent each block in partition Q and each block in partition X by a record that we shall not distinguish from the block itself. A block S of X is *simple* if it contains only a single block of Q (equal to S but indicated by its own record) and *compound* if it contains two or more blocks of Q. The various records are linked together in the following ways. Each edge xEy points to element x. Each element y points to a list of the edges xEy. This allows scanning of the set $E^{-1}(\{y\})$ in time

Downloaded 12/25/12 to 152.3.102.242. Redistribution subject to SIAM license or copyright; see http://www.siam.org/journals/ojsa.php

proportional to its size. Each block of Q has an associated integer giving its size and points to a doubly linked list of the elements in it. (The double linking allows deletion in $O(1)$ time.) Each element points to the block of Q containing it. Each block of X points to a doubly linked list of the blocks of Q contained in it. Each block of Q points to the block of X containing it. We also maintain the set C of compound blocks of X. Initially C contains the single block U, which is the union of the blocks of P. (If P contains only one block, P itself is the coarsest stable refinement, and no computation is necessary.)

To facilitate three-way splitting we need one more collection of records. For each block S of X and each element $x \in E^{-1}(S)$, we maintain a record containing the integer $count(x, S) = |S \cap E(\{x\})|$. We shall not distinguish this record from the count itself. Each edge xEy such that $y \in S$ contains a pointer to $count(x, S)$. Initially there is one count per vertex, $count(x, U) = |E(\{x\})|$; each edge xEy contains a pointer to $count(x, U)$.

The space needed for all the data structures is $O(m)$, as is the initialization time. (Recall that $m \geqq n$.) The refinement algorithm consists of repeating refinement steps until $C = \varnothing$. A refinement step is performed as follows:

Step 1 (select a refining block). Remove some block S from C. (Block S is a compound block of X.) Examine the first two blocks in the list of blocks of Q contained in S. Let B be the smaller. (Break a tie arbitrarily.)

Step 2 (update X). Remove B from S and create a new (simple) block S' of X containing B as its only block of Q. If S is still compound, put S back into C.

Step 3 (compute $E^{-1}(B)$). Copy the elements of B into a temporary set B'. (This facilitates splitting B with respect to itself during the refinement.) Compute $E^{-1}(B)$ by scanning the edges xEy such that $y \in B$ and adding each element x in such an edge to $E^{-1}(B)$, if it has not already been added. Duplicates are suppressed by marking elements as they are encountered and linking them together for later unmarking. During the same scan, compute $count(x, B) = |\{y \in B \mid xEy\}|$, store this count in a new *count* record, and make x point to it.

Step 4 (refine Q with respect to B). For each block D of Q containing some element of $E^{-1}(B)$, split D into $D_1 = D \cap E^{-1}(B)$ and $D_2 = D - D_1$. Do this by scanning the elements of $E^{-1}(B)$. To process an element $x \in E^{-1}(B)$, determine the block D of Q containing it, and create an associated block D' if one does not already exist. Move x from D to D'.

During the scanning, construct a list of those blocks D that are split. After the scanning, process the list of split blocks. For each such block D with associated block D', mark D' as no longer being associated with D (so that it will be correctly processed in subsequent iterations of Step 4); eliminate the record for D if D is now empty; and, if D is nonempty and the block of X containing D and D' has been made compound by the split, add this block to C.

Step 5 (compute $E^{-1}(B) - E^{-1}(S - B)$). Scan the edges xEy such that $y \in B'$. To process an edge xEy, determine $count(x, B)$ (to which x points) and $count(x, S)$ (to which xEy points). If $count(x, B) = count(x, S)$, add x to $E^{-1}(B) - E^{-1}(S - B)$ if it has not been added already.

Step 6 (refine Q with respect to $S - B$). Proceed exactly as in Step 4 but scan $E^{-1}(B) - E^{-1}(S - B)$ (computed in Step 5) instead of $E^{-1}(B)$.

Step 7 (update counts). Scan the edges xEy such that $y \in B'$. To process an edge xEy, decrement $count(x, S)$ (to which xEy points). If this count becomes zero, delete the *count* record, and make xEy point to $count(x, B)$ (to which x points). After scanning all the appropriate edges, discard B'.

Downloaded 12/25/12 to 152.3.102.242. Redistribution subject to SIAM license or copyright; see http://www.siam.org/journals/ojsa.php

The correctness of this implementation follows in a straightforward way from our discussion above of three-way splitting. The time spent in a refinement step is $O(1)$ per edge scanned plus $O(1)$ per vertex of B, for a total of $O(|B|+\Sigma_{y\in B}|E^{-1}(\{y\})|)$ time. An $O(m \log n)$ time bound for the entire algorithm follows as discussed above. It is possible to improve the efficiency of the algorithm by a constant factor by combining various steps; we have kept the steps separate for clarity.

We close this section with some remarks about another partition refinement problem. Let U be a finite set, E a binary relation over U, and P a partition over U. Suppose $S \subseteq U$. Partition P is *size-stable* (abbreviated *s-stable*) with respect to S if for all blocks $B \in P$, $|E(\{x\}) \cap S| = |E(\{y\}) \cap S|$ for all elements $x, y \in B$. Partition P is *s-stable* if it is s-stable with respect to each of its blocks. The problem is to find the coarsest s-stable refinement of P. This problem is central to certain heuristics for graph isomorphism [16]. Observe that s-stability implies stability but not vice versa. An example of a stable partition that is not s-stable is a single block $B = \{1, 2\}$ such that $E(\{1\}) = \{1, 2\}$ and $E(\{2\}) = \{1\}$.

A partition refinement algorithm like the one we have described can be used to find the coarsest s-stable refinement in $O(m \log n)$ time. The problem is both simpler and more complicated than the stable coarsest refinement problem: simpler in that Hopcroft's approach applies directly, but more complicated in that when a block splits, it splits into an arbitrarily large number of new blocks. We shall sketch the algorithm. An algorithm with the same running time was proposed earlier by Cardon and Crochemore [3].

For a partition Q and a set S, let *s-split*(S, Q) be the partition formed from Q by splitting each block $B \in Q$ into blocks $B_0, B_1, \cdots, B_k$, where $B_i = \{x \in B: |E(\{x\}) \cap S| = i\}$. (Any of the B_i's can be empty; for a proper split to take place, at least two must be nonempty.) The algorithm maintains a queue L of possible splitters, initially containing every block of P. The algorithm consists of initializing $Q = P$ and applying the following step until L is empty, at which time Q is the coarsest s-stable refinement:

> *S-Refine.* Remove from L its first set S. Replace Q by *s-split*(S, Q). Whenever a block $B \in Q$ splits into two or more nonempty blocks, add all but the largest to the back of L.

The correctness of this algorithm follows from the fact that if Q is s-stable with respect to a set S and S is split into smaller blocks $S_1, S_2, \cdots, S_k$, refining with respect to all but one of the sets S_i guarantees s-stability with respect to the last one. The implementation of the algorithm is analogous to that of the stable coarsest refinement algorithm. The data structures are much simplified because when refining with respect to a block $B \subseteq S$ we do not need to refine with respect to $S - B$. This removes the need for the auxiliary partition X and the count records. On the other hand, the splitting of a block becomes more complicated, because it can split into many pieces. Implementing the *s-refine* step to take $O(|S|+\Sigma_{y\in S}|E^{-1}(\{y\})|)$ time, where S is the set removed from L, is an interesting exercise in list processing that we leave to the reader. An $O(m \log n)$ time bound for the entire algorithm follows from the observation that a given element y can be in only $\log_2 n + 1$ sets removed from L.

Although our s-refinement algorithm resembles Cardon and Crochemore's algorithm, Hopcroft's algorithm for the functional coarsest partition problem, and Kanellakis and Smolka's algorithm for the bounded fanout case of the relational coarsest partition problem, our method is somewhat simpler because, by refining with respect to old blocks rather than current ones, we are able to avoid some updating.

Downloaded 12/25/12 to 152.3.102.242. Redistribution subject to SIAM license or copyright; see http://www.siam.org/journals/ojsa.php

(The other algorithms must update L each time a block in L is split.) Our approach thus gives a slightly simplified algorithm for the functional problem. It is also easily generalized to handle several relations, where we require stability or s-stability with respect to each one. See, e.g., Cardon and Crochemore's paper.

4. Double lexical ordering. The third and last problem we consider is one of matrix reordering. Let M be an n by k nonnegative matrix containing m nonzeros, called its *entries.* The *double lexical ordering problem* is that of independently permuting the rows and columns of M so that both the row vectors and the column vectors are in nonincreasing lexicographic order. Lexicographic order is defined as in § 2, reading each row from left to right and each column from top to bottom.

Lubiw [11] defined the double lexical ordering problem and found a number of applications, including the efficient recognition of totally balanced matrices and strongly chordal graphs. In her version of the problem, rows are read from right to left and columns from bottom to top (i.e., vectors are read from last to first component), and the required rows and columns are in nondecreasing lexicographic order. To convert our results to her setting, it suffices to exchange "left" with "right," "top" with "bottom," and "increasing" with "decreasing" throughout this section.

Lubiw developed a double lexical ordering algorithm taking $O(m \log (n+k)^2 + n+k)$ time and $O(m+n+k)$ space. We develop an implementation of her algorithm running in $O(m \log (n+k)+k)$ time and $O(m+n+k)$ space. Since all Lubiw's applications are for the special case of zero-one matrices, we give a simplified algorithm with the same resource bounds for this case.

As input for the ordering problem, we assume a matrix representation consisting of a list of triples, each composed of an entry, its row, and its column. From this input we can construct lists of the entries in each row using a radix sort by row. We can construct similar lists for the columns. This takes $O(m+n)$ time. We assume that $m \geqq n \geqq k$. We can validate the assumption $n \geqq k$ by transposing the matrix if necessary, which takes $O(m)$ time. We can validate the assumption $m \geqq n$ by finding all the rows with no entries and ordering them last, which takes $O(m+n)$ time. Ordering the submatrix consisting of the remaining rows and all the columns suffices to order the entire matrix.

We begin by describing a variant of Lubiw's ordering algorithm. We need some terminology. An *ordered partition* of a finite set S is a partition of S whose blocks are totally ordered. We denote an ordered partition by a list of the blocks in order. A total ordering $<$ of S is *consistent* with an ordered partition P of S if whenever two blocks $S_1 \in P$ and $S_2 \in P$ satisfy $S_1 < S_2$, then every $e_1 \in S_1$ and $e_2 \in S_2$ satisfy $e_1 < e_2$. If P and Q are ordered partitions of S, Q is a *refinement* of P if every block of Q is contained in a block of P and every total ordering of S consistent with Q is consistent with P.

Let M be a nonnegative matrix with n rows and k columns. $M(i, j)$ denotes the entry in the ith row and jth column. An ordered partition $R_1, R_2, \cdots, R_p$ of the rows of M and an ordered partition $C_1, C_2, \cdots, C_q$ of the columns of M divide M into $p \times q$ *matrix blocks* $M(R_i, C_j)$. Block $M(R_i, C_j)$ is the submatrix defined by the rows in row block R_i and the columns in column block C_j. Let $B = M(R_i, C_j)$ be a matrix block. If $M(r, c) = M(r', c')$ for all $r, r' \in R_i$, $c, c' \in C_j$, then block B is *constant.* We denote by $\max(B)$ the maximum value in matrix block B. A *row slice* of B is the intersection of a row $r \in R$ with B, i.e., the submatrix $M(r, C_j)$. A *splitting row* of B is a row $r \in R_i$ such that the row slice $M(r, C_j)$ is nonconstant and contains at least one entry equal to $\max(B)$. If $B' = M(R_{i'}, C_{j'})$ is another matrix block, then B' is *above* B if $i' < i$ and $j' = j$; B' is *left* of B if $i' = i$ and $j' < j$.

Downloaded 12/25/12 to 152.3.102.242. Redistribution subject to SIAM license or copyright; see http://www.siam.org/journals/ojsa.php

The ordering algorithm maintains an ordered partition of the rows, initially containing one block of all the rows, and an ordered partition of the columns, initially containing one block of all the columns. The algorithm refines the row and column partitions until every matrix block defined by the partitions is constant, at which time any total ordering of the rows and columns consistent with the final partitions is a double lexical ordering. The algorithm consists of repeating the following step until every matrix block is constant:

Refine. Let $B = M(R, C)$ be any nonconstant matrix block such that all blocks left of B and all blocks above B are constant. If B has a splitting row r, replace C in the ordered column partition by the ordered pair $C' = \{c \in C \mid M(r, c) = \max(B)\}$, $C'' = \{c \in C \mid M(r, c) < \max(B)\}$ (so that C' comes before C'' in the column ordered partition). If B has no splitting row, replace R in the ordered row partition by the ordered pair $R' = \{r \in R \mid M(r, c) = \max(B) \forall c \in C\}$, $R'' = \{r \in R \mid \exists c \in C$ such that $M(r, c) < \max(B)\}$ (so that R' comes before R'' in the ordered row partition).

Lubiw's proof of correctness for her algorithm is valid without change for our algorithm. However, our algorithm differs in two ways from Lubiw's algorithm as presented in [11]. First, in each refinement step, her implementation selects a matrix block $B = M(R, C)$ that has C minimum among all nonconstant blocks, and among blocks within C has R minimum. Selected matrix blocks are totally ordered from top to bottom within each column block and from left to right among column blocks. In contrast, our algorithm selects blocks according to the partial order defined by "left" and "above" (i.e., if block B is left of or above block B', then B precedes B'). This additional flexibility in our algorithm contributes to the improved time bound. Second, her algorithm arbitrarily chooses either a splitting row or column (defined analogously to splitting row) when both exist, whereas our algorithm chooses a splitting column only when there are no splitting rows. However, this is not an essential difference, because whenever there is no splitting row (respectively splitting column), the refinement made by her algorithm is the same as ours. For example, if there is no splitting row, her algorithm chooses any splitting column c and replaces the row block R by the ordered pair $R' = \{r \in R \mid M(r, c) = \max(B)\}$, $R'' = \{r \in R \mid M(r, c) < \max(B)\}$. If there is no splitting row, every row slice of $M(R, C)$ containing at least one value equal to $\max(B)$ must have all values equal to $\max(B)$. Thus, in this case the refinement made by her algorithm is independent of the choice of splitting column and is the same as the refinement made by our algorithm.

Both Lubiw's efficient implementation and ours are based on Hopcroft's "process the smaller half" idea, which we have already used in § 3. Since the special case of zero-one matrices is much simpler than the general case and since it is what occurs in all of Lubiw's applications, we discuss it first. In the zero-one case, the refinement step becomes the following:

Refine. Let $B = M(R, C)$ be any nonconstant matrix block such that all blocks left of B and all blocks above B are constant. If there is a row r such that $M(r, C)$ is nonconstant, replace column block C by the ordered pair $C' = \{c \in C \mid M(r, c) = 1\}$, $C'' = \{c \in C \mid M(r, c) = 0\}$. Otherwise, replace row block R by the ordered pair $R' = \{r \in R \mid M(r, c) = 1 \forall c \in C\}$, $R'' = \{r \in R \mid M(r, c) = 0 \forall c \in C\}$. (In the latter case both $M(R', C)$ and $M(R'', C)$ are constant.)

To implement this algorithm, we need to use a rather elaborate data structure, since we must keep track of a row partition, a column partition, and the matrix blocks

Downloaded 12/25/12 to 152.3.102.242. Redistribution subject to SIAM license or copyright; see http://www.siam.org/journals/ojsa.php

they define. We also keep track of row slices. We need only keep track of row slices and matrix blocks containing at least one entry; thus in what follows the terms "row slice" and "matrix block" refer to those with at least one entry. Since we are in the zero-one case, every entry is a one. The data structure contains a record for each entry, row, column, row block, column block, row slice, and matrix block. In our discussion we do not distinguish between an object and the corresponding record. Each entry has pointers to the row, column, and row slice containing it. Each row points to the row block containing it and to a list of its entries. Each column points to the column block containing it and to a list of its entries. Each row block has its size (number of rows) and points to a doubly linked list of its rows. Each column block has its size (number of columns) and points to a doubly linked list of its columns. Each row slice has its size (number of entries) and points to a doubly linked list of its entries and to the matrix block containing it. Each matrix block points to a list of its row slices. In addition, we maintain three lists: one of the row blocks in order, one of the column blocks in order, and one of the matrix blocks, in an order consistent with the partial orders "left" and "above". (If block B is left of or above B', B occurs before B' in the list. See Fig. 1.) Initializing the data structure takes $O(m)$ time.

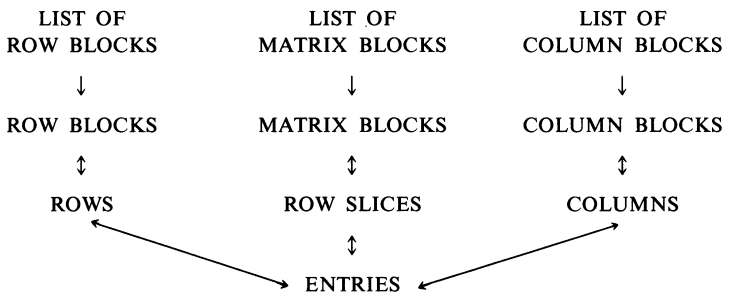

FIG. 1. *Schematic representation of the data structure for the double lexical ordering algorithm.*

To perform a refinement step, we remove the first block $B = M(R, C)$ from the list of matrix blocks. We determine the row block R and column block C (by finding a row slice of B, finding an entry in the row slice, finding the row and column of the entry, and finding the row block of the row and the column block of the column). We scan the row slices of B one at a time until finding one, say s, with fewer than $|C|$ entries. This takes $O(1)$ time per slice. If we find such an s, we scan its entries and determine the set of columns C' containing these entries. We split C into C' and $C'' = C - C'$ and update the data structures accordingly. Then we add $M(R, C')$ and $M(R, C'')$ to the front of the list of matrix blocks (so that $M(R, C')$ is the first block in the list). If there is no such s, we find the set of rows R' containing the row slices of B, split R into R' and $R'' = R - R'$, and update the data structure accordingly.

In updating the data structure we use the "relabel the smaller" half idea. When updating to reflect a split of a column block C into C' and C'', we process whichever of C' and C'' contains fewer columns, spending time proportional to its number of entries. Similarly, when updating to reflect a split of a row block R into R' and R'', we process whichever of R' and R'' contains fewer rows, spending time proportional to its number of entries. Thus, the cost of splitting a row or column can be regarded as the sum of unit time charges to each entry e in the smaller half of the split block. Each two consecutive times that an entry e is charged because of a split in column

Downloaded 12/25/12 to 152.3.102.242. Redistribution subject to SIAM license or copyright; see http://www.siam.org/journals/ojsa.php

(respectively row) blocks, the size of the new block containing e is reduced by at least a factor of two. Thus, any entry e can be charged no more than $O(\log n)$ times, and the total running time of the algorithm is $O(m \log n)$. The space needed is $O(m)$.

Let us discuss what happens when a column block is split. Such a split causes matrix blocks and their row slices to be split also. Finding a row slice s in B with fewer than $|C|$ entries takes $O(1)$ time plus $O(1)$ time for each row slice in B with $|C|$ entries (in the worst case). A row slice in B with $|C|$ entries has at least one entry in both of the new column blocks; thus the cost of finding s is $O(1)$ per entry in the new block with fewer columns. Let s contain $|s|$ entries. Once s is found, we can traverse the list of entries in s to find the columns in the new block C' in $O(|s|)$ time. Once C is split into C' and C'', all the entries in s are in a constant row slice. It follows that a given entry is in only one row slice s used for splitting; hence the total number of columns of new blocks C' (which bounds the total cost of traversing row slices s) formed during the running of the algorithm is $O(m)$. Removing the columns of C' from C takes $O(|C'|)$ time. Suppose that C' has fewer columns than C''. (The other case is similar.) We scan the entries in C', removing them from their current row slices to form new row slices. When the first row slice in a matrix block is split, we form a new matrix block as well, to hold the new row slices split from the block. The time for constructing new row slices and matrix blocks is proportional to the number of entries in C'. When an old row slice or matrix block becomes empty (because all its entries have been deleted), we delete it. We insert a new matrix block into the list of matrix blocks just in front of the block from which it is split. The remaining details of the updating are straightforward.

The updating when a row block is split is similar except that row slices are not split but only moved from old matrix blocks to new ones. The time to split R into R' and R'' is proportional to the number of row slices in B plus the number of entries in whichever of the blocks R' and R'' is processed. It follows from the discussion above that the total running time of the double lexical ordering algorithm is $O(m \log n)$.

We turn now to the general case, in which the matrix entries are arbitrary positive real numbers. Our method is the same as in the zero-one case, with two exceptions. First, we change the data structure slightly, to allow for entries having different values. Second, and more fundamental, we need a new algorithmic tool, to solve a problem in data structures we call the *unmerging problem*: given a list L containing p elements and a set of pointers to q elements in L, partition L into two lists: L', containing the elements indicated by the pointers, and L'', containing the remaining elements. The list order must be preserved, i.e., if element x precedes element y in either L' or L'', then x precedes y in L. This ordering requirement is what makes the problem hard; without it, unmerging takes $O(q)$ time using doubly linked lists, a fact we have used repeatedly in this paper. In the Appendix, we describe how to solve the unmerging problem in $O(q(1+\log(p/q)))$ time by representing the lists as balanced search trees. This efficient solution to the unmerging problem allows us to beat Lubiw's double lexical ordering algorithm by a logarithmic factor even in the general case.

One new concept is needed in order for the zero-one algorithm to handle the general case. A *row slice overlay* is a maximal set of entries in a row slice all having the same value. We change the data structure as follows. Instead of representing row slices, we represent row slice overlays. We represent each overlay by a record, which in our discussion will not be distinguished from the overlay itself. Each overlay has a pointer to the matrix block containing it, a pointer to a doubly linked list of its entries, and an integer giving the number of its entries. Each entry points to the overlay containing it. Each matrix block points to a list of the overlays it contains, in nonincreas-

Downloaded 12/25/12 to 152.3.102.242. Redistribution subject to SIAM license or copyright; see http://www.siam.org/journals/ojsa.php

ing order by the value of their entries. Each list of overlays is represented by a balanced search tree of a kind that supports efficient unmerging and also allows an insertion next to a specified overlay or a deletion to be performed in $O(1)$ amortized time.[1] Red-black trees [6], [17] or weak B-trees [8], [12] will do. Initializing the entire data structure takes $O(m \log n)$ time since we need to sort the entries in each row to determine the overlays and to order the lists of overlays.

We make the following changes in the implementation of a refinement step. Let $B = M(R, C)$ be the first matrix block on the list of matrix blocks. We examine overlays on the list of B's overlays until finding one containing fewer than $|C|$ entries or until the next candidate overlay has entries smaller than $\max(B)$. In the former case, we split C into C', containing the entries in the selected overlay, and $C'' = C - C'$; then we add $M(R, C')$ and $M(R, C'')$ to the front of the list of matrix blocks (with $M(R, C')$ first). In the latter case, we split R into R', containing the entries of B equal to $\max(B)$, and $R'' = R - R'$; then we add $M(R'', C)$ to the front of the list of matrix blocks ($M(R', C)$ is a constant block).

The updating of the data structure proceeds as in the zero-one case except for the updating of the overlay lists. Consider a split of a column block C into C' and C''. Without loss of generality, suppose the entries in C' are processed to perform the updating. Let B be a block containing entries in C'. As entries in C' are examined, they are removed from their old overlays and inserted into new overlays, each of which is inserted in B's list of overlays next to the overlay from which it split. An unmerging operation is then performed to split the overlay list into two lists, one for each of the blocks into which B splits. The updating when a row block splits is similar, except that overlays do not split but are merely moved from one list to another by unmerging operations.

Excluding the unmerging operations, the total time for updating the data structure over the entire algorithm is $O(m \log n)$. We shall derive the same bound for the total time of the unmerging operations. Consider an unmerging operation that splits an overlay list L for a matrix block B into two lists, L' and L'', respectively, for the blocks B' and B'' into which B splits. Let B' be the block containing the entries processed to perform the splitting. Let p be the number of entries in B and q the number in B'. The time for the unmerging operation is $O(|L'|(1+\log((|L'|+|L''|)/|L'|))) = O(q(1+\log(p/q)))$, since $p \geqq |L'|+|L''|$, $q \geqq |L'|$, and $x(1+\log(c/x))$ is an increasing function of x.

We charge $O(q)$ time of the unmerging operation to the processing of the entries in B'. The total of all such charges over the entire algorithm is $O(m \log n)$, since each entry is charged $O(\log n)$ times. It remains to account for $O(q \log(p/q))$ time. We call this the *excess time* associated with the unmerge operation. Let $T(x)$ be the maximum total excess time spent doing unmerging operations associated with a block B containing at most x entries and all smaller blocks formed later contained in B. Then $T(x)$ obeys the recurrence $T(1) = 0$, $T(x) \leqq \max_{y<x}\{T(y) + T(x-y) + O(y \log(x/y))\}$ for $x \geqq 2$. The bound $T(x) = O(x \log x)$ follows easily by induction. Hence the total excess unmerging time over the entire algorithm is $O(m \log m) = O(m \log n)$. We conclude that the algorithm runs in $O(m \log n)$ time total. An $O(m)$ space bound is obvious.

By way of conclusion, we note that our algorithm not only is asymptotically faster than Lubiw's algorithm but it uses less space by a constant factor, since we have

[1] By *amortized time* we mean the time per operation averaged over a worst-case sequence of operations. See Tarjan's survey paper [18] for a full discussion of this concept.

Downloaded 12/25/12 to 152.3.102.242. Redistribution subject to SIAM license or copyright; see http://www.siam.org/journals/ojsa.php

eliminated several unnecessary parts of her data structure: row slices (which she uses in the general case), column slices, column slice overlays, and matrix block overlays. Whether the data structure can be further simplified is an open question. Our algorithm improves the speed of Lubiw's algorithms for recognizing totally balanced matrices and strongly chordal graphs by a logarithmic factor.

5. Remarks. We have presented efficient partition refinement algorithms for three rather different problems. These three algorithms together with our earlier work [15] are part of a larger investigation of why some partition refinement algorithms run in linear time while others do not. We hope that our ideas on partition refinement and our solution to the unmerging problem (see the Appendix) will contribute to the more efficient solution of other problems, such as perhaps the congruence closure problem [4].

Appendix. An efficient unmerging algorithm. Let L be a list containing p elements and let S be a set of pointers to q elements in L. The unmerging problem is to divide L into two lists: L', containing the elements of L indicated by the pointers in S; and L'', containing the remaining elements. The order of elements in L' and L'' must be the same as in L, i.e., if x precedes y in either L' or L'', then x precedes y in L.

To solve the unmerging problem, we represent the input list L by a balanced search tree T. For definiteness, we shall assume that T is a red-black tree [6], [17] or a weak B-tree [8], [12], although many other kinds of balanced trees will do as well. We assume some familiarity with balanced search trees. There are three steps to the algorithm.

Step 1. Color all nodes indicated by pointers yellow and all their ancestors green. To do this, process the nodes indicated by the pointers, one at a time. To process a node x, color x yellow, then the parent of x green, then the grandparent of x green, and so on, until reaching an already-colored node.

Once Step 1 is completed, the colored nodes define a subtree T' of T containing $O(q(1+\log(p/q)))$ nodes [2], [8]. The time required by Step 1 is proportional to the number of nodes in T', i.e., $O(q(1+\log(p/q)))$ time.

Step 2. Perform a symmetric-order traversal of T', uncoloring each colored node as it is reached and adding it to a list if it is yellow.

The list constructed during Step 2 is L'. A red-black tree or weak B-tree representing L' can be constructed in $O(q)$ time (by successive insertions of the items in order [2], [8], [12], [17]).

Step 3. Delete from T the elements indicated by the pointers in S one at a time.

The amortized time per deletion in Step 3 is $O(1)$ [8], [12], [17]. The total worst-case deletion time is $O(q(1+\log(p/q)))$ [2], [8]. Hence the total time for unmerging is $O(q(1+\log(p/q)))$, a bound that is valid both in the worst case and in the amortized case.

Acknowledgment. We thank Anna Lubiw for her perceptive comments concerning the double lexical ordering algorithm.

REFERENCES

[1] A. V. AHO, J. E. HOPCROFT AND J. D. ULLMAN, *Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, MA, 1974.

[2] M. R. BROWN AND R. E. TARJAN, *Design and analysis of a data structure for representing sorted lists*, this Journal, 9 (1980), pp. 594–614.

[3] A. CARDON AND M. CROCHEMORE, *Partitioning a graph in* $O(|A|\log_2|V|)$, Theoret. Comput. Sci., 19 (1982), pp. 85–98.

[4] P. J. DOWNEY, R. SETHI AND R. E. TARJAN, *Variations on the common subexpression problem*, J. Assoc. Comput. Mach., 27 (1980), pp. 758–771.
[5] D. GRIES, *Describing an algorithm by Hopcroft*, Acta Inform., 2 (1973), pp. 97–109.
[6] L. J. GUIBAS AND R. SEDGEWICK, *A dichromatic framework for balanced trees*, Proc. 19th Annual IEEE Symposium on the Foundations of Computer Science, 1978, pp. 18–21.
[7] J. HOPCROFT, *An n* log *n algorithm for minimizing states in a finite automation*, in Theory of Machines and Computations, Z. Kohavi and A. Paz, eds., Academic Press, New York, 1971, pp. 189–196.
[8] S. HUDDLESTON AND K. MEHLHORN, *A new data structure for representing sorted lists*, Acta Inform., 17 (1982), pp. 157–184.
[9] P. C. KANELLAKIS AND S. A. SMOLKA, *CCS expressions, finite state processes, and three problems of equivalence*, Proc. ACM Symposium on Principles of Distributed Computing, 1983, pp. 228–240.
[10] D. KIRKPATRICK AND S. REISCH, *Upper bounds for sorting integers on random access machines*, Theoret. Comput. Sci., 28 (1984), pp. 263–276.
[11] A. LUBIW, *Doubly lexical orderings of matrices*, Proc. 17th Annual ACM Symposium on the Theory of Computing, 1985, pp. 396–404; this Journal, 16 (1987), pp. 854–879.
[12] D. MAIER AND S. C. SALVETER, *Hysterical B-trees*, Inform. Process. Lett., 12 (1981), pp. 199–202.
[13] K. MEHLHORN, *Data Structures and Algorithms* 1: *Sorting and Searching*, Springer-Verlag, Berlin, 1984.
[14] R. MILNER, *A Calculus of Communicating Systems*, Lecture Notes in Computer Science 92, Springer-Verlag, Berlin, 1980.
[15] R. PAIGE, R. E. TARJAN AND R. BONIC, *A linear time solution to the single function coarsest partition problem*, Theoret. Comput. Sci., 40 (1985), pp. 67–84.
[16] R. C. READ AND D. G. CORNEIL, *The graph isomorphism disease*, J. Graph Theory, 1 (1977), pp. 339–363.
[17] R. E. TARJAN, *Data Structures and Network Algorithms*, CBMS-USF Regional Conference Series in Applied Mathematics 44, Society for Industrial and Applied Mathematics, Philadelphia, PA, 1983.
[18] ———, *Amortized computational complexity*, SIAM J. Algebraic Discrete Methods, 6 (1985), pp. 306–318.

Downloaded 12/25/12 to 152.3.102.242. Redistribution subject to SIAM license or copyright; see http://www.siam.org/journals/ojsa.php