THREE PARTITION REFINEMENT ALGORITHMS

Robert Paige

Robert E. Tarjan

Three Partition Refinement Algorithms

Robert Paige [1,2]

Computer Science Department
Rutgers University
New Brunswick, NJ 08903

Robert E. Tarjan [2]

Computer Science Department
Princeton University
Princeton, NJ 08544
and
AT&T Bell Laboratories
Murray Hill, New Jersey 07974

January, 1986

ABSTRACT

We present improved partition refinement algorithms for three problems: lexicographic sorting, relational coarsest partition, and double lexical ordering.

Three Partition Refinement Algorithms

Robert Paige [1,2]

Computer Science Department

Rutgers University

New Brunswick, NJ 08903

Robert E. Tarjan [2]

Computer Science Department

Princeton University

Princeton, NJ 08544

and

AT&T Bell Laboratories

Murray Hill, New Jersey 07974

January, 1986

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 Section 2, we consider the *lexicographic sorting problem*: given a set 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 Section 3, we consider the *relational coarsest partition problem*: given a partition P of a set U and a 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 $B_1 \cap E^{-1}(B_2) = \phi$. Here $E^{-1}(B_2)$ is the preimage set, $E^{-1}(B_2) = \{x \mid \exists y \epsilon B_2 \text{ such that } x E y\}$. Kanellakis and Smolka [8] studied this problem in connection with equivalence testing of CCS expressions [13]. 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.

(iii) In Section 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 [10] 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 algorithm running in $O(m \log(n+k) + n + k)$ time and $O(m+n+k)$ space.

We conclude in Section 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. Lexicograhic Sorting

Let Σ be an alphabet of k symbols and let Σ^* be the set of finite length strings over Σ. For $x \epsilon \Sigma^*$ 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 $x, y \epsilon \Sigma^*$, x is a *prefix* of y iff $|x| \leqslant |y|$ and $x(i) = y(i)$ for $1 \leqslant i \leqslant |x|$. Thus $x = y$ iff x is a prefix of y and $|x| = |y|$. Let $<$ be a total ordering of Σ. We extend $<$ to a total ordering of Σ^*, called *lexicographic order*, by defining $x < y$ iff either (i) $\exists k$ such that $x(k) < y(k)$ and $x(i) = y(i)$ for $1 \leqslant i < k$, or (ii) x is a prefix of y.

The *lexicographic sorting problem* is that of arranging a set of strings $U \subseteq \Sigma^*$ in increasing lexicographic order. In considering this problem we shall denote the strings in U by $x_1, x_2, \ldots, x_n$ and their total length $\sum_{i=1}^{n} |x_i|$ by m. We shall assume that

$\Sigma = \{1, 2, \ldots, k\}$ and that $<$ is numeric order on Σ.

For convenience we append to the end of each string the character 0, which serves as an end marker. For a string $x_i \epsilon U$, the *distinguishing prefix* of x_i in U, denoted by x_i', is the shortest prefix of x_i that is a prefix of no other string in U. The presence of end markers implies that x_i' is always defined. We denote the total length $\sum\limits_{i=1}^{n} |x_i'|$ of the distinguishing prefixes by m'.

Aho, Hopcroft, and Ullman [1] have 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. But an $O(m + k)$ time bound is not necessarily optimum. To sort the strings it suffices to scan only the distinguishing prefixes; thus we might hope for an $O(m' + k)$ time bound. We expect m' to be 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.

Sorting lexicographically in $O(m' + k)$ time seems to require scanning the strings from first position to last instead of vice-versa. Recently Mehlhorn [12] studied the special case of lexicographic sorting in which all strings have the same length. He concluded that any algorithm scanning strings from first position to last must use $\Omega(km)$ time (see [12], exercise 18, p. 100).

We shall 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, Ullman algorithm and disproving Mehlhorn's conclusion. The auxiliary space required by our algorithm (not counting space for the input strings) is $O(n + k)$. Our algorithm requires fewer passes than the Aho, Hopcroft, Ullman algorithm and may be superior in practice as well as in theory.

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 carry out the first step, 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. Once a string is in a singleton block, its distinguishing prefix is known. The algorithm maintains three objects: a partition P of the strings, a set S

containing the nonsingleton blocks of P, and an integer I indicating the 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 in S'.

> *Refine.* Remove B from S'. Partition B into blocks $B_1, B_2, \ldots, B_j$ on the basis of the Ith symbol in each string of B. Add each block B_i containing two or more symbols to S. For each block B_i containing only one string, say x, define the distinguishing prefix x' of x to be the prefix of length I.

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 I^{th} symbols, simultaneously making a list of all non-empty buckets. (When a string is added to a non-empty bucket, we add that bucket to the list of non-empty buckets.) Once the strings are distributed among the buckets, we examine each non-empty 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 [2]. 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. Let us call a refinement step during step one 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 S intact. We define a *refinement tree* T whose nodes are the distinct blocks produced by the algorithm, such that the children of a block B are the blocks into which B is split by a

proper refinement step. (If an improper refinement step is applied to a block, we regard the original block and the block produced by the refinement as being the same block.) Since each node of T has either no children or at least two children, T contains at most $2n-1$ nodes, i.e. at most $2n-1$ distinct blocks are produced during step one.

While performing step one, we compute the following additional information for use in step two. We represent each block created during step one by a record. We compute for each block B its parent $p(B)$ in the refinement tree. If block B is a singleton, we store with the block a pointer to the string it contains. We also construct a second auxiliary array of buckets numbered 0 through n. These buckets contain blocks. When a new block B_i is created during the examination of symbols in position I, we add B_i to the bucket corresponding to the Ith symbol of the strings in B_i. (The index of this bucket is the same as that of the bucket from which the strings of B_i were removed in forming B_i.) We call the Ith symbol of the strings in B_i the *associated symbol* of B_i. These extra computations increase the time and auxiliary space used in step one by only a constant factor.

With this extra information available, we can carry out step two by sorting the children of each block on their associated symbols and performing a preorder traversal of the resulting ordered refinement tree, listing the strings in the order the corresponding leaves of the tree are encountered. We order the tree as follows. For each block, we initialize its list of children to be empty. Then we scan the buckets containing the blocks in increasing order, emptying each bucket. When removing a block B from a bucket, we add B to the back of the list of children of block $p(B)$.

Reordering the entire 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 at the front of each bucket and blocks at the back. This saves $k+1$ words of space.

We conclude this section with three observations. First, the lexicographic sorting algorithm can be easily adapted to sort multisets, i.e. to allow duplicate strings. For this purpose we merely stop refining a block when it is a singleton or when its associated character is the end marker 0. Second, the algorithm can be simplified if the alphabet is of constant size, i.e. $k = O(1)$. In this case we can combine part of step two with step one, because we can afford to scan empty buckets of the array when forming new blocks during a partitioning 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 non-empty bucket. This allows us to construct the refinement tree with the

children correctly ordered during step one, and step two then consists merely of a preorder traversal of the refinement tree. We can obtain a further improvement by not constructing the refinement tree explicitly. Instead, we construct in step one a singly linked list of blocks in preorder with respect to the refinement tree; step two is merely a traversal of this list.

Our third 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 $O(m'/c + n + k^c)$, and the space used is $O(n + k^c)$. In the case of a binary alphabet, this means that we can sort in $O(m'/\log n + n)$ 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 [9] can be somehow 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 $x E y$. For any subset $S \subseteq U$, $E(S) = \{y \mid \exists x \in S \text{ such that } x E y\}$ and $E^{-1}(S) = \{x \mid \exists y \in S\} \text{ such that } x E y\}$. If P is a partition of U and $S \subseteq U$, P is *stable* with respect to S if for all blocks $B \in P$, either $B \subseteq E^{-1}(S)$ or $B \cap E^{-1}(S) = \phi$. A partition P is *stable* if it is stable with respect to each of its 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 one with fewest blocks. (As we shall show below, the coarsest stable refinement is unique.) In discussing time bounds for this problem, we shall let n denote this size of U and m the size of E. This problem was studied by Kanellakis and Smolka [8] in connection with testing congruence (a kind of equivalence) of finite state processes in the calculus of communicating systems (CCS) [13]. 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 shall develop an $O(m \log n)$ time algorithm, thereby verifying their conjecture. Our algorithm combines Hopcroft's "process the smaller half" strategy [1,6] 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 automation [1,4,6]. 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.

We begin by describing a naive algorithm for this problem. 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 $B \cap E^{-1}(S) \neq \phi$ and $B \cap (U-E^{-1}(S)) \neq \phi$ by the two blocks $B' = B \cap E^{-1}(S)$ and $B'' = B - E^{-1}(S)$. We call S a $splitter$ of Q if $split\,(S,Q) \neq Q$, i.e. Q is unstable with respect to S. 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 $split\,(S,Q)$.

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. Stability is inherited under refinement; that is, if R is a refinement of Q and Q is stable with respect to a set S then so is R. It follows that a given set S can be used as a splitter in the algorithm only once.

LEMMA 1. *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. Let B be a block of Q that is split and let B_1 be a block of R contained in B. We must show that either $B_1 \subseteq E^{-1}(S)$ or $B_1 \cap E^{-1}(S) = \phi$. Since set 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. For any block B_2 of R, either $B_1 \subseteq E^{-1}(B_2)$ or $B_1 \cap E^{-1}(B_2) = \phi$. If $B_1 \subseteq E^{-1}(B_2)$ for some block B_2 of R contained in S, then $B_1 \subseteq E^{-1}(S)$. If $B_1 \cap E^{-1}(B_2) = \phi$ for every block B_2 of R contained in S, then $B_1 \cap E^{-1}(S) = \phi$. It follows that R is a refinement of $split\,(S,Q)$. The lemma is true by induction. $\square$

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

$Proof.$ Each step increases the size of Q. Since the initial size is at least one and the final size is at most n, the algorithm terminates after at most $n-1$ refinement steps. Once no more refinement steps are possible, Q is stable, and by Lemma 1 any stable refinement is a refinement of Q. It follows that Q is the unique coarsest stable refinement. $\square$

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 $|E(\{x\})| \geqslant 1$ 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 $|E(\{x\})| \geqslant 1$ for all $x \in u$. This implies $m \geqslant n$.

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 $|B| \leqslant |S/2|$. Replace X by the partition formed by splitting block S into B and $S - B$ and replace Q by $split(S - B, split(B,Q))$.

The correctness of this improved algorithm follows from the correctness of the original algorithm and the observation made previously that if a partition is stable with respect to a set, any refinement of the partition is also stable with respect to the 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 $B_1 \cap E^{-1}(S-B) = \phi$, 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 is not correct. 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. Let us investigate the effect on the partition Q of refining first with respect to B and then with respect to $S - B$, given that Q is stable with respect to a set S that is a union of some of the blocks of Q. A block $D \in Q$ is split only if $D \cap R^{-1}(B) \neq \phi$, which implies $D \subseteq R^{-1}(S)$ since Q is stable with respect to a set S, that is Refining with respect to B splits D into two blocks, $D_1 = D \cap R^{-1}(B)$ and $D_2 = D - D_1$. (Either D_1 or D_2 may be empty.) Refining with respect to $S - B$ splits D_1 into two blocks, $D_{11} = D_1 \cap R^{-1}(S-B)$ and $D_{12} = D_1 - D_{11}$ (either of which may be empty), but it does not split D_2, since $D_2 \subseteq R^{-1}(S) - R^{-1}(B)$ implies $D_2 \subseteq R^{-1}(S-B)$. Performing this three-way splitting (of a block D into D_{11}, D_{12}, and D_2) is the hard part of the algorithm. The crucial observation that we shall use in our implementation is that since $D_1 \subseteq R^{-1}(B)$, $D_{12} = D_1 - R^{-1}(S-B) = D_1 \cap (R^{-1}(B) - R^{-1}(S-B))$.

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 $O(|B| + \sum_{y \in B} |E^{-1}(\{y\})|)$ 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 $x\,E\,y$ by a record that we shall call an *edge* and denote by $x\,E\,y$. 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 $x\,E\,y$ points to element x. Each element y points to a list

of the edges $x \, E \, y$. This allows scanning of the set $E^{-1}(\{y\})$ in time 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 $x \, \epsilon \, E^{-1}(S)$, 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 of $x \, E \, y$ such that $y \, \epsilon \, S$ contains a pointer to $count \, (x,S)$. Initially there is one count per vertex, $count \, (x,U) = |E(\{x\})|$; each edge $x \, E \, y$ 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 $m \geqslant n$.) The refinement algorithm consists of repeating refinement steps until $C = \phi$. 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 $R^{-1}(B)$). Copy the elements of B into a temporary set B'. (This facilitates splitting B with respect to itself during the refinement.) Compute $R^{-1}(B)$ by scanning the edges $x \, E \, y$ such that $y \, \epsilon \, B$ and adding each element x in such an edge to $R^{-1}(B)$. During the same scan, compute $count \, (x,B) = |\{y \, \epsilon \, B \mid x \, E \, y\}|$, 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 $R^{-1}(B)$, split D into $D_1 = D \cap R^{-1}(B)$ and $D_1 = D - D_1$. Do this by scanning the elements of $R^{-1}(B)$. To process an element $x \, \epsilon \, R^{-1}(B)$, determine the block D of Q containing it, and apply the appropriate one of the following three cases:

(a) If block D existed before the refinement and no new associated block D' has been created for it, create such a new block and move x into it.

(b) If block D existed before the split and it already has an associated new block D',

move x into D'.

(c)   If block D did not exist before the refinement (it is an associated block of some other block), do nothing; x has already been moved from its original block.

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

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

Step 7 (update counts). Scan the edges $x \, E \, y$ such that $y \in B'$. To process an edge $x \, E \, y$, decrement $count(x,S)$ (to which $x \, E \, y$ points); if this count becomes zero, delete the count record. Make $x \, E \, y$ point to $count(x,B)$ (to which x points). After scanning all the appropriate edges, scan the elements of B', resetting their pointers to counts so that they point to nothing.

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 $O(|B| + \sum_{y \in B} |E^{-1}(\{y\})|)$ 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 [15]. Observe that s-stability implies stability but not vice-versa.

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.

For a partition Q and set S, let *s-split* (S,Q) be the partition formed from Q by splitting each block $B \in Q$ into blocks $B_0, B_1, \ldots, B_k$, where $B_i = \{x \in B : |R(\{x\}) \cap S| = i\}$. (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 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 stable with respect to a set S and S is split into smaller blocks $S_1, S_2, \ldots, S_k$, refining with respect to all but one of the sets S_i guarantees 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 we do not need to refine with respect to its relative complement. 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 $O(|S| + \sum_{y \in S} |E^{-1}(\{y\})|)$ 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 Hopcroft's algorithm for the functional 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. (Hopcroft's algorithm must update L each time a block on 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.

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 Section 2, reading each row from left to right and each column from top to bottom.

Lubiw [10] 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 bottom to top and columns from right to left, and the required row and column order is nondecreasing lexicographic. 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 running in $O(m \log (n+k)^2 + n + k)$ time and $O(m+n+k)$ space. We shall develop an algorithm running in $O(m \log (n+k) + n + k)$ time and $O(m+n+k)$ space. Since all Lubiw's applications are for the special case of zero-one matrices, we shall 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 shall assume that $m \geqslant n \geqslant k$. We can validate the assumption $n \geqslant k$ by transposing the matrix if necessary, which takes $O(m)$ time. We can validate the assumption $m \geqslant n$ 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 shall begin by describing a slight generalization 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 shall 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 $e_1 \epsilon S_1 \epsilon P$, $e_2 \epsilon S_2 \epsilon P$ and $S_1 < S_2$ imply $e_1 < e_2$. 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. An ordered partition $R_1, R_2, \ldots, R_p$ of the rows of M and an ordered partition $C_1, C_2, \ldots, C_q$ 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 R_i and the columns in C_j. Let $B = M(R_i, C_j)$ be a matrix block. If $M(r,c) = M(r',c')$ for all $r, r' \epsilon R$, $c, c' \epsilon C_j$, block B is *constant*. We denote by $max(B)$ the maximum value in B. A *row slice* of B is the intersection of a row $r \epsilon R$ with B, i.e. the sumbatrix $M(r, C_j)$. A *splitting row* of B is a row $r \epsilon R_i$ 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, B' is *above* B if $i' < i$ and $j' = j$; B' is *left* of B if $i' = i$ and $j' < j$.

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 $C' = \{c \epsilon C \,|\, M(r,c) = max(B)\}$, $C'' = \{c \epsilon C \,|\, M(r,c) < max(B)\}$. If B has no splitting row, replace R in the ordered row partition by the ordered pair $R' = \{r \epsilon R \,|\, M(r,c) = max(B) \;\forall\; c \epsilon C\}$, $R'' = \{r \epsilon R \,|\, \exists c \epsilon C$ such that $M(r,c) < max(B)\}$.

This algorithm differs in two ways from Lubiw's. First, hers requires that the matrix block $B = M(R,C)$ selected in a refinement step have R minimum among all non-constant blocks, and among blocks of equal R have C minimum. Her proof of correctness is valid without change for our generalization. Second, if there is no splitting row, her algorithm chooses any splitting column c (defined analogously to a splitting row) and replaces the row block R by the ordered pair $R' = \{r \epsilon R \,|\, M(r,c) = max(B)\}$, $R'' = \{r \epsilon R \,|\, M(r,c) < max(B)\}$. 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 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 Section 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 shall 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 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 shall 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 and points to a doubly linked list of its rows. Each column block has its size and points to a doubly linked list of its columns. Each row slice has its size 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 Figure 1.) Initializing the data structure takes $O(m)$ time.

[Figure 1]

To perform a refinement step, we remove the first block $B = M(R,C)$ from the list of matrix blocks. We determine the row blocks R and 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 (with $M(R,C')$ in front). 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. Each constant amount of time charged to the size of an entry corresponds to the size of either its row block or its column block falling by at least a factor of two. It follows that 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. Finding a row slice s in B with fewer than $|C|$ entries takes $O(1)$ time plus $O(1)$ time per row slice in B with $|C|$ entries. 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. Finding the columns in new block C' takes $O(|s|)$ 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 size of new blocks C' formed during the running of the algorithm is $O(m)$. Removing the rows of C' from C'' takes $O(|C'|)$ time. Suppose that C' has fewer rows 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 just as in Section 3 we formed new blocks of the partition. 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 new 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 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.

We shall describe the changes in the zero-one algorithm needed to handle the general case. We use one new concept. 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 we shall not distinguish 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 nonincreasing 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.* Red-black trees [5,16] or weak B-trees [7,11] 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

---

* By *amortized time* we mean the time per operation averaged over a worst-case sequence of operations. See Tarjan's survey paper [17] for a full discussion of this concept.

updating of the overlay lists. Consider a split of a row block R into R' and R''. Without loss of generality, suppose the entries in R' are processed to perform the updating. Let B be a block containing entries in R'. As entries in R' 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 column block splits is similar, except that overlays do not split but are merely moved from one list to another by unmerging operations.

Excluding the unmerged 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 $p \geqslant |L'| + |L''|$, $q \geqslant |L'|$, 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 containing B. Then $T(x)$ obeys the recurrence $T(1) = 0$, $T(x) \leqslant \max_{y < x}\{T(y) + T(x - y) + O(y \log(x/y))\}$ for $x \geqslant 2$. 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 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 [14] 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 [3].

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 [5,16] or a weak B-tree [7,11], although many other kinds of balanced trees will do as well. We assume some familiarity with balanced search trees. There are there steps to the algorithm.

Step 1. Color all nodes indicated by pointers red 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 red, 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 [2,7]. 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 red.

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 [2,7,11,16]).

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 $O(1)$ [7,11,16]. The total worst-case deletion time is $O(q(1+\log(p/q)))$ [2,7]. 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.

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," *SIAM J. Comput.* 9(1980), 594-614.

[3] P. J. Downey, R. Sethi, and R. E. Tarjan, "Variations on the common subexpression problem," *J. Assoc. Comp. Mach.* 27(1980), 758-771.

[4] D. Gries, "Describing an algorithm by Hopcroft," *Acta Informatica* 2(1973), 97-109.

[5] L. J. Guibas and R. Sedgewick, "A dichromatic framework for balanced trees," *Proc. 19th Annual IEEE Symp. on Foundations of Computer Science* (1978), 18-21.

[6] J. Hopcroft, "An $n \log n$ algorithm for minimizing states in a finite automation," *Theory of Machines and Computations*, Z. Kohavi and A. Paz, eds., Academic Press, New York, 1971, 189-196.

[7] S. Huddleston and K. Mehlhorn, "A new data structure for representing sorted lists," *Acta Informatica* 17(1982), 157-184.

[8] P. C. Kanellakis and S. A. Smolka, "CCS expressions, finite state processes, and three problems of equivalence," *Proc. ACM Symp. on Principles of Distributed Computing* (1983), 228 240.

[9] D. Kirkpatrick and S. Reisch, "Upper bounds for sorting integers on random access machines," *Theoretical Computer Science* 28(1984), 263-276.

[10] A Lubiw, "Doubly lexical orderings of matrices," *Proc. 17th Annual ACM Symp. on Theory of Computing* (1985), 396-404.

[11] D. Maier and S. C. Salveter, "Hysterical B-trees," *Info. Process. Lett.* 12(1981), 199-202.

[12] K. Mehlhorn, *Data Structures and Algorithms 1: Sorting and Searching*, Springer-Verlag, Berlin, 1984.

[13] R. Milner, *A Calculus of Communicating Systems, Lecture Notes in Computer Science 92*, Springer-Verlag, Berlin, 1980.

[14] R. Paige, R. E. Tarjan, and R. Bonic, "A linear time solution to the single function coarsest partition problem," *Theoretical Comp. Sci.* 40(1985), 67-84.

[15] R. C. Read and D. G. Corneil, "The graph isomorphism disease," *J. Graph Theory* 1(1977), 339-363.

[16] R. E. Tarjan, *Data Structures and Network Algorithms*, Society for Industrial and

Applied Mathematics, Philadelphia, PA, 1983.

[17] R. E. Tarjan, "Amortized computational complexity," *SIAM J. Alg. Disc. Math.* 6(1985), 306-318.

LIST OF
ROW BLOCKS

↓

ROW BLOCKS

↕

ROWS

LIST OF
MATRIX BLOCKS

↓

MATRIX BLOCKS

↕

ROW SLICES

↕

LIST OF
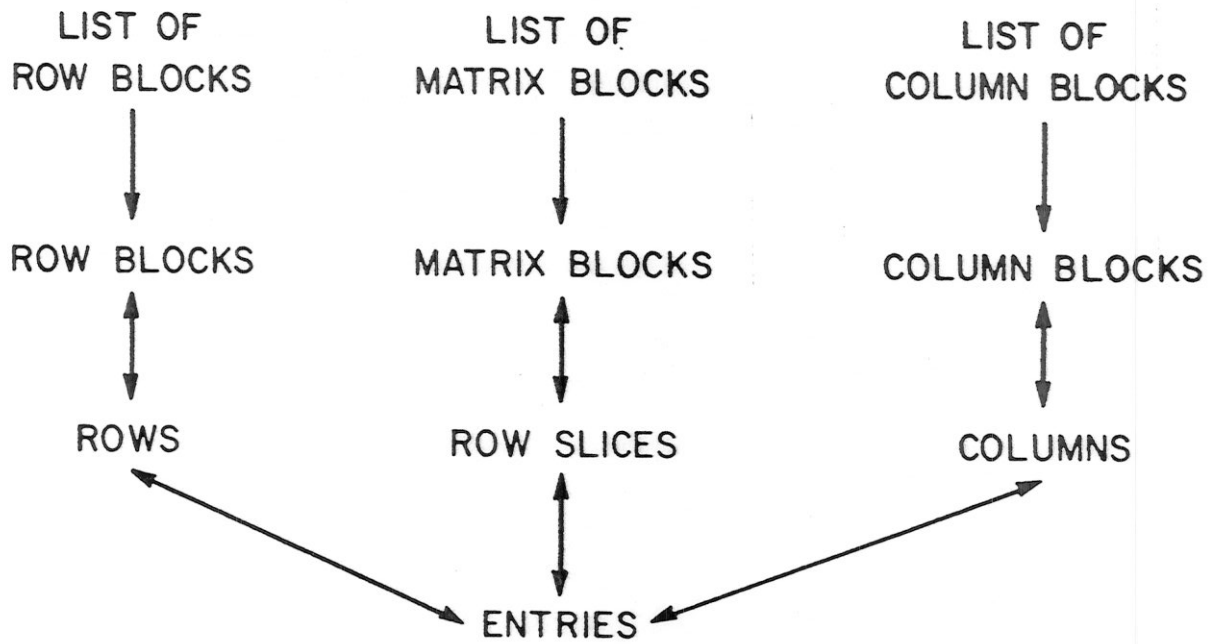COLUMN BLOCKS

↓

COLUMN BLOCKS

↕

COLUMNS

ENTRIES

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