November, 1985

A NOTE ON UNMERGING

B. Reed*, J.S. Salowe** and W.L. Steiger**

DCS-TR-161

*Department of Computer Science
McGill University

**Department of Computer Science
Rutgers University

Department of Computer Science
Hill Center for the Mathematical Sciences
Busch Campus
Rutgers University
New Brunswick, NJ 08903

A NOTE ON UNMERGING

ABSTRACT

In the Summer '85 issue of SIGACT News, N. Santoro enquired about the space-time complexity of unmerging. Suppose that lists A and B, each of size n/2, are merged to produce the list L. The problem is to separate L into A and B in their original order. By studying a simple, but non optimal stable merging algorithm, we obtain an unmerging algorithm which, using extra space O(k), runs in time $O(n\log_k n)$. Thus, given $\epsilon > 0$, the algorithm can unmerge in linear time provided $O(n^\epsilon)$ space is available; if only O(logn) extra space can be used, the algorithm will need O(nlogn/(loglogn)) time.

1 INTRODUCTION AND OVERVIEW

In [4], N. Santoro introduced the unmerging problem: Two sorted lists A and B, each of size n/2 are merged to produce the sorted list L. The problem is to separate L into the two constituent sublists A and B, each in their original order.

A fundamental question posed by this problem concerns the space-time complexity of unmerging. "Time" takes into account comparisons, movement of data, and pointer manipulation; "space" involves any extra storage used beyond the n locations in which the input list L is presented and would include storage for maintaining any pointer values. Amongst the specific queries put forth by Santoro, one involves the time complexity of unmerging if only constant extra space is available. Another asks for the extra space needed if the unmerge is to be done in time O(n).

These questions have an added interest because merging is so well understood. Indeed [2] there exists an algorithm to merge A and B in linear time and using constant extra space, and one [3] which will even do it in a stable fashion. Both algorithms may be shown to be optimal [1] except possibly with regard to the values of constants of propor-

tionality in the space-time bounds. In view of the tight bounds on merging complexity it is tempting to imagine unmerging by somehow reversing the operations in the optimal merge, while retaining its performance characteristics (see [4]). However the algorithm in [3] is complex and it is not clear what it means to reverse its operations on an already merged input list.

With extra space n/2 it is trivial to unmerge L in linear time. Santoro [4] shows how to reduce the space to $O(n^{1/2})$, still using linear time. In the present paper we describe a stable unmerging algorithm which, using extra space O(k), can separate L in time $O(n \log_k n)$. This implies that if $O(n^{\epsilon})$ extra space is available, $\epsilon > 0$ given, unmerging can be performed in linear time. Furthermore is is interesting that the algorithm is related to a merge algorithm with the same control structure and possessing the *same* space-time complexity; this merge is necessarily suboptimal. The structure is depicted in the following figure.

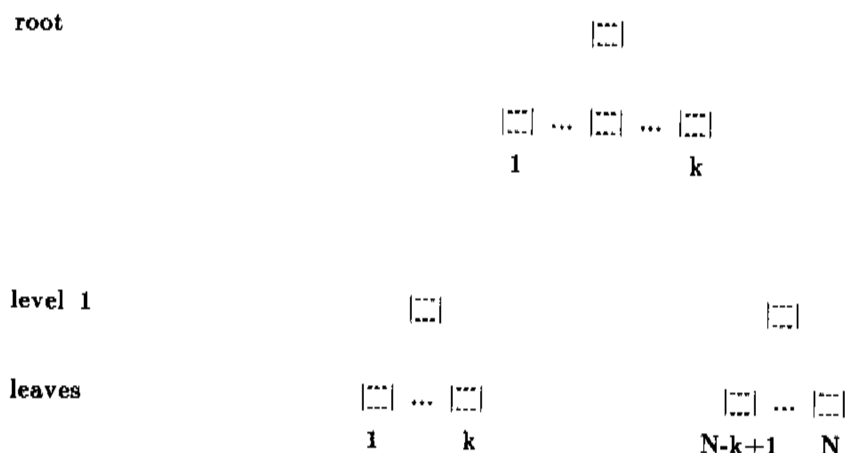

Fig. 1. Control Structure of the Unmerge and Merge Algorithms

Let k be a given positive integer less than n. Both merge and unmerge traverse a k-ary tree. In the above diagram each box is a node with k children. The leaf nodes represent the input list. Null elements are added until there are $N = k^d$ leaves, where $d = \lceil \log_k n \rceil$ is the depth of the tree. All parent nodes are obtained by organizing the elements

of its k offspring blocks into one large block. Each level thus contains a rearrangement of the elements in the preceeding level and has $N = O(n)$ entries. Continuing up to the top, the root node at level d will be the final, desired form for the input list: in the case of merging, it will be L; for unmerging, it will be L separated into A|B|Nulls.

We refer to the process of combining k offspring blocks into one large parent block as the blocking algorithm. The blocking algorithm for unmerging differs from that of merging. In the former case, parents of leaf nodes are each blocks of k elements divided into two groups of consecutive A's followed by consecutive B's. The A's in successive blocks are consecutive as are the B's. To move up to level 2, k of these blocks are combined into a block of k^2 elements, again consisting of consecutive A's followed by consecutive B's. At the root node, L is unmerged. The details are described in the next section.

For merging the blocking is conceptually similar. A node at level j represents k^j consecutive elements of the final merged list, but the blocks on that level may be out of order and a few tricks are necessary in representing the blocks. Accordingly, the details for achieving this structure in linear time and space $O(k)$ are more difficult to describe and they will be omitted.

Suppose we are on level j of the tree where there are k^{d-j} blocks, each of size k^j elements and assume that the blocking algorithm has the following properties:

1. k blocks may be combined into one parent block in time $O(k^{j+1})$
2. k blocks may be combined into one parent block using extra space $O(k)$

These assumptions imply that $O(n)$ time and $O(k)$ extra space are sufficient to pass to level j+1 possessing k^{d-j-1} blocks, each of size k^{j+1}. The $O(n \log_k n)$ time bound now follows because there are d levels. In the next section we describe the blocking algorithm and show that it has these two required properties.

First it is worth observing that no tree traversal algorithm of the type just outlined, and satisfying the two blocking assumptions, can operate in linear time and constant space; constant space requires $O(n \log n)$ time. It remains a possibility that unmerging is inherently

more complex than merging with respect to space-time. In fact linear time might even force $O(n^{\epsilon})$ extra space in which case our algorithm would be optimal. However because the companion merge is so inefficient in space, we believe that a more efficient unmerge may be constructed along the lines of optimal merging.

2 THE ALGORITHM

We begin by describing the blocking algorithm for unmerging. It takes as inputs K, the branching factor of the tree, J, the current level, and L, the current reorganization of the data. If J is 0, L contains the original merged list of A's and B's followed by the null elements. For $J > 0$, L is organized into blocks of size K^J. Each non-null block begins with a group of A's in order, followed by a group of B's, also in order. The A's in contiguous blocks are consecutive A's, as are the B's. The root block (J=d) is completely unmerged.

ALGORITHM BLOCK

Input:

K	branching factor
J	level in control tree
L	merged list reorganized for level J into K^{d-J} unmerged blocks of size K^J each
N	the size of L

Output:

L	elements reorganized into unmerged blocks of size K^{J+1}

Variables:

SIZE	K^J, the size of each block
NA,NB	arrays of size K+1. NA[I] is the number of A's encountered by end of block I; NA[0] = 0. Similarly with NB
B1	address of the first B in the big block being built
BLK,I	indexes
T1,T2,TO	
FRM	temporary storage
START	address of start of bigblock being formed

Functions:

TYPE returns 'A', 'B' or 'NULL' for any element of L

Initialize

```
[0]   SIZE ← K**J; START ← 1
[1]   BLK ← 1; B1 ← 0; NA[0] ← 0; NB[0] ← 0
[2]   WHILE BLK ≤ K
[3]      I ← 0
[4]      WHILE TYPE{L[START+I+(BLK-1)*SIZE]} = 'A'
[5]         I ← I + 1
[6]      NA[BLK] ← NA[BLK-1] + I
[7]      NB[BLK] ← NB[BLK-1] + SIZE - I
[8]      BLK ← BLK + 1
[9]   B1 ← NA[K] + START
[10]  I ← NA[1] + START
```

Build Big Block:

```
[11]  WHILE I < B1
[12]  FRM ← I
[13]  T1 ← L[FRM]
[14]  WHILE TYPE{T1} = 'B'
[15]     BLK ← ⌈FRM/K⌉
[16]     TO ← B1 + NB[BLK-1] + FRM - NA[BLK]
[17]     IF TYPE{T1} = 'A' THEN
[18]        TO ← NA[BLK-1] + FRM - (BLK-1)*SIZE
[19]     WHILE TO ≠ I
[20]        T2 ← L[TO]
[21]        L[TO] ← T1
[22]        T1 ← T2
[23]        FRM ← TO
[24]        GO 15
[25]     L[I] ← T1
```

Loop:

```
[26]  START ← START + K*SIZE
[27]  IF START < N GO 1
```

The input to the algorithm is depicted below:

```
[...A.......|..B..] [..A..|.....B.....] ... [......A......|..B..]
        1                  2                        k
```

Each block has size K^J, begins with some consecutive A's, and ends with some consecutive B's. A's and B's are also ordered between successive blocks. Block 1 begins at START, initially equal to 1. This structure is reorganized into one large block of size k^{J+1} beginning with the consecutive A's and ending with the consecutive B's.

The initialize phase defines parameters and computes the partial sums of the number of 'A' and 'B' elements over the non-null block. The top **WHILE** loop passes through each of the K blocks, one element at a time, so it has time complexity is $O(K^{J+1})$. Its space is O(K) because of the two arrays, NA and NB.

The actual blocking is performed by the next set of instructions. Elements are processed sequentially starting with the first B in block 1. For an element x, we compute the address of its final position, p_x, in the bigblock. If p_x is also its current position, no action is taken and we iterate to the next element. Otherwise we move x to position p_x and then compute the final position of the element that was just displaced from p_x, etc. By the time we iterate to position B1, all the A's and all the B's will have been blocked and sorted. Each element is examined at most two times, once when it is moved to its final position and possibly once more, when it is examined and found to be already in its final position. Once in its final position, it can never be displaced because no other item has the same final position. The time complexity for building a big block is thus $O(K^{J+1})$ and the extra space is O(K).

The details for computing final positions are easy. If the item is a B, its final position will be B1 plus the number of B items that precede it amongst the K blocks currently being processed. NB[BLK-1] counts the B's in earlier blocks and FRM-NA[BLK] counts the preceding B's in the current block. This is statement 16. An A item will move to final position START+x if x-1 A items precede it amongst the K blocks under consideration. There are NA[BLK-1] A items in preceding blocks and FRM-SIZE*(BLK-1) A items that precede the A item located at FRM in the current block. This is captured by statement 18.

The looping assures that we move through L, reblocking groups of K^{J+1} items, one at a time. If the blocking algorithm is run iteratively from J=0 through J=d-1, a breadth-first traversal of the control tree will occur. Alternatively, as soon as K large blocks have been formed at the current level, they may be reblocked on the next level, etc. This would correspond to the depth-first traversal.

For the breadth-first algorithm, the overall space requirement is O(K). The time costs are $O(n \log_k n)$, O(n) steps at each of the d levels of the tree. Thus if $k = n^\epsilon$ for some given $\epsilon > 0$, unmerging may be achieved in linear time. If only k = logn extra space is available, the time required by this algorithm is O(n logn/(loglogn)) while if k is fixed for all n, O(n logn) time is needed. The unmerge is stable because items of the same type are placed sequentially, always to positions beyond those occupied by previously placed items.

References

1. Horvath, Edward C. Efficient Minimum Extra Space Stable Sorting. Proceedings of the 6th Annual Symposium of the Theory of Computing (SIGACT 6), Association for Computing machinery, New York, 1974, pp. 194-215.

2. Kronrod, M.A. "Optimal Ordering Algorithm Without Operational Field". *Soviet Math. Doklady 10* (1969), 744-746.

3. Pardo, Luis Trabb. "Stable Sorting and Merging with Optimal Space and Time Bounds". *SIAM J. Computing 6* (1977), 351-372.

4. Santoro, N. "On the Unmerging Problem". *SIGACT News 17-1* (1985), 5.