BIT 32 (1992), 580–585.

STABLE MINIMUM SPACE PARTITIONING IN LINEAR TIME

JYRKI KATAJAINEN and TOMI PASANEN

Department of Computer Science,
University of Copenhagen,
Universitetsparken 1,
DK-2100, Copenhagen East, Denmark

Department of Computer Science,
University of Turku,
Lemminkäisenkatu 14 A,
SF-20520 TURKU, Finland

Abstract.

In the stable *0-1* sorting problem the task is to sort an array of n elements with two distinct values such that equal elements retain their relative input order. Recently, Munro, Raman and Salowe gave an algorithm which solves this problem in $O(n \log^* n)$[†] time and constant extra space. We show that by a modification of their method the stable *0-1* sorting is possible in $O(n)$ time and $O(1)$ extra space. Stable three-way partitioning can be reduced to stable *0-1* sorting. This immediately yields a stable minimum space quicksort, which sorts multisets in asymptotically optimal time with high probability.

CR categories: E.5, F.2.2.

The *stable 0-1 sorting* problem is defined as follows: Given an array of n elements and a function f mapping each element to the set $\{0, 1\}$, the task is to rearrange the elements such that all elements, whose f-value is zero, come before elements, whose f-value is one. Moreover, the relative order of elements with equal f-values should be maintained. For the sake of simplicity, we hereafter refer to bits instead of the f-values of elements.

Stable partitioning is a special case of stable 0-1 sorting, where the f-values are obtained by comparing every element x_i to some pivot element x_j (which will not take part in partitioning):

$$f(x_i) = \begin{cases} 0 \text{ for } x_i < x_j, \text{ or } x_i = x_j \text{ and } i < j \\ 1 \text{ for } x_i > x_j, \text{ or } x_i = x_j \text{ and } i > j. \end{cases}$$

Another interesting special case is the *stable unmerging* problem studied in [9]: Given two lists A and B stably merged together into a single list L, the task is to separate L into its constituent sublists A and B in their original order. The informa-

† $\log^* n$ is defined as $1 + \log^*(\log_2 n)$, if $n > 2$, and otherwise it is 1.

Received April 1991. Revised February 1992.

tion associated with each element, indicating from which of the sublists the element originated, corresponds to its f-value. In the case of equal-valued elements, A-elements are considered to be smaller than B-elements. Therefore, in the stable unmerging problem, one never sees a block of equal-valued A-elements intermixed with B-elements, or vice versa. (Observe that the algorithm of Salowe and Steiger [9] makes use of this fact and hence it cannot be immediately adapted to solve the stable 0-1 sorting problem.)

Recently, Munro, Raman, and Salowe [6] showed that stable 0-1 sorting is possible in $O(n\log^* n)$ time when only $O(1)$ extra space is available. In this note we improve this result by reducing the running time to $O(n)$ still maintaining the space bound.

Before proceeding we should define precisely what we mean by a *minimum space* or *in-place* algorithm. In addition to the array containing the n elements, we allow one storage location for storing an array element. This is needed when swapping two data elements. The elements are regarded to be atomic. They can only be moved and used as arguments when computing f-values. Of course the evaluation of the f-function is assumed to be a constant time operation. Moreover, we assume that a constant number of extra storage locations, each capable of storing a word of $O(\log_2 n)$ bits, is available and that operations $\{<, =, >, +, -, \text{shift}\}$ take constant time for these words. An *unrestricted shift operation* takes two integer operands v and i and produces $\lfloor v \cdot 2^i \rfloor$. It should be observed that our model of computation is not as general as that of Munro, Raman, and Salowe [6] since their algorithms can be implemented without the shift operation.

Next we briefly review the techniques used in our minimum space algorithm.

Blocking: The input array is divided into equal sized blocks. Most efficient in-place algorithms in the literature are based on the blocking technique.

Block interchanging: A block A can be reversed in-place in linear time by swapping the pair of end elements, then the pair next to the ends, etc. Let A^R be A reversed. The order of two consecutive blocks (not necessarily of the same size) A and B may be interchanged by performing three block reversals, namely $BA = (A^R B^R)^R$. This idea seems to be part of computer folklore.

Packing small integers: Let us assume that we have t small integers $i_1, \ldots, i_t$, each of which can be represented by m bits. That is, the integers are from the domain $[0, 2^m - 1]$. Further, assuming that $t \cdot m \leq \log_2 n$, the integers can be packed into one word w of $\log_2 n$ bits. The integer i_j, $1 \leq j \leq t$, is stored by using the bits $(j-1)m, \ldots, jm - 1$ of w. Each integer is easily recovered from w in constant time if multiplications and divisions by a power of two are constant operations. The value v of i_j is obtained as follows: $v = \{w - [(w \text{ shift } -jm) \text{ shift } jm]\} \text{ shift } -(j-1)m$. (Observe that in our algorithm m can be chosen to be a power of 2, so we do not need general multiplication.) With a code similar to this the value of i_j can be updated. Previously the packing technique has been used for example in [2, 4].

Without loss of generality, we can assume that n, the number of elements, is a power of 2. If this is not the case, the following recursive method can be used to reduce the original problem to subproblems, whose size is a power of 2. First, compute by repeated doubling the smallest 2^k such that $2^k \leq n < 2^{k+1}$. Second, partition the first 2^k elements with Algorithm D to be given later. Third, call the same method recursively for the last $n - 2^k$ elements. Finally, interchange the block of ones (if any) among the first 2^k elements with the block of zeros (if any) lying after the ones. This method runs clearly in linear time if the running time of Algorithm D is linear. It is an easy matter to establish an iterative implementation of the method above without using any recursion stack, that is $O(1)$ extra space is enough here.

The stable 0-1 sorting problem is easily solved in linear time when $O(n)$ extra space is available. We describe Algorithm A that performs this. The algorithm first computes the total number of zeros in the array. Let this number be Z_{total}. Then the input array is scanned another time. During this scan two counters C_0 and C_1 are maintained, the former counting the number of zeros and the latter the number of ones to the left of the current position. Together with each element $b_i \in \{0, 1\}$ its *rank* $b_i Z_{total} + C_{b_i} + 1$ is stored. Reserving one more bit for each element, telling whether the element is in its final position or not, the elements are permuted to their final positions (for details, see [6; Lemma 1]). Hence we have

LEMMA 1. *Algorithm A sorts stably a bit-array of size n in O(n) time with n bits and* $n + O(1)$ *counters, each requiring at most* $\lceil \log_2(n + 1) \rceil$ *bits.*

Munro, Raman, and Salowe [6; Lemma 3] presented an algorithm similar to Algorithm A but they did not state explicitly the size of the counters.

Another building block is Algorithm B developed by Munro, Raman, and Salowe. Its performance is given in the following

LEMMA 2. [6; Theorem 1] *Algorithm B sorts stably a bit-array of size n in* $O(n \log_2 n)$ *time and constant extra space, but makes only* $O(n)$ *moves.*

To improve these algorithms we divide the input into blocks of size $\lg n$, which denotes the smallest power of 2 greater than or equal to $\log_2 n$. Since n is a power of 2, it is divisible by $\lg n$. Now the *basic steps* of our algorithms are

(i) *element sorting*: Sort stably every block of $\lg n$ elements. When sorting the blocks the same storage space is used.
(ii) *transformation from sorted blocks to typed blocks*: Rearrange the elements stably such that each block (except perhaps the last one) contains only either zeros or ones. We say that a block is of *type* 0 or 1 depending on its contents.

Munro, Raman and Salowe [6; Lemma 2; Step 3] showed that this transformation from sorted blocks to typed blocks is possible in $O(n)$ time and $O(1)$ extra space. (Their proof is for blocks of size $\sqrt{n}$, but it is not difficult to see that the same method works for any block size.)

(iii) *block sorting*: Sort stably the blocks according to their type.

(iv) *cleaning up*: Interchange the zeros (if any) in the last block into their correct positions.

When following the basic steps we have to specify the routines that are used in element sorting and block sorting. In Algorithm C, Algorithm A is applied in both places. Therefore we have

LEMMA 3. *Algorithm C sorts stably a bit-array of size n in $O(n)$ time with $\lg n + n/\lg n$ bits and $\lg n + n/\lg n + O(1)$ counters, each requiring at most $\lceil \log_2(n+1) \rceil$ bits.*

The final algorithm, Algorithm D is also based in $\lg n$-blocking. Now, however, the elements are sorted by Algorithm C and the typed blocks by Algorithm B. When sorting the elements of a block, we have to store $O(\log_2 n/\log_2 \log_2 n)$ counters, each of $O(\log_2 \log_2 n)$ bits (and $O(1)$ indices, each of $O(\log_2 n)$ bits). The total number of bits required is only $O(\log_2 n)$. Therefore we can pack the integers into a few words and manipulate them efficiently with shift operations.

Algorithm D is used for proving our main result.

THEOREM 1. *A bit-array of size n can be stably sorted in $O(n)$ time and $O(1)$ extra space.*

PROOF. The most critical part of Algorithm D is element sorting. But, due to constant-time shift operations, each block is sorted in $O(\log_2 n)$ time. Block sorting requires $O(n/\lg n \cdot \log_2(n/\lg n))$ time for comparisons and pointer manipulations, and $n/\lg n$ block moves; that is $O(n)$ time in total. Since all the steps element sorting, creation of typed blocks, block sorting, and cleaning up take linear time, the overall running time of Algorithm D is $O(n)$. ■

Algorithm D is quite complicated. It is therefore natural to ask, whether there exists a simpler algorithm which solves the stable 0-1 sorting problem in minimum space. (Here it should be observed that, if it is only required to maintain the stability of zeros *or* ones, a simple algorithm exists which is based on the wheel technique [1; Section 10.2].) We consider the technique of packing small integers important and believe that it can be used in other applications as well. However, the technique requires that the shift operation takes constant time. Can stable, in-place 0-1 sorting be done in linear time by allowing only comparisons, movement of data, additions, and subtractions?

By computing the minimum, performing stable 0-1 sorting such that elements equal to the minimum are interpreted as zeros and other elements as ones, and repeating this for the ones, we obtain

THEOREM 2. *An array of n elements with k distinct values can be stably sorted in $O(kn)$ time and $O(1)$ extra space.*

In a decision tree model, $\Omega(n\log_2 n - \Sigma_{i=1}^{k} n_i \log_2 n_i + n)$ is a lower bound for sorting a multiset with multiplicities $n_1, n_2, \ldots, n_k$ (where $n = \Sigma_{i=1}^{k} n_i$) [5]. An interesting open question is whether one can improve Theorem 2 and devise an in-place algorithm that sorts multisets stably in asymptotically optimal time.

We can give a partial answer to this question, since quicksort can be adapted to sort a multiset by doing a three-way partition at each recursive step [12]. If elements less than, equal to, and greater than the pivot are considered to have values 0, 1, and 2, respectively; the stable three-way partitioning will reduce to stably sorting of elements with three distinct values. By avoiding the recursion stack as proposed in [3] or [13], applying the algorithm of Theorem 2 in partitioning, and combining this with the analysis of Seidel [10], we get the following

THEOREM 3. *With probability $1 - O(n^{-\alpha(\log_2 \alpha - 1)})$, for any constant $\alpha \geq 1$, randomized quicksort sorts stably a multiset of size n with multiplicities $n_1, n_2, \ldots, n_k$ in $O(\alpha n\log_2 n - \Sigma_{i=1}^{k} n_i \log_2 n_i + n)$ time and $O(1)$ extra space.*

PROOF. First of all, one should observe that the running time of quicksort is proportional to the total number of comparisons performed. Since we are not interested in constant factors, we assume that at each partition every element involved is compared to the pivot only once. In [10] (for similar results, see for example [8, 11]) it has been proved that, with probability $1 - O(n^{-\alpha(\log_2 \alpha - 1)})$, none of the elements will take part in more than $2\alpha\log_e n$ partitions. Due to the three-way partitions, all redundant comparisons between a pivot and elements equal to the pivot are avoided. This means that $n_i \log_2 n_i - O(n_i)$ comparisons are saved for a class of n_i equal elements (cf. [7; Theorem 3.1]). From this the claim follows. ∎

Acknowledgements.

We thank Christos Levcopoulos for introducing us to the technique of packing small integers; and Svante Carlsson, Jeffrey Salowe, and Jukka Teuhola for their help. We are also grateful to a referee whose report helped to improve the presentation of this note.

Note added in proof.

We have been able to solve the open problem posed after Theorem 2. This finding will be presented at the 3rd Scandinavian Workshop on Algorithm Theory.

REFERENCES

1. J. Bentley, *Programming Pearls*, Addison-Wesley, 1986.
2. S. Carlsson, J. I. Munro, P. V. Poblette, *An implicit binomial queue with constant insertion time*, 1st Scandinavian Workshop on Algorithm Theory, Lecture Notes in Computer Science 318, Springer-Verlag, 1988, pp. 1–13.
3. D. Ďurian, *Quicksort without a stack*, Mathematical Foundations of Computer Science 1986, Lecture Notes in Computer Science 233, Springer-Verlag, 1986, pp. 283–289.
4. C. Levcopoulos, O. Petersson, *An optimal adaptive in-place sorting algorithm*, 8th International Conference of Fundamentals of Computation Theory, Lecture Notes in Computer Science 529, Springer-Verlag, 1991, pp. 329–338.
5. J. I. Munro, V. Raman, *Sorting multisets and vectors in-place*, 2nd Workshop on Algorithms and Data Structures, Lecture Notes in Computer Science 519, Springer-Verlag, 1991, pp. 473–480.
6. J. I. Munro, V. Raman, J. S. Salowe, *Stable in situ sorting and minimum data movement*, BIT 30 (1990) 220–234.
7. J. I. Munro, P. M. Spira, *Sorting and searching in multisets*, SIAM Journal on Computing 5 (1976) 1–8.
8. P. Raghavan, *Lecture notes on randomized algorithms*, Computer Science Report RC 15340, IBM Research Division, T. J. Watson Research Center, 1990.
9. J. S. Salowe, W. L. Steiger, *Stable unmerging in linear time and constant space*, Information Processing Letters 25 (1987) 285–294.
10. R. Seidel, *Backwards analysis of randomized geometric algorithms*, Technical Report, Computer Science Division, University of California Berkeley, 1991.
11. S. Sen, *Random sampling techniques for efficient parallel algorithms in computational geometry*, Ph.D. thesis, Computer Science Department, Duke University, 1989.
12. L. M. Wegner, *Quicksort for equal keys*, IEEE Transactions on Computers C34 (1985) 362–367.
13. L. M. Wegner, *A generalized, one-way, stackless quicksort*, BIT 27 (1987) 44–48.