ON A STABLE MINIMUM STORAGE MERGING ALGORITHM

Krzysztof DUDZIŃSKI and Andrzej DYDEK
Institute of Informatics, Warsaw University, Poland

Received 5 May 1980; revised version received 27 November 1980

Analysis of algorithms, stable merging

1. Introduction

A sorting algorithm is said to be stable when it preserves the order of occurrence of equal elements and is said to require minimum storage if it uses only $O((\log N)^2)$ space besides the space needed to store N records. An algorithm using O(1) space is said to be in place. 'Is there a stable minimum storage sorting algorithm which requires less than $O(N^2)$ units of time in its worst case, and/or on the average?' Knuth posed this question [2].

This problem was solved by Trabb Pardo [4], whose stable algorithm merges two ordered vectors of length m and n performing O(m + n) assignments and comparisons and its application to sort a vector of length N works in place in running time O(N log N).

The aim of this paper is to present a stable algorithm of merging vectors of length m and n ($m \leqslant n$) performing $O(m \log(n/m + 1))$ comparisons and $O((m + n) \log m)$ assignments in $O(\log m)$ space.

2. An optimal in place algorithm of circular shift

The basic procedure in this merging algorithm is an in place algorithm of vector exchange, transforming vector uv, $m = |u|$, $n = |v|$, into vector vu.

There is a known vector exchange algorithm [3] performing $3(\lfloor (m+n)/2 \rfloor + \lfloor n/2 \rfloor + \lfloor m/2 \rfloor)$ assignments. The problem of vector exchange can be solved with $m + n + \gcd(m, n)$ assignments, where gcd(m, n) is the greatest common divisor of m and n.

The vector exchange can be reduced to the following problem: shift in place vector $w = (w_0, w_1, w_2, \ldots, w_{\ell-1})$ by n ($0 < n < \ell$) positions to the right obtaining vector $(w_{\ell-n}, \ldots, w_{\ell-1}, w_0, \ldots, w_{\ell-n-1})$.

Let us define:

(1) $w = (w_0, \ldots, w_{\ell-1})$, $S = \{0, 1, \ldots, \ell - 1\}$, $0 < n < \ell$, $p = \gcd(\ell, n)$.
(2) $\sigma : S \to S$, $x = \mathrm{mod}(x + n, \ell)$, where mod(a, b) is the remainder of dividing a by b.
(3) $S_i = \{j \in S: \mathrm{mod}(j, p) = i\}$ for $i = 0, \ldots, p - 1$.

Theorem 2.1. σ is a permutation with cycles $S_0, \ldots, S_{p-1}$ such that $(w_{\sigma(0)}, w_{\sigma(1)}, \ldots, w_{\sigma(\ell-1)}) = (w_{\ell-n}, \ldots, w_{\ell-1}, w_0, \ldots, w_{\ell-n-1})$.

Proof. It is obvious that σ is a permutation corresponding to this shift, sets $S_0, \ldots, S_{p-1}$ form a partition of S. We have to prove that for $i = 0, \ldots, p - 1$:

(1) $\vec{\sigma}(S_i) = S_i$, where $\vec{\sigma}(A) = \{y: \exists x \in A(\sigma(x) = y)\}$;
(2) if $\emptyset \subsetneq Q \subseteq S_i$ and $\sigma(Q) = Q$, then $Q = S_i$.

To prove (1) it is enough to notice that for any $x \in S$ $\mathrm{mod}(x, p) = \mathrm{mod}(x + n, p)$, which implies $\mathrm{mod}(x, p) = \mathrm{mod}(\sigma(x), p)$ since $\sigma(x) = \mathrm{mod}(x + n, \ell)$ and $\mathrm{mod}(\ell, p) = 0$. To prove (2) it is enough to show that Q has $\ell/p = |S_i|$ different elements. Let $x \in Q$, then $\sigma^j(x) \in Q$ for any integer $j \geqslant 0$ and $\sigma^j(x) = \mathrm{mod}(x + jn, \ell)$ ($\sigma^0(x) = x$). Observe that $\sigma^j(x) = \sigma^k(x) \Leftrightarrow \exists_s(k - j = s\ell/n) \Leftrightarrow \exists_r(k - j = r\ell/p)$ because it holds that sp/n is an integer since $\gcd(\ell, n) = p$. Thus for $k, j \in \{0, 1, \ldots, \ell/p - 1\}$ and $j \neq k$ we have $\sigma^j(x) \neq \sigma^k(x)$ and this means there are ℓ/p different elements in Q.

Using as the base Theorem 2.1 we construct the procedure CHANGE which performs the exchange of two vectors u and v. Let us denote $w = (w_0, w_1, ..., w_{\ell-1}) = uv$, $m = |u|$, $n = |v|$, $\ell = m + n$ and $r = \gcd(\ell, n) = \gcd(m, n)$.

Algorithm CHANGE(u, v)

```
comment A[0 : m − 1] contains vector u and
A[m : ℓ − 1] contains vector v;
for i := 0 step 1 until p − 1 do
   begin a := A[i]; k := i; j := σ(i);
   comment the cycle S_i is performed;
   repeat A[k] := A[j]; k := j; j := σ(j)
      until j = i; A[k] := a
   end;
```

Let us notice that this procedure can be easily modified in the case when vector uv is in an array $A[i : i + \ell]$.

Theorem 2.2. Procedure CHANGE works in place and requires $m + n + \gcd(m, n)$ assignments.

Proof. Procedure CHANGE performs $|S_i| + 1 = (m + n)/p + 1$ assignments for each $i = 0, 1, ..., p - 1$. Thus the algorithm performs $p((m + n)/p + 1) = m + n + \gcd(m, n)$ assignments and uses only one additional record.

Lemma 2.1. Each algorithm performing a permutation f in place needs at least $N + p$ assignments, where f has p cycles $C_1, ..., C_p$ such that $|C_i| \geq 2$, and $N = \sum_{i=1}^{p} |C_i|$.

Proof. Since each element must be moved from its initial position in some cycle C_i to its final position (which lies in the same cycle), we need at least N assignments. For each cycle C_i we consider a moment when for the first time we place an element on its final position in this cycle. The element which had occupied this position previously must have been stored to avoid destroying it, but not on its final position (which lies in the same cycle). So for all cycles at least p additional assignments are required.

Theorem 2.3. Algorithm CHANGE is optimal with respect to the number of assignments.

Proof. Follows immediately from Lemma 2.1 and Theorem 2.2.

3. A fast merging algorithm

For any ordered vectors u and v let us distinguish parts u_1, u_2, v_1, v_2 and an element x such that:

(a) if $|u| \leq |v|$, then
(1) x is the middle element of u, i.e. $u = u_1 x u_2$ and $|u_1| = \lfloor |u|/2 \rfloor$, $|u_2| = \lfloor (|u| - 1)/2 \rfloor$,
(2) v_1 consists of all elements from v less than x, and v_2 of all elements from v not less than x;

(b) if $|u| > |v|$, then
(3) x is the middle element of v, i.e. $v = v_1 x v_2$ and $|v_2| = \lfloor |v|/2 \rfloor$, $|v_1| = \lfloor (|v| - 1)/2 \rfloor$;
(4) u_1 consists of all elements from u not greater than x and u_2 of all elements from u greater than x.

It is important for the stability that in point (2) there is a strong inequality to distinguish v_1, and in point (4) a weak inequality to distinguish u_1.

Notice that having defined parts u_1, u_2, v_1, v_2 and the element x it is enough to merge u_1 with v_1 and u_2 with v_2 (x is on its final position). So on the base of this idea we construct the recursive algorithm RECMERGE merging two ordered vectors.

Algorithm RECMERGE(u, v)

```
comment u is in A[i : j − 1] and v in A[j : k];
if |u| ≠ 0 and |v| ≠ 0 then
   begin
   if |u| ≤ |v| then
      begin p := (i + j − 1)/2;
         FIND(p + 1, j, k, ℓ);
      comment u_1 is in A[i : p], x is A[p + 1],
         u_2 in A[p + 2 : j],
      and v_1 is in A[j : ℓ], v_2 in A[ℓ + 1 : k];
      CHANGE(xu_2, v_1)
      end
   else
      begin p := (k + j − 1)/2;
         RFIND(p + 1, i, j, ℓ);
      comment v_1 is in A[j : p], x is A[p + 1],
         v_2 in A[p + 2 : k],
      and u_1 is in A[i : ℓ], u_2 in A[ℓ + 1 : j];
      CHANGE(u_2, v_1 x)
      end; RECMERGE(u_1, v_1); RECMERGE(u_2, v_2)
   end;
```

Table 1
Comparison of algorithms for merging two ordered vectors of length m and n (m ⩽ n)

	Algorithm	Space	Assignments	Comparisons
1	With extra vector [2]	O(m)	O(m + n)	O(m + n)
2	Links (lists) [2]	O(m + n)	O(1)	O(m + n)
3	BLOCKMERGE [4]	O(1)	$O(m^2 + n)$	O(m + n)
4	Balanced trees [1]	O(m + n)	O(1)	O(m log(n/m + 1))
5	Hwang and Lin [2]	O(1)	$O(m + n^2)$	O(m log(n/m + 1))
6	RECMERGE	O(log m)	O((m + n) log m)	O(m log(n/m + 1))
7	Trabb Pardo [4]	O(1)	O(m + n)	O(m + n)
	Optimal (unknown)	O(1)	O(m + n)	O(m log(n/m + 1))

Here, FIND(i, j, k, ℓ) is a binary search procedure finding partition v_1 and v_2 of v satisfying (2) and performing $\lfloor \log |v| \rfloor + 1$ [1] comparisons, and RFIND(i, j, k, ℓ) is similar to FIND satisfying (4).

Let us denote m = min(|u|, |v|), n = max(|u|, |v|), $k = \lfloor \log m \rfloor$ and let m_j^i and n_j^i denote the minimum and maximum of lengths of vectors merging on the i^{th} recursion level for i = 0, 1, ..., k and j = 1, 2, 3, ..., 2^i (initially $n_1^0 = n$ and $m_1^0 = m$), some of n_j^i and m_j^i can be equal to zero. It is obvious that, on each level i = 0, 1, ..., k, $\Sigma_{j=1}^{2^i}(m_j^i + n_j^i) \leqslant m + n$.

Theorem 3.1. The number of assignments in procedure RECMERGE does not exceed (m + n) log m + O(m + n) and the algorithm needs O(log m) space.

Proof. Since on each level i = 0, ..., k of recursion disjoint parts of u are merged with disjoint parts of v, all calls of procedure CHANGE perform at most $\Sigma_{j=1}^{2^i}((m_j^i + 1)/2 + n_j^i + \gcd((m_j^i + 1)/2, n_j^i)) \leqslant m + n + 2^i$ assignments. Thus for all levels there are at most $(m + n)(k + 1) + \Sigma_{i=0}^{k} 2^i \leqslant (m + n) \log m + 3m + n$ assignments. O(log m) memory space is needed to implement recursion.

Lemma 3.1. If $k = \Sigma_{j=1}^{2^i} k_j$ for any $k_j > 0$ and integer $i \geqslant 0$, then $\Sigma_{j=1}^{2^i} \log k_j \leqslant 2^i \log(k/2^i)$.

Proof. It is obvious since function log x is concave.

Theorem 3.2. Procedure RECMERGE performs O(m log(n/m + 1)) comparisons.

[1] In this chapter log x denotes $\log_2 x$.

Proof. The number of comparisons for each binary search is equal to $\lfloor \log n_j^i \rfloor + 1$ for i = 1, ..., k and j = 1, 2, 3, ..., 2^i for not null vectors and is equal to zero otherwise, therefore in neither case it exceeds $\Sigma_{j=1}^{2^i}(\log(n_j^i + 0.5) + 1)$ on the i^{th} recursion level. Since $n_j^i + 0.5 > 0$ and $\Sigma_{j=1}^{2^i}(n_j^i + 0.5) \leqslant m + n + 2^{i-1} \leqslant 3m/2 + n$, by Lemma 3.1, algorithm RECMERGE performs at most $2^i + 2^i \log((3m/2 + n)/2^i)$ comparisons on each recursion level. For all levels, the number of comparisons is less than $\Sigma_{i=0}^{k}(2^i(1 + \log(3m/2 + n)) - i2^i)$. Since $\Sigma_{i=0}^{k} i2^i = (k - 1)2^{k+1} + 2$, algorithm RECMERGE carries out at most $(\log(3m/2 + n) + 1)2^{k+1} - (k - 1)2^{k+1} \leqslant 2m(\log(3m/2 + n) - \log m + 3) \leqslant 2m(\log((3m + 2n)/m) + 2) =$ O(m log(n/m + 1)) comparisons.

Corollary. The application of procedure RECMERGE to sort a vector of length N requires $O(N(\log N)^2)$ assignments, O(N log N) comparisons and O(log N) space, since when m = n the merging algorithm performs O(m) comparisons and O(m log m) assignments.

There is a certain trade-off in the sorting and merging problem between three values: space, number of comparisons, number of assignments. Table 1 compares different merging algorithms.

4. Practical results

We have implemented two emerging algorithms: Trabb Pardo's [4] and RECMERGE. Table 2 compares their costs for sequences of randomly generated integers.

Table 2
Comparison of stable in minimal space merging algorithms (m, n are lengths of merged vectors)

m	n	Execution time in ms	
		Trabb Pardo	RECMERGE
500	500	733	557
	1000	1096	818
	1500	1418	1057
	2000	1615	1253
	2500	2102	1448
1000	1000	1499	1232
	1500	1719	1559
	2000	2234	1794
	2500	2542	2048
	3000	2872	2271
1500	1500	2762	1976
	2000	2576	1967
	2500	2989	2529
	3000	3279	2815
	3500	3608	3052
2000	2000	2996	2621
	2500	3397	2942
	3000	3698	3243

Acknowledgment

We wish to thank Dr. L. Banachowski for his initial suggestions and constructive comments and Prof. W.M. Turski for his positive suggestions, critique of the style and presentation of this paper.

References

[1] M. Brown and R. Tarjan, A fast merging algorithm, J. ACM 2 (1979).
[2] D.E. Knuth, The Art of Computer Programming, Vol. 3: Sorting and Searching (Addison-Wesley, Reading, MA, 1973) ch. 5.
[3] J. van Leeuwen, The Complexity of Data Organisation, Mathematical Centre Tracts 81 (Mathematical Centre, Amsterdam, 1976).
[4] L. Trabb Pardo, Stable sorting and merging with optimal space and time, SIAM J. Comput. (1977).