Computers and Mathematics with Applications 60 (2010) 2200–2203

Contents lists available at ScienceDirect

Computers and Mathematics with Applications

journal homepage: www.elsevier.com/locate/camwa

On computing an optimal permutation of ranks for multiselection

M.H. Alsuwaiyel

Department of Information and Computer Science, King Fahd University of Petroleum & Minerals, Dhahran 31261, Saudi Arabia

ARTICLE INFO

Article history:
Received 28 October 2009
Received in revised form 22 July 2010
Accepted 22 July 2010

Keywords:
Multiselection
Selection
Algorithms
Dynamic programming

ABSTRACT

Given a set of n elements, and a sorted sequence $K = k_1, k_2, \ldots, k_r$ of positive integers between 1 and n, it is required to find the k_ith smallest element for all values of i, $1 \leq i \leq r$. We present a dynamic programming algorithm for computing an optimal permutation of the input ranks that results in the least number of comparisons when used as a preprocessing step with any algorithm that uses repetitive calls to an algorithm for selection. The running time of the proposed algorithm is $O(r^3)$.

1. Introduction

Let S be a set of n numbers, and $K = k_1, k_2, \ldots, k_r$ be a sorted sequence of positive integers between 1 and n. The *multiselection* problem is to find the k_ith smallest element, denoted by a_i, for all values of i, $1 \leq i \leq r$. It has important applications in order statistics and set theory [1,2]. It seems that the algorithm in [1], which we will refer to as MSELECT, is the only existing (sequential) deterministic algorithm. The algorithm is straightforward: Find the kth smallest element a_k, where $k = \lceil r/2 \rceil$, and partition the set of remaining elements $S - \{a_k\}$ into two subsets S_1 and S_2 of elements smaller and larger than a_k, respectively. Next, make two recursive calls, one with S_1 and $K_1 = k_1, k_2, \ldots, k_{\lceil r/2 \rceil - 1}$, and the other with S_2 and $K_2 = k_{\lceil r/2 \rceil + 1} - k, k_{\lceil r/2 \rceil + 2} - k, \ldots, k_r - k$. The algorithm is shown in Fig. 1. Here, Algorithm SELECT is the well-known classical deterministic linear time algorithm [3]. Recently, a randomized algorithm that runs in time $O(n \log r)$ with probability $1 - O(n^{-1})$ appeared in [4]. It seems that the first parallelization of the multiselection problem was that of Shen [5], in which he presented an optimal parallel algorithm that runs in time $O(n^{\epsilon} \log r)$ on the EREW PRAM with $n^{1-\epsilon}$ processors, $0 < \epsilon < 1$. More on the work of Shen on parallel algorithms for the multiselection problem on interconnection networks can be found in [6–9].

The way by which Algorithm MSELECT processes the set of ranks is input-insensitive in the sense that the order in which processing ranks takes place is fixed regardless of their values or the size of S. Consider, for example, the instance in which $K = 1, 2, 5, 10$ and $n = |S| = 12$. The algorithm will find a_2, a_1, a_5 and a_{10}, in this order. Clearly, it is more efficient to start by finding a_5, and then proceed to find the elements corresponding to the remaining ranks: 1, 2, 10. This is because when finding a_1 and a_2, all elements of rank 5 or more will not be processed by Algorithm SELECT. Now, suppose we change the size of S to $n \geq 20$. Clearly, processing rank 10 first will force the algorithm to discard all elements greater than or equal to a_{10}, which results in reducing the number of redundant element comparisons significantly.

From the simple example above, it is eminent that a better ordering of the input ranks is desirable, which would render the algorithm sensitive to changes in the set of ranks and/or the size of S. This should cause the algorithm to avoid performing unnecessary redundant comparisons. In the next section, an algorithm for computing an optimal permutation of ranks is developed. It finds an optimal arrangement of ranks that minimizes the total number of comparisons. The algorithm presented uses the dynamic programming paradigm, and it turns out that it is similar to that for matrix chain

E-mail address: suwaiyel@kfupm.edu.sa.

0898-1221/$ – see front matter
doi:10.1016/j.camwa.2010.07.040

Algorithm MSELECT (S, K)

1. $k \leftarrow k_{\lceil r/2 \rceil}$
2. $a_k \leftarrow \text{SELECT}(S, k)$; **output** a_k
3. $S_1 \leftarrow \{a \in S \mid a < a_k\}$; $S_2 \leftarrow \{a \in S \mid a > a_k\}$
4. $K_1 \leftarrow \langle k_j \mid 1 \le j < \lceil r/2 \rceil \rangle$; $K_2 \leftarrow \langle k_j - k \mid \lceil r/2 \rceil < j \le r \rangle$
5. $\text{MSELECT}(S_1, K_1)$; $\text{MSELECT}(S_2, K_2)$.

Fig. 1. The original multiselection algorithm.

multiplication: Given matrices $M_1, M_2, \ldots, M_r$, find the order by which these matrices are to be multiplied in order to minimize the total number of scalar multiplications.

2. Computing an optimal permutation of ranks

Algorithm MSELECT makes use of the input sequence $k_1, k_2, \ldots, k_r$, where $k_1 < k_2 < \cdots < k_r$, for processing the set of ranks in the order $k_{i_1}, k_{i_2}, \ldots, k_{i_r}$, where $i_1 = \lceil |K|/2 \rceil = \lceil r/2 \rceil$, $i_2 = \lceil |K_1|/2 \rceil$, etc. In this section, an algorithm is presented for deriving a permutation π^*, which should result in the least total number of element comparisons performed by Algorithm MSELECT. In what follows, the names permutation, arrangement and ordering will be used interchangeably.

Algorithm MSELECT makes repetitive calls to Algorithm SELECT with parameters

$$(S_{i_1}, k_{i_1}), (S_{i_2}, k_{i_2}), \ldots, (S_{i_r}, k_{i_r}), \tag{1}$$

in some order, where $S_{i_1} = S$, $k_{i_1} = k_{\lceil r/2 \rceil}$, and for $2 \le j \le r$, $S_{i_j} \subseteq S$. As Algorithm SELECT is a deterministic linear time algorithm, given any arbitrary input (S', k), its running time is at most $c|S'|$ for some positive constant $c > 0$ that is dependent on the algorithm. Hence, the total running time of the calls in (1) is at most $\sum_{j=1}^{r} c\, n_{i_j}$, where $n_{i_j} = |S_{i_j}|$, $1 \le j \le r$. It follows that π^* is an optimal permutation of ranks if it minimizes the quantity

$$\sum_{j=1}^{r} n_{i_j}. \tag{2}$$

Hence, our task now reduces to finding a permutation π^* that minimizes the value of the sum in (2).

For convenience and brevity, we extend the sequence K of ranks by adding two dummy ranks, namely $k_0 = 0$ and $k_{r+1} = n+1$. So, let $J = k_0, k_1, \ldots, k_{r+1}$ be the extended set of ranks. Given two ranks k_i and k_j, where $0 \le i < j \le r+1$, we define the (open) *interval* $\mathbb{I}_{i,j}, j > i$, bounded by the two ranks k_i and k_j, as the *set* of integers (strictly) greater than k_i and (strictly) less than k_j. Note that k_i may be 0 and k_j may be $n+1$. More precisely, the interval $\mathbb{I}_{i,j}$ associated with the two ranks k_i, k_j, $0 \le i < j \le r+1$, is defined by

$$\mathbb{I}_{i,j} = \begin{cases} \phi & \text{if } j = i+1 \\ \{k_i + 1, k_i + 2, \ldots, k_j - 1\} & \text{if } j > i+1. \end{cases}$$

That is, for $j > i+1$, $\mathbb{I}_{i,j}$ consists of all integers between k_i and k_j exclusive of k_i and k_j. The *length* of interval $\mathbb{I}_{i,j}$ is thus defined as

$$L(\mathbb{I}_{i,j}) = \begin{cases} 0 & \text{if } j = i+1 \\ k_j - k_i - 1 & \text{if } j > i+1, \end{cases}$$

and its *cost* is defined recursively as

$$C(\mathbb{I}_{i,j}) = \begin{cases} 0 & \text{if } j = i+1 \\ \min\limits_{1<l<j} \{L(\mathbb{I}_{i,j}) + C(\mathbb{I}_{i,l}) + C(\mathbb{I}_{l,j})\} & \text{if } j > i+1. \end{cases} \tag{3}$$

2.1. The dynamic programming algorithm

The detailed algorithm is shown in Fig. 2. Here we use the cost matrix M, whose row-indices range from 0 to r and column-indices from 1 to $r+1$, with the property that $M[i, j]$ holds the cost of interval $\mathbb{I}_{i,j}$. That is, $M[i, j] = C(\mathbb{I}_{i,j})$. Hence, noting that for $0 < l < r+1$, $J_l = k_l$, $M[i, j]$ can be expressed as

$$M[i, j] = \begin{cases} 0 & \text{if } j = i+1 \\ \min\limits_{i<l<j} \{J[j] - J[i] - 1 + M[i, l] + M[l, j]\} & \text{if } j > i+1. \end{cases} \tag{4}$$

In particular,

$$M[0, r+1] = \min_{1 \le l \le r} \{n + M[0, l] + M[l, r+1]\} \tag{5}$$

is the minimum total cost corresponding to the quantity expressed by Eq. (2).

Let diagonal d refer to that one in which $j - i - 1 = d$, so the main diagonal is numbered 0 (recall that rows are numbered 0 to r and columns are numbered 1 to $r+1$). The algorithm first initializes the main diagonal to 0's, and proceeds by filling diagonals $1, 2, \ldots, r$ using Eq. (4). This is done in Steps 12–19 of the algorithm.

```
Algorithm OPT-PERM (K, n)
 1. J ← k_0, k_1, ..., k_{r+1}
 2. comment: Intialize diagonal 0. M[0..r, 1..r+1] is the cost matrix.
 3. for i ← 1 to r + 1
 4.     M[i − 1, i] ← 0
 5. comment: for each diagonal d = 1 to d = r.
 6. for d ← 1 to r
 7.     comment: for each one of the r + 1 − d entries in diagonal d.
 8.     for l ← 1 to r + 1 − d
 9.         i ← l − 1
10.         j ← (i + 1) + d
11.         c ← ∞
12.         for l ← i + 1 to j − 1
13.             tmp ← M[i, l] + M[l, j]
14.             if tmp < c then
15.                 c ← tmp
16.                 p ← l
17.             endif
18.         endfor
19.         M[i, j] ← c + (J[j] − J[i] − 1)
20.         rank[i, j] = p
21.         leftchild[i, j] = (i, p)
22.         rightchild[i, j] = (p, j)
23.     endfor
24. endfor
25. comment: Compute permutation π* using preorder traversal of the binary tree.
26. St ← (0, r + 1);  π* ← φ
27. while St is not empty
28.     (i,j) ← POP(St)
29.     append rank[i, j] to π*
30.     PUSH(rightchild[i, j])
31.     PUSH(leftchild[i, j])
32. endwhile
33. output π*
```

Fig. 2. The dynamic programming algorithm.

The ordering of ranks is defined by a binary tree T that is constructed by tracing back the computation of M as follows. The root of T is $J[p] = k_p$, where p, $1 \le p \le r$, is that value of l, which minimizes $M[0, r+1]$ in Eq. (5). The left and right subtrees of T are defined recursively by the two values of l that minimized $M[0, p]$ and $M[p, r+1]$ in Eq. (4). The binary tree is constructed in Steps 20–22 of the algorithm. Finally, a preorder traversal of T using a stack in Steps 26–23 gives the optimal permutation of ranks.

Consider, for example, the instance discussed in the introduction, in which $K = 1, 2, 5, 10$ and $n = 12$. First, K is extended to $J = 0, 1, 2, 5, 10, 13$. At the end of the algorithm,

$$M = \begin{pmatrix} 0 & 1 & 5 & 14 & 24 \\ 0 & \mathbf{0} & \mathbf{3} & 11 & 21 \\ 0 & 0 & 0 & \mathbf{7} & 17 \\ 0 & 0 & 0 & \mathbf{0} & 7 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix}.$$

As an illustration, we will compute $M[1, 4]$ (The four values in M used to compute $M[1, 4] = 11$ are shown in bold). Since $J[4] - J[1] - 1 = 10 - 1 - 1 = 8$, substituting in Eq. (4) yields

$$\begin{aligned} M[1, 4] &= \min_{1<l<4} \{(J[4] - J[1] - 1) + M[1, l] + M[l, 4]\} \\ &= \min\{8 + M[1, 2] + M[2, 4],\ 8 + M[1, 3] + M[3, 4]\} \\ &= \min\{8 + 0 + 7,\ 8 + 3 + 0\} \\ &= 11. \end{aligned}$$

The matrix below, which we will call Q, shows the values of *rank*, *leftchild* and *rightchild* at the end of the algorithm. As in matrix M, rows are numbered 0 to r and columns are numbered 1 to $r+1$. As an example, the entry $Q[0, 5] = 5, (0, 3), (3, 5)$ means that the root is rank 5, its left child is $Q[0, 3]$ (rank 2) and its right child is $Q[3, 5]$(rank 10)

$$\begin{pmatrix} 0, (), () & 1, (0, 1), (1, 2) & 2, (0, 2), (2, 3) & 5, (0, 3), (3, 4) & 5, (0, 3), (3, 5) \\ 0, (), () & 0, (), () & 2, (1, 2), (2, 3) & 5, (1, 3), (3, 4) & 5, (1, 3), (3, 5) \\ 0, (), () & 0, (), () & 0, (), () & 5, (2, 3), (3, 4) & 5, (2, 3), (3, 5) \\ 0, (), () & 0, (), () & 0, (), () & 0, (), () & 10, (3, 4), (4, 5) \\ 0, (), () & 0, (), () & 0, (), () & 0, (), () & 0, (), () \end{pmatrix}.$$

The binary tree T associated with the new permutation π^* is shown in Fig. 3(a). A preorder traversal of T gives $\pi^* = 5, 2, 1, 10$, which is an optimal ordering. Fig. 3(b) shows the output binary tree when the size of S is changed to $n \ge 20$.

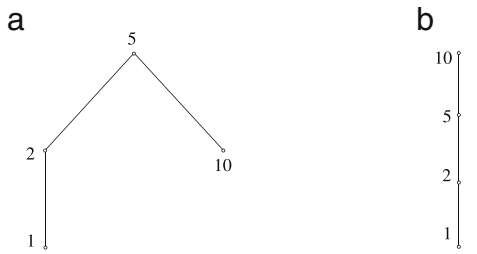

Fig. 3. Binary tree representation of the final permutation π^*: (a) $n = 12$. (b) $n = 20$.

Now it remains to show how to use the new permutation in computing the elements in S with the ranks in K rearranged (if necessary) as in π^*. For this, we only need to modify Algorithm MSELECT so that it processes the ranks in the right order. When constructing K_1 and K_2, it is important that their relative order in π^* be preserved. In other words, if k_i and k_j are both in K_1 (K_2) and k_i precedes k_j in K, then k_i also precedes k_j in K_1 (K_2). Hence, Steps 1 and 4 of Algorithm MSELECT are simply changed to

1. $k \leftarrow K[1]$
4. $K_1 \leftarrow \langle k_j | k_j \in K \text{ and } k_j < k\rangle$; $K_2 \leftarrow \langle k_j - k | k_j \in K \text{ and } k_j > k\rangle$.

Another possibility is to change the form in which the new permutation is generated. For instance, Algorithm OPT-PERM can be modified so that it outputs the sequence in the form of a tree: (left subtree, rank, right subtree). In the above example, the output may look like

$$(\ (\ (\ 1\ )\ 2\ (\ )\ )\ 5\ (\ (\ )\ 10\ (\ )\ )\ )\,,$$

which is the tree shown in Fig. 3(a). A preorder traversal is then used; $k = 5$, K_1 corresponds to the left subtree, and K_2 corresponds to the right subtree.

As a consequence of the principle of optimality in dynamic programming, it is noteworthy that if the ranks in K have optimal ordering, then the orderings in both K_1 and K_2 are also optimal.

2.2. Correctness and complexity

The proof of algorithm's correctness is embedded in its description. Eq. (4) guarantees that the output permutation π^* is optimal. Clearly, the amount of work done by the algorithm is in the order of $O(r^3)$, and the space required is in the order of $O(r^2)$. The algorithm is to be used as a preprocessing step for Algorithm MSELECT (or any other multiselection algorithm that works by using an algorithm for selection for all ranks). Since the running time of Algorithm MSELECT is $O(n \log r)$, the overall running time becomes $O(r^3 + n \log r)$, which is $O(n \log r)$ whenever $r = O((n \log r)^{1/3})$. In particular, when $r = O(\log^k n)$ for some $k > 0$, i.e., polylogarithmic in n, then since $\log^k n = o(n)$, the running time of Algorithm MSELECT combined with Algorithm OPT-PERM is $O(\log^{3k} n + n \log \log^k n) = O(n \log \log n)$. If the input ranks are clustered in a relatively small interval, say $k_r - k_1 + 1 = O(n^\epsilon)$, $0 < \epsilon < 1$, then the running time should be in the order of $\Theta(n)$. This is because the algorithm will find the element(s) on the boundary of the interval in the first few iterations, after which the size of S is reduced to $O(n^\epsilon)$.

References

[1] M.L. Fredman, T.H. Spencer, Refined complexity analysis for heap operations, Journal of Computer and System Sciences (1987) 269–284.
[2] D.G. Kirkpatrick, A unified lower bound for selection and set partitioning problems, Journal of the Association for Computing Machinery 28 (1981) 150–165.
[3] M. Blum, R.W. Floyd, V.R. Pratt, R.L. Rivest, R.E. Tarjan, Time bounds for selection, Journal of Computer and System Sciences 7 (1973) 448–461.
[4] M.H. Alsuwaiyel, A random algorithm for multiselection, Journal of Discrete Mathematics and Applications 16 (2) (2006) 175–180. 6.
[5] H. Shen, Optimal parallel multiselection on EREW PRAM, Parallel Computing 23 (1997) 1987–1992.
[6] H. Shen, Efficient parallel multiselection on hypercubes, in: Proc. 1997 Intern. Symp. on Parallel Architectures, Algorithms and Networks, I-SPAN, IEEE CS Press, 1997, pp. 338–342.
[7] H. Shen, Optimal multiselection in hypercubes, Parallel Algorithms and Applications 14 (2000) 203–212.
[8] H. Shen, Y. Han, Y. Pan, D.J. Evans, Optimal parallel algorithms for multiselection on mesh-connected computers, International Journal of Computer Mathematics 80 (2) (2003) 165–179.
[9] H. Shen, F. Chin, Selection and multiselection on multi-dimensional meshes, in: Proc. of the International Conference on Parallel and Distributed Processing Techniques and Applications, 2002, pp. 899–906.