Available online at www.sciencedirect.com
ScienceDirect
Theoretical Computer Science 362 (2006) 162–170
Theoretical Computer Science
www.elsevier.com/locate/tcs

Improved algorithms for the k maximum-sums problems ☆

Chih-Huai Cheng[a], Kuan-Yu Chen[a], Wen-Chin Tien[a], Kun-Mao Chao[a,b,*]

[a] *Department of Computer Science and Information Engineering, National Taiwan University, Taipei, Taiwan 106, Taiwan*
[b] *Graduate Institute of Networking and Multimedia, National Taiwan University, Taipei, Taiwan 106, Taiwan*

Received 9 September 2005; received in revised form 29 May 2006; accepted 1 June 2006

Communicated by Ding-Zhu Du

Abstract

Given a sequence of n real numbers and an integer k, $1 \leqslant k \leqslant \frac{1}{2}n(n-1)$, the k maximum-sum segments problem is to locate the k segments whose sums are the k largest among all possible segment sums. Recently, Bengtsson and Chen gave an $O(\min\{k + n\log^2 n, n\sqrt{k}\})$-time algorithm for this problem. Bae and Takaoka later proposed a more efficient algorithm for small k. In this paper, we propose an $O(n + k\log(\min\{n, k\}))$-time algorithm for the same problem, which is superior to both of them when k is $o(n\log n)$. We also give the first optimal algorithm for delivering the k maximum-sum segments in non-decreasing order if $k \leqslant n$. Then we develop an $O(n^{2d-1} + k\log\min\{n, k\})$-time algorithm for the d-dimensional version of the problem, where $d > 1$ and each dimension, without loss of generality, is of the same size n. This improves the best previously known $O(n^{2d-1}C)$-time algorithm, also by Bengtsson and Chen, where $C = \min\{k + n\log^2 n, n\sqrt{k}\}$. It should be pointed out that, given a two-dimensional array of size $m \times n$, our algorithm for finding the k maximum-sum subarrays is the first one achieving cubic time provided that k is $O(m^2 n/\log n)$.
© 2006 Elsevier B.V. All rights reserved.

Keywords: Maximum-sum subsequence; Maximum-sum subarray; Sequence analysis

1. Introduction

The maximum-sum subarray problem was first surveyed by Bentley in his "Programming Pearls" column of CACM [5,6]. The one-dimensional case is also called the maximum-sum segment problem and is well known linear-time solvable using Kadane's algorithm [5]. In the two-dimensional case, the task is to find a subarray such that the sum of its elements is maximized. The maximum segment (subarray) problem is widely used in pattern recognition [12,17], image processing [11], biological sequence analysis [1,8,10,13,15,16,21], data mining [11], and many other applications.

Computing the k largest sums over all possible segments is a natural extension of the maximum-sum segment problem. This extension has been considered from two perspectives, one of which allows the segments to overlap, while the other disallows. Linear-time algorithms for finding all the non-overlapping maximal segments were given in [7,18]. In

☆ A preliminary version of this work appeared in Proceedings of the 16th International Symposium on Algorithms and Computation, Sanya, China, 2005.

* Corresponding author. Department of Computer Science and Information Engineering, National Taiwan University, Taipei, Taiwan 106, Taiwan. Tel.: +886 2 33664888; fax: +886 2 23628167.

E-mail address: kmchao@csie.ntu.edu.tw (K.-M. Chao).

0304-3975/$ - see front matter © 2006 Elsevier B.V. All rights reserved.
doi:10.1016/j.tcs.2006.06.007

this paper, we focus on finding the k maximum-sum segments whose overlapping is allowed. We will use the terms "k maximum segments" and "k maximum-sum segments" interchangeably. A naïve approach is to choose the k largest from the sums of all possible contiguous subsequences which requires $O(n^2)$ time. Bae and Takaoka [2] presented an $O(kn)$-time algorithm for the k maximum segment problem. An improvement to $O(\min\{k + n\log^2 n, n\sqrt{k}\})$ was given by Bengtsson and Chen [4]. Bae and Takaoka [3] later proposed a more efficient algorithm for small k. Their new algorithm runs in $O(n \log k + k^2)$, which is superior to Bengtsson and Chen's when k is $o(\sqrt{n}\log n)$. In this paper, we propose an $O(n + k \log(\min\{n, k\}))$-time algorithm which is superior to all the previous methods when k is $o(n \log n)$. It should be noted that recently Lin and Lee [14] gave a randomized $O(n \log n + k)$-time algorithm, which is a better choice when k is sufficiently large.

It is not difficult to see that a lower bound of the k maximum segment problem is $\Omega(n + k)$. There is still a gap between the trivial lower bound and our method. However, if the k maximum segments are requested in non-decreasing order, we give an $\Omega(n + k \log k)$-time lower bound for the $k \leqslant n$ case. A simple variant of our algorithm can deliver the k maximum segments in non-decreasing order in $O(n + k \log k)$ time, which is optimal if $k \leqslant n$.

To avoid misunderstanding, we will use the term "subarray" instead of "segment" in the multiple-dimensional cases. In the two-dimensional case, we are given an $m \times n$ array of real numbers. The fastest algorithm for the maximum subarray problem stayed at $O(m^2 n)$ time for a long period of time. In 1998, the first subcubic-time algorithm was proposed by Tamaki and Tokuyama [20]. The time complexity of the latest algorithm for the maximum subarray problem is $O(m^2 n(\log\log m/\log m)^{1/2})$, which was given by Takaoka [19]. Clearly, it is still close to $O(m^2 n)$.

Our goal for the two-dimensional case is to find the k maximum-sum subarrays in the array. Bae and Takaoka [2] gave an $O(m^2 nk)$-time algorithm for this problem. Bengtsson and Chen [4] presented an improved algorithm in $O(\min\{m^2 C, m^2 n^2\})$ time, where $C = \min\{k + n\log^2 n, n\sqrt{k}\}$. Recently, Bae and Takaoka [3] gave an $O(n^3 \log k + k^2 n^2)$ algorithm for the problem whose input is an $n \times n$ array. It runs in cubic time when k is $O(\sqrt{n})$. We propose an $O(m^2 n + k \log(\min\{n, k\}))$-time algorithm, which is superior to the previous results for every value of k. Note that our algorithm is the first cubic-time algorithm for the k maximum subarray problem when k is $O(m^2 n/\log n)$. For the d-dimensional case, the best previously known algorithm, by Bengtsson and Chen [4], runs in $O(n^{2d-1} C)$ time, where $C = \min\{k + n\log^2 n, n\sqrt{k}\}$. We propose an improved $O(n^{2d-1} + k \log \min\{n, k\})$-time algorithm.

The rest of the paper is organized as follows. Section 2 gives a formal definition of the k maximum segment (subarray) problem. In Section 3, we give the algorithm for the k maximum segment problem based on an iterative partial-table building approach and discuss the issue of reporting the k maximum segments in non-decreasing order. We then extend the results to the multiple-dimensional cases in Section 4. Finally, we close the paper by mentioning a few open problems.

2. Problem definitions and notations

Given a sequence of n real numbers $A[1 \ldots n]$, a segment is simply a contiguous subsequence of that sequence. Let P denote the prefix-sum array of A where $P[i] = a_1 + a_2 + \cdots + a_i$ for $i = 1, \ldots, n$. Let $A[i \ldots j]$ denote the segment $\langle a_i, a_{i+1}, \ldots, a_j\rangle$. Let $S(i, j)$ be the sum of $A[i \ldots j]$, i.e. $S(i, j) = a_i + a_{i+1} + \cdots + a_j$. It is easy to see that $S(i, j) = P[j] - P[i - 1]$. Let $[i, j]$ denote the set $\{i, i + 1, \ldots, j\}$ for $i \leqslant j$.

Problem 1. k maximum-sum segments.

Input: a sequence of n real numbers $A = \langle a_1, a_2, \ldots, a_n\rangle$ and an integer k, $1 \leqslant k \leqslant \frac{1}{2}n(n - 1)$.

Output: k maximum-sum segments such that the sums of these segments are the k largest among all possible segment sums.

We also consider the k maximum-sum problem in higher dimensions.

Problem 2. k d-dimensional maximum-sum subarrays.

Input: a d-dimensional array of real numbers and a positive integer k.

Output: k d-dimensional subarrays such that the sums of these subarrays are the k largest among all possible subarray sums.

In particular, the two-dimensional problem is sometimes referred to as the k maximum-sum subarray problem.

3. The k maximum-sum segment problem

Finding the k largest elements of a sequence is essential in the construction of our algorithm for the k maximum-sum segment problem. Thus, we first describe how to find the k largest elements in Lemma 1.

Lemma 1. *Given a sequence of numbers, the k maximum (or minimum) elements can be found in linear time.*

Proof. According to [9], the kth maximum element of a sequence can be found in linear time. Suppose a_i denotes the kth maximum element. To obtain the k maximum elements, we simply compare a_i with all the elements of the sequence. We first output those elements whose values are greater than a_i. Then, we append additional elements equal to a_i to the output so that k elements are yielded. □

A naïve quadratic-time solution to the k maximum segment problem is to build a table of size $n \times n$, storing all the possible segments. By Lemma 1, the k maximum segments can be retrieved from the table in $O(n^2)$ time. To speed up, we introduce a partial-table building method for the k maximum segment problem.

3.1. An iterative partial-table building approach

Instead of building the entire table at once, we adopt an iterative strategy, in the sense that in each iteration we build only a partial table. Before introducing our main algorithm, let us define some notations first.

Definition 1. Let $R_{i,j}$ denote the segment ending at index i such that $R_{i,j}$ is the jth largest among those segments that end at i. That is, $R_{i,j} = A[p \dots i]$ where $S(p, i)$ is the jth largest among $S(q, i)$ for all $q \in [1, i]$.

Definition 2. Let $T_{i,j}$ denote the set of segments $R_{i,1}, R_{i,2}, \dots, R_{i,j}$. In other words, $T_{i,j}$ contains all the j largest segments ending at index i.

The naïve approach, described at the beginning of this section, compares all the segments in $T_{1,n}, T_{2,n}, \dots, T_{n,n}$, each of which contains at most n segments. However, we know that if $R_{i,j}$ is not one of the k maximum segments of A, then neither are $R_{i,j+1}, R_{i,j+2}, \dots, R_{i,n}$ since each of them has a smaller sum than $R_{i,j}$ by definition. Furthermore, given an integer ℓ, there are only two possible cases for every index as follows:

1. For some index i, if not all the segments in $T_{i,\ell}$ belong to the k largest segments retrieved from $T_{1,\ell}, T_{2,\ell}, \dots, T_{n,\ell}$, then we need not consider segments $R_{i,\ell+1}, R_{i,\ell+2}, \dots, R_{i,n}$ anymore.
2. Conversely, for some index i', if all the segments in $T_{i',\ell}$ belong to the k largest segments retrieved from $T_{1,\ell}, T_{2,\ell}, \dots,$ $T_{n,\ell}$, then we need to consider $R_{i',\ell+1}, R_{i',\ell+2}, \dots, R_{i',n}$ since they are still candidates for the k maximum segments of A.

In conclusion, by comparing segments in $T_{1,\ell}, T_{2,\ell}, \dots, T_{n,\ell}$, we can bypass the indices in case 1, and only have to consider the indices in case 2 since they may contribute more than ℓ segments to the k maximum-sum segments. We call those indices in case 2 the "qualified right ends".

Algorithm 1 (KMaxSums).

1: $Q \leftarrow \{1, 2, \dots, n\}, n_s \leftarrow n, K \leftarrow \phi$;
2: **repeat**
3: find ℓ such that $n_s \times (\ell - 1) < 2k \leqslant n_s \times \ell$;
4: **if** $\ell > n$ **then**
5: $\ell \leftarrow n$;
6: **end if**
7: **for all** $i \in Q$ **do**
8: compute $T_{i,\ell}$;
9: **end for**
10: $K \leftarrow$ the k largest segments from $K \cup \bigcup_{i \in Q} T_{i,\ell}$;
11: $Q \leftarrow \{i \mid T_{i,\ell} \subseteq K \; \forall i \in Q\}$;
12: $n_s \leftarrow |Q|$;
13: **until** $n_s = 0$ or $\ell = n$
14: output the segments in K;

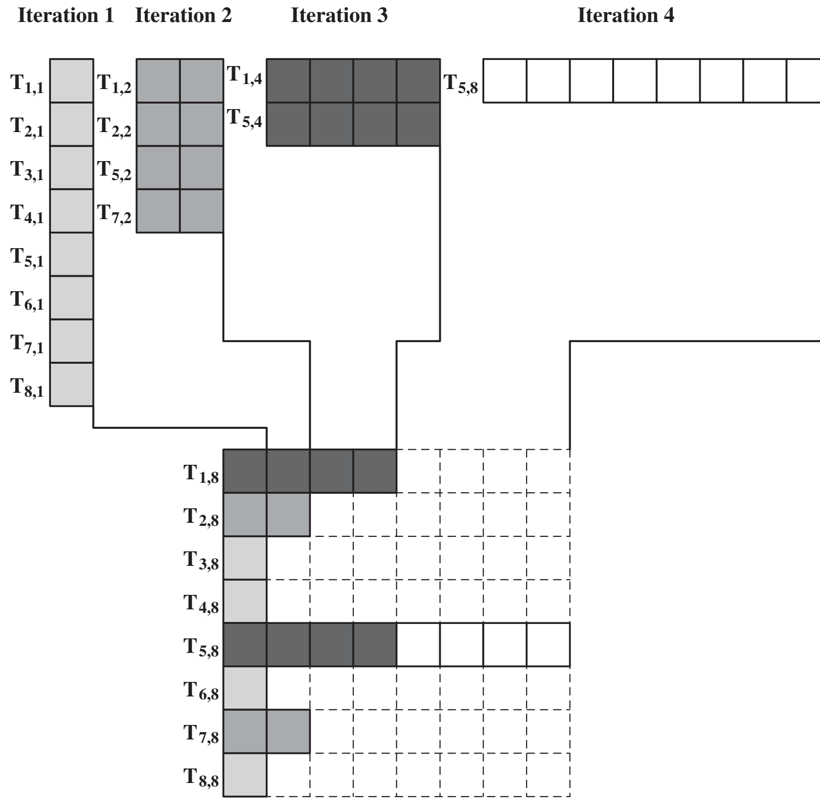

Fig. 1. An illustration of the partial-table building approach in our algorithm. As you can see, in each iteration a partial table of some fixed size is built. Only those rows with all their entries being selected will be extended in the next iteration. The algorithm terminates when the width of the partial table (solid lines) matches with the width of the virtual complete table (dashed lines).

The pseudo-code for the problem of k maximum-sum segments is given in Algorithm 1. Let Q denote the list of n_s distinct qualified right ends, which are initially the n positions of A and $Q[1], Q[2], \ldots, Q[n_s]$ refer to the n_s right ends in Q, respectively. We let K, initially empty, denote a list of candidate segments for the k maximum segments. The algorithm repeats the following procedure. In each iteration, we choose $\ell = 2\lceil k/n_s \rceil$ and then compute $T_{Q[1],\ell}, T_{Q[2],\ell}, \ldots, T_{Q[n_s],\ell}$ which contain around $2k$ segments. We next retrieve the k largest segments from the set of segments, obtained by incorporating segments in $T_{Q[1],\ell}, T_{Q[2],\ell}, \ldots, T_{Q[n_s],\ell}$ with the segments in K. It should be noted here that the k largest segments retrieved in this manner are not certainly the k maximum-sum segments of A because we only consider the ℓ largest segments ending at $Q[1], Q[2], \ldots, Q[n_s]$. Since only k largest segments are retrieved, there would be at most half qualified right ends left over for the next iteration. (Imagining k balls are thrown into an $n_s \times \ell$ table, $n_s \times \ell \cong 2k$, we know that there would be at most half rows full of balls.) Meanwhile, the value of ℓ will at least double in the next iteration. We set Q the qualified right ends, set n_s the size of Q, and then restart the procedure. The procedure will terminate either when the number of qualified right ends decreases to zero or ℓ increases to n. Fig. 1 illustrates the above idea.

Lemma 2. *Algorithm KMaxSums terminates in at most* $\log n + 1$ *iterations.*

Proof. Suppose algorithm KMaxSums terminates in the ith iteration, and the values of ℓ in each iteration are denoted by $\ell_1, \ell_2, \ldots, \ell_i$, respectively. Our goal is to prove that $i \leqslant \log n + 1$. For any two consecutive iterations j and $j+1$, suppose that n_s^j and n_s^{j+1} are the corresponding values of n_s in the jth iteration and the $(j+1)$th iteration. We have $n_s^j \times (\ell_j - 1) < 2k \leqslant n_s^j \times \ell_j$ and $n_s^{j+1} \times (\ell_{j+1} - 1) < 2k \leqslant n_s^{j+1} \times \ell_{j+1}$. By definition, n_s^{j+1} is the number of

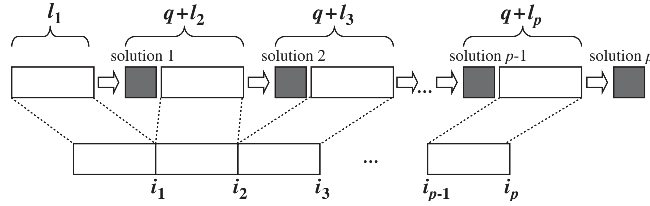

Fig. 2. An illustration of computing $T_{i_1,q}, T_{i_2,q}, \ldots, T_{i_p,q}$. To compute $T_{i_j,q}$ where $j \in [2, p]$, we find q minimum values from $q + \ell_j$ prefix sums.

qualified right ends with all their ℓ_j largest segments being retrieved, so it is clear that $n_s^{j+1} \leqslant \lfloor k/\ell_j \rfloor$. This yields $2k \leqslant n_s^{j+1} \times \ell_{j+1} \leqslant \lfloor k/\ell_j \rfloor \times \ell_{j+1} \leqslant k/\ell_j \times \ell_{j+1}$. We conclude that $\ell_{j+1} > 2\ell_j$.

Next we show by induction that $\ell_i \geqslant 2^{i-1}$. The basis holds since $\ell_1 \geqslant 2^0 = 1$. For any j we know $\ell_{j+1} \geqslant 2\ell_j$, so by inductive hypothesis, $\ell_j \geqslant 2^{j-1}$, we can deduce that $\ell_{j+1} \geqslant 2^j$. Thus, $\ell_i \geqslant 2^{i-1}$ by induction. Moreover, since ℓ_i is at most n, it follows that $n \geqslant \ell_i \geqslant 2^{i-1}$, which leads to $i \leqslant \log n + 1$. □

Lemma 3. *Given p distinct increasing indices* $i_1, i_2, \ldots, i_p$, $1 \leqslant p \leqslant n$, *and a positive integer* q, $1 \leqslant q \leqslant n$, *we can compute* $T_{i_1,q}, T_{i_2,q}, \ldots, T_{i_p,q}$ in $O(n + pq)$ *time.*

Proof. Basically, we adopt a dynamic approach (see Fig. 2 for an illustration). The input sequence A is partitioned into $p+1$ contiguous subsequences, $A[1 \ldots i_1]$, $A[i_1+1 \ldots i_2]$, $\ldots$, $A[i_{p-1}+1 \ldots i_p]$, and $A[i_p+1 \ldots n]$. Let ℓ_j denote the length of the jth contiguous subsequence, i.e. ℓ_j is the length of $A[i_{j-1}+1 \ldots i_j]$. Note that $\ell_1 + \ell_2 + \cdots + \ell_p$ is at most n.

To compute $T_{i_j,q}$, it suffices to find the q minimum values in the prefix-sum array $P[0 \ldots i_j]$ for all $j \in [1, p]$. So, in the first step we find the q minimum values among $P[0 \ldots i_1]$, which can be done in $O(\ell_1 + 1)$ time by Lemma 1. Let Q record these q minimum values. In the second step, the q minimum values among $P[0 \ldots i_2]$ is found by retrieving the q minimum values from Q and $P[i_1 + 1 \ldots i_2]$. This requires $O(\ell_2 + q)$ time by Lemma 1. Proceeding in this manner, we compute the $T_{i_j,q}$ for each index i_j in $O(\ell_j + q)$ time. Therefore, the total running time is $(\ell_1 + 1) + (\ell_2 + q) + (\ell_3 + q) + \cdots + (\ell_j + q) + \cdots + (\ell_p + q) = (\ell_1 + \ell_2 + \cdots + \ell_p) + (p-1)q + 1 = O(n + pq)$. □

Lemma 4. *Algorithm KMaxSums runs in* $O((n + k)\log n)$ *time.*

Proof. Since $n_s \times (\ell - 1) < 2k \leqslant n_s \times \ell \Rightarrow 2k \leqslant n_s \times \ell < 2k + n_s \leqslant 2k + n$, we can derive that $n_s \times \ell = O(n + k)$. By Lemma 3, the time required for computing the l largest segments ending at n_s qualified ends is $O(n + n_s \times \ell) = O(n + k)$. Retrieving k largest segments from $O(n_s \times \ell)$ segments takes only $O(k)$ time by Lemma 1. So, it takes $O(n+k)$ time in each iteration. Since there are at most $\log n+1$ iterations by Lemma 2, we conclude that the total time is $O((n + k)\log n)$. □

If $k \geqslant n$, we can write the time complexity of KMaxSums as $O(k \log n)$. If $k < n$, we can write the time complexity as $O(n \log n)$. In what follows, we show that in the $k < n$ case, we can further reduce the running time to $O(n+k \log k)$, which leads to Theorem 1.

Theorem 1. *The problem of finding the k maximum-sum segments can be solved in* $O(n + k \log(min\{n, k\}))$ *time.*

As we will see, the $O(n)$ term in $O(n + k \log k)$ comes from the time needed to compress the input sequence, and the $O(k \log k)$ term comes from the time of executing k iterations, each of which costs $O(\log k)$ time.

3.2. Improving on the time complexity in the $k < n$ case

The strategy is to compress the input sequence A into a sequence of size at most $2k$ by preprocessing A in $O(n)$ time. We can find k distinct positions containing all the right ends of the k maximum segments. Similarly, we find

k distinct positions containing all the left ends. To find the k maximum segments, we only have to consider these $2k$ positions.

Specifically, let $r_1, r_2, \ldots, r_k$ denote the right ends of the k largest segments retrieved from $R_{1,1}, R_{2,1}, \ldots, R_{n,1}$.

Lemma 5. *There exists an optimal solution S^* of the k maximum-sum segments such that all the right ends of S^* belong to $\{r_1, r_2, \ldots, r_k\}$.*

Proof. Suppose that $S_1, S_2, \ldots, S_k$ is an optimal solution of the k maximum segments. If all the end points of $S_1, S_2, \ldots, S_k$ belong to $\{r_1, r_2, \ldots, r_k\}$, the proof is done. Otherwise, without loss of generality, we assume $S_1, S_2, \ldots, S_p$ do not end at indices in $\{r_1, r_2, \ldots, r_k\}$. Let Δ denote the set of the right ends of $S_1, S_2, \ldots, S_k$. We find p distinct indices $r_1^*, r_2^*, \ldots, r_p^* \in \{r_1, r_2, \ldots, r_k\} - \Delta$. We replace S_i by segment $R_{r_i^*,1}$ without decreasing the sum for $1 \leqslant i \leqslant p$. Therefore, we obtain another optimal solution whose end points are all in $r_1, r_2, \ldots, r_k$ and the proof is completed. □

Similarly, we find the largest segments starting at each index of A. To do so, we scan the suffix sum array in the reverse order and keep the minimum value on the fly. Let us use a similar notation to refer to these segments, say $L_{1,1}, L_{2,1}, \ldots, L_{n,1}$. Let $l_1, l_2, \ldots, l_k$ denote the left ends of the k largest segments retrieved from $L_{1,1}, L_{2,1}, \ldots, L_{n,1}$. It can be shown in the same way that the k maximum segments of A must start at $l_1, l_2, \ldots, l_k$.

We construct a compressed sequence recording the prefix sums of $l_1, l_2, \ldots, l_k$ and $r_1, r_2, \ldots, r_k$. Algorithm KMaxSums takes this compressed sequence as input and runs in $O(k \log k)$ time. Since we use $O(n)$ time to compress A, the total time is $O(n + k \log k)$.

3.3. Finding k maximum-sum segments in order

The problem of k maximum segments has a trivial lower bound $\Omega(n + k)$. Note that the k maximum segments are not sorted. Lemma 6 states that if we want to output k maximum segments in non-decreasing order, the lower bound becomes $\Omega(n + k \log k)$ when k is no more than n.

Lemma 6. *When $k \leqslant n$, it requires $\Omega(n + k \log k)$ time to output the k maximum segments in non-decreasing order.*

Proof. If there exists an $o(n + k \log k)$-time algorithm for computing the k maximum segments in non-decreasing order, we show that sorting k random numbers can be done in $o(k \log k)$ time. Assume to the contrary that there exists an algorithm Λ computing the k sorted maximum segments in $o(n + k \log k)$ time. Given k random numbers, we obtain a new sequence of length $2k - 1$ as follows. For each two consecutive random numbers, we augment a negative number whose absolute value is larger than them. That way the k maximum segments in this new sequence are all atomic elements. The output of Λ on this new sequence is equivalent to the k sorted random numbers. The running time of Λ is $o((2k - 1) + (2k - 1)\log(2k - 1)) = o(k \log k)$. However, the lower bound of sorting k random numbers is well-known to be $\Omega(k \log k)$ which contradicts to our assumption. □

Corollary 2. *A simple variant of algorithm KMaxSums yields an optimal solution to the problem of finding the k sorted maximum segments when $k \leqslant n$.*

Proof. We first run algorithm *KMaxSums* to find the k maximum segments in $O(n + k \log k)$ time. We next sort the k maximum segments by sum, which requires $O(k \log k)$ time. □

It is not difficult to see that when $k > n$, algorithm *KMaxSums* also leads to an $O(k \log k)$-time solution to the problem of finding the k sorted maximum segments. However, we do not know if $\Omega(k \log k)$ is the actual lower bound of this sorted problem when $k > n$.

4. Multiple-dimensional cases

It is helpful to introduce the two-dimensional case before extending the results to the multiple-dimensional cases. Recall the definition of the k maximum-sum subarray problem in d dimensions. Its goal is to find k d-dimensional subarrays such that the sums of those subarrays are the k largest among all possible d-dimensional subarray sums.

4.1. Two-dimensional case

The input sequence is replaced by a two-dimensional array $X = [x_{ij}]_{1 \leqslant i \leqslant m, 1 \leqslant j \leqslant n}$. We define $X[p \dots q, r \dots s]$ as the subarray expanded by the four corners (p, r), (p, s), (q, r) and (q, s). The idea is to transform the input array $X[1 \dots m, 1 \dots n]$ into a pile of one-dimensional sequences. The k maximum subarray problem is then reduced to finding the k largest segments from these one-dimensional sequences. Using similar techniques presented in the previous sections, we can solve the k two-dimensional maximum subarrays problem. Let us show the transformation in detail. Given two indices i and j where $1 \leqslant i \leqslant j \leqslant m$, we convert the subarray $X[i \dots j, 1 \dots n]$ into a new sequence $X_{i,j}[1 \dots n]$ such that $X_{i,j}[q] = \sum_{p=i}^{j} x_{pq}$ for $q = 1, \dots, n$. Clearly, each segment $X_{i,j}[p \dots q]$ corresponds to the subarray $X[i \dots j, p \dots q]$, respectively. The maximum-sum subarray problem is equivalent to finding k maximum segments from the $O(m^2)$ converted sequences, $X_{i,j}$ for $1 \leqslant i \leqslant j \leqslant m$.

Given an integer ℓ, observe that each converted sequence's ℓ maximum segments are the ℓ local maxima with respect to the k maximum subarray problem. Obviously, the k maximum subarrays of X are the k global maxima. A naïve approach is to find every converted sequence's k local maxima, and the k maximum subarrays are the k largest among the $O(m^2 k)$ local maxima. Instead of finding $O(m^2 k)$ local maxima at once, we adopt the same trick to speed up the computation. That is, in each iteration we compute only $2k$ local maxima and eliminate half of them.

The pseudo-code for the k maximum-sum subarray problem is given in Algorithm 2. Let n_s denote the number of "qualified sequences", which will be defined later, and n_s is initialized as $O(m^2)$. In each iteration, we use algorithm KMaxSums to find ℓ maximum segments, $\ell = 2\lceil k/n_s \rceil$, from n_s converted sequences and then retrieve the k largest which are the candidates of the k maximum subarrays. Clearly, if the ℓth largest local maximum is not one of the candidates, neither is the $(\ell + 1)$th largest local maximum. We call the sequences whose ℓth largest local maximum belongs to the k candidates the "qualified sequences", and the rest the unqualified sequences. Only the qualified sequences need to be considered in the next iteration. The algorithm terminates when all the sequences are unqualified.

Algorithm 2 (KMaxSums2D).

1: $Q \leftarrow \{(i, j) \mid 1 \leqslant i \leqslant j \leqslant m\}$, $n_s \leftarrow m(m-1)/2$, $K \leftarrow \phi$;
2: Compute a new array, $Y = [y_{ij}]$ of order $m \times n$, where $y_{ij} = \sum_{h=0}^{i} x_{hj}$
3: **for** each i and j, $1 \leqslant i \leqslant j \leqslant m$ **do**
4:     Compute sequence $X_{i,j}[1 \dots n]$ such that $X_{i,j}[h] = y_{jh} - y_{ih}$ for $h = 1, 2, \dots, n$
5: **end for**
6: **repeat**
7:     find ℓ such that $n_s \times (\ell - 1) < 2k \leqslant n_s \times \ell$;
8:     **if** $\ell > n(n-1)/2$ **then**
9:         $\ell \leftarrow n(n-1)/2$;
10:     **end if**
11:     **for all** $(i, j) \in Q$ **do**
12:         $L_{i,j} \leftarrow \ell$ maximum segments of $X_{i,j}$ computed by KMaxSums;
13:     **end for**
14:     $K \leftarrow$ the k largest segments from $K \cup \bigcup_{(i,j) \in Q} L_{i,j}$;
15:     $Q \leftarrow \{(i, j) \mid L_{i,j} \subseteq K \ \forall \ (i, j) \in Q\}$;
16:     $n_s \leftarrow |Q|$;
17: **until** $n_s = 0$ or $\ell = n(n-1)/2$
18: output the segments in K;

Now we turn to the time complexity analysis. In lines 2–4, it takes $O(m^2 n)$ time to transform the two-dimensional input array into $O(m^2)$ one-dimensional sequences. Recall that at most half of the qualified sequences are left over after each round. We know that finding the ℓ maximum segments takes $O(n + \ell \log(\min\{n, \ell\}))$ time by Theorem 1. Below, we discuss it in two possible cases. When $k \leqslant n$, the number of qualified sequences is reduced to k in the second iteration. So, in the worst case the entire while-loop takes $O(m^2 \times (n + 1 \log 1) + k) + O(k \times (n + 2 \log 2) + k) + O(k/2 \times (n + 4 \log 4) + k) + \cdots + O(1 \times (n + k \log k) + k) = O(m^2 n + k \log k)$ time. When $k > n$, the total time becomes $O(m^2 \times (n + \ell \log \ell) + k) + O(m^2/2 \times (n + 2\ell \log 2\ell) + k) + O(m^2/4 \times (n + 4\ell \log 4\ell) + k) + \cdots +$

$O(m^2/\min\{m^2, k\} \times (n + (\ell \min\{m^2, k\}) \log(\ell \min\{m^2, k\}) + k) = O(m^2 n + k \log n)$ where $\ell = 2\lceil k/m^2 \rceil$. Therefore, we have the following theorem.

Theorem 3. *Algorithm KMaxSums2D finds the k maximum-sum subarrays in* $O(m^2 n + k \log(\min\{n, k\}))$ *time.*

4.2. Higher-dimensional cases

Without loss of generality, we assume each dimension is of equal size n. Given a d-dimensional array $X = [x_{i_1 i_2 \ldots i_d}]_{1 \leqslant i_1, i_2, \ldots, i_d \leqslant n}$, we wish to find k d-dimensional subarrays with maximum sums. Similar techniques in the two-dimensional case are used here. We transform the d-dimensional input array into sequences first, where each element of a converted sequence stores $(d-1)$-dimensional values in X. That is, the converted sequence is defined as follows

$$X_{i_2, j_2, i_3, j_3, \ldots, i_d, j_d}[q] = \sum_{p_2=i_2}^{j_2} \sum_{p_3=i_3}^{j_3} \ldots \sum_{p_d=i_d}^{j_d} x_{q p_2 p_3 \ldots p_d} \quad \forall 1 \leqslant q \leqslant n.$$

Because there are $O(n^2)$ combinations in every dimension, the number of the converted sequences is $O(n^{2d-2})$. A similar analysis to the two-dimensional case yields the following theorem.

Theorem 4. *The k d-dimensional maximum-sum subarrays can be found in* $O(n^{2d-1} + k \log \min\{n, k\})$ *time.*

5. Conclusions

We close this paper by mentioning a few open problems. First, is there an algorithm running in $o(k \log k)$ time for finding the k maximum segments in non-decreasing order when $k > n$? Second, it would be interesting to find a tight lower bound for the multiple-dimensional k maximum subarray problem.

Acknowledgements

Chih-Huai Cheng, Kuan-Yu Chen, Wen-Chin Tien and Kun-Mao Chao were supported in part by NSC Grants 92-2213-E-002-059 and 93-2213-E-002-029 from the National Science Council, Taiwan.

References

[1] L. Allison, Longest biased interval and longest non-negative sum interval, Bioinformatics 19 (2003) 1294–1295.
[2] S.E. Bae, T. Takaoka, Algorithms for the Problem of k Maximum Sums and a VLSI Algorithm for the k Maximum Subarrays Problem, in: Proc. Seventh Internat. Symp. Parallel Architectures, Algorithms and Networks, 2004, pp. 247–253.
[3] S.E. Bae, T. Takaoka, Improved Algorithms for the K-Maximum Subarray Problem for Small K, in: Proc. 11th Annual Internat. Computing and Combinatorics Conference, 2005, pp. 621–631.
[4] F. Bengtsson, J. Chen, Efficient algorithms for k maximum sums, in: Proc. 15th Internat. Symp. on Algorithms and Computation, Lecture Notes in Computer Science, Vol. 3341, 2004, pp. 137–148.
[5] J. Bentley, Programming pearls: algorithm design techniques, Commun. ACM (1984) 865–871.
[6] J. Bentley, Programming pearls: perspective on performance, Commun. ACM (1984) 1087–1092.
[7] K.-Y. Chen, K.-M Chao, On the range maximum-sum segment query problem, in: Proc. 15th Internat. Symp. Algorithms and Computation, Lecture Notes in Computer Science, Vol. 3341, 2004, pp. 294–305.
[8] K.-Y. Chen, K.-M. Chao, Optimal algorithms for locating the longest and shortest segments satisfying a sum or an average constraint, Inform. Process. Lett. 96 (2005) 197–201.
[9] T.H. Cormen, C.E. Leiserson, R.L. Rivest, C. Stein, Introduction to Algorithms, second ed., The MIT Press, Cambridge, 1999, pp. 185–196.
[10] T.-H. Fan, S. Lee, H.-I. Lu, T.-S. Tsou, T.-C. Wang, A. Yao, An optimal algorithm for maximum-sum segment and its application in bioinformatics, in: Proc. Eighth Internat. Conf. on Implementation and Application of Automata, Lecture Notes in Computer Science, Vol. 2759, 2003, pp. 251–257.
[11] T. Fukuda, Y. Morimoto, S. Morishita, T. Tokuyama, Data mining using two-dimensional optimized association rules: scheme, algorithms, and visualization, in: Proc. 1996 ACM SIGMOD Internat. Conf. on Management of Data, 1996, pp. 13–23.
[12] U. Grenander, Pattern Analysis, Springer, New York, 1978.
[13] X. Huang, An algorithm for identifying regions of a DNA sequence that satisfy a content requirement, Comput. Appl. Biosci. 10 (1994) 219–225.

[14] T.-C. Lin, D.T. Lee, Randomized algorithm for the sum selection problem, in: Proc. 16th Internat. Symp. on Algorithms and Computation, Lecture Notes in Computer Science, Vol. 3827, 2005, pp. 515–523.
[15] Y.-L. Lin, X. Huang, T. Jiang, K.-M. Chao, MAVG: locating non-overlapping maximum average segments in a given sequence, Bioinformatics 19 (2003) 151–152.
[16] Y.-L. Lin, T. Jiang, K.-M. Chao, Efficient algorithms for locating the length-constrained heaviest segments with applications to biomolecular sequence analysis, J. Comput. System Sci. 65 (2002) 570–586.
[17] K. Perumalla, N. Deo, Parallel algorithms for maximum subsequence and maximum subarray, Parallel Process. Lett. 5 (1995) 367–373.
[18] W.L. Ruzzo, M. Tompa, A linear time algorithm for finding all maximal scoring subsequences, in: Proc. Seventh Internat. Conf. on Intelligent Systems for Molecular Biology, 1999, pp. 234–241.
[19] T. Takaoka, Efficient Algorithms for the Maximum Subarray Problem by Distance Matrix Multiplication, Electronic Notes in Theoretical Computer Science, Vol. 61, 2002, pp. 1–10.
[20] T. Tamaki, T. Tokuyama, Algorithms for the maximum subarray problem based on matrix multiplication, in: Proc. Ninth Annu. ACM-SIAM Symp. on Discrete Algorithms, 1998, pp. 446–452.
[21] L. Wang, Y. Xu, SEGID: identifying interesting segments in (multiple) sequence alignments, Bioinformatics 19 (2003) 297–298.