A Sub-cubic Time Algorithm for the *k*-Maximum Subarray Problem

Sung Eun Bae and Tadao Takaoka

Department of Computer Science, University of Canterbury
Christchurch, New Zealand
E-mail {seb43, tad}@cosc.canterbury.ac.nz

Abstract. We design a faster algorithm for the k-maximum sub-array problem under the conventional RAM model, based on distance matrix multiplication (DMM). Specifically we achieve $O(n^3\sqrt{\log\log n/\log n} + k\log n)$ for a general problem where overlapping is allowed for solution arrays. This complexity is sub-cubic when $k = o(n^3/\log n)$. The best known complexities of this problem are $O(n^3 + k\log n)$, which is cubic when $k = O(n^3/\log n)$, and $O(kn^3\sqrt{\log\log n/\log n})$, which is sub-cubic when $k = o(\sqrt{\log n/\log\log n})$.

1 Introduction

The maximum subarray (MSA) problem is to compute a rectangular portion in a given two-dimensional (n, n)-array that maximizes the sum of array elements in it. If the array elements are all non-negative we have the trivial solution of the whole array. Thus we normally subtract the mean or median value from each array element. This problem has wide applications in graphics and data mining for marketing, as described in [4].

This problem was first introduced by Grenander and brought to computer science by Bentley [7] with an algorithm of $O(n^3)$. Tamaki and Tokuyama [22] obtained a sub-cubic algorithm based on distance matrix multiplication (DMM), by reducing the problem to DMM and showing that the time complexities of the two problems are of the same order. Takaoka [19] simplified the algorithm for implementation.

The k-maximum subarray (k-MSA) problem is to obtain the maximum sub-array, the second maximum subarray, ..., the k-th maximum subarray in sorted order for k up to $O(n^4)$. We can define two such problems. One is the general case where we allow overlapping portions, and the other is for disjoint portions. We consider the general problem in this paper. Let $M(n)$ be the time complexity for DMM for an (n, n)-matrix. We solve the problem in $O(M(n) + k\log n)$ time for the general problem with an (n, n)-array, where $M(n) = O(n^3\sqrt{\log\log n/\log n})$.

Preceding results for the one-dimensional problem are $O(kn)$ by Bae and Takaoka [1], $O(min(n\sqrt{k}, n\log^2 n))$ by Bengtsson and Chen [5], $O(n\log k)$ by Bae and Takaoka [2], $O(n + k\log n)$ by Bae [4], Cheng, et. al. [11], Bengtsson, et. al. [6], $O(n\log n + k)$ expected time by Lin, et. al. [17], and $O(n + k)$ by Brodal,

et. al. [8]. Obviously we can solve the two-dimensional problem by applying the one-dimensional algorithm to all $O(n^2)$ strips of the array, resulting in the time complexity multiplied by $O(n^2)$. For the algorithms specially designed for the two-dimensional case, we have $O(kn^3(\log\log n/\log n)^{1/2})$ by [2] and $O(n^3+k)$ by [8]. The last is for k maximum subarrays in unsorted order.

These results are mainly based on extension of optimal algorithms for the one-dimensional problem to the two-dimensional problem. Our results in this paper and [2] show an extension of an optimal algorithm in one dimension to two dimensions does not produce optimal solutions for the two-dimensional problem.

The best known results for the disjoint case are the straightforward $O(kM(n))$, which is sub-cubic for small k such as $k = o(\sqrt{\log n/\log\log n})$, where $M(n)$ is the time for DMM, and $O(n^3 + kn^2\log n)$ by Bae and Takaoka [3] for larger k. The problem here is to find the maximum, the second maximum, etc. from the remaining portion.

In the application of graphics, our problem is to find the brightest spot, second brightest spot, ..., k-th brightest spot. In the application of data mining, suppose we have a sales database with records of sales amount of some commodity with numerical attributes such as age, annual income, etc. Then the rectangular portion of age and annual income in some range that maximizes the amount corresponds to obtaining the association rule that maximizes the confidence that if a person is in the range, then he is most likely to buy the commodity. Similarly we can identify the second most promising customer range, etc.

The computational model in this paper is the conventional RAM, where only arithmetic operations, branching operations, and random accessibility with $O(\log n)$ bits are allowed.

The engine for our problem is an efficient algorithm for DMM. Since a sub-cubic algorithm for DMM was achieved by Fredman [14], there have been several improvements [18], [15], [16], [20], [23], [21], [9], [10]. We modify the algorithm in [18] for DMM whose complexity is $O(n^3\sqrt{\log\log n/\log n})$, and extend it to our problem. The recent improvements for DMM after [18] are slightly better, and it may be possible they can be tuned for speed-up of the k-MSA problem.

The main technique in this paper is tournament. Specifically we reorganize the structure of the maximum subarray algorithm based on divide-and-conquer into a tournament structure, which serves as an upper structure. We also reorganize the DMM algorithm into a tournament, which works as a lower structure. Through the combined tournament, the maximum, second maximum, etc. are delivered in $O(\log n)$ time per subarray.

In section 2, basic definitions of tournaments and DMM are given. In section 3, the $X+Y$ problem is defined and a well-known algorithm for it is described for the later development.

In section 4, we give the definition of the maximum subarray problem and a divide-and-conquer algorithm for it. In section 5, we reorganize the algorithm in section 4 into a tournament style, and explain how to combine it with DMM to

solve the k-MSA problem. The $X + Y$ algorithm is used as glue in this combination.

The DMM algorithm used is based on two-level divide-and-conquer. In section 6, the upper division is described. In section 7, the lower division is handled through a table look-up. The table in [18] is enhanced to handle several integers in an encoded form, rather than a single integer, at each table entry.

Section 8 concludes the paper, discussing possibilities for further speed-up and extension of similar ideas to the disjoint problem.

This paper achieves a new time complexity through a combination of known methods and tools. Note that we use the same name k in two different meanings; indexing in arrays, and the k for the k-MSA problem.

2 Basic definitions

An r-ary tournament T is an r-ary tree such that each internal node has r internal nodes and some external nodes as children, or some external nodes only as children. It also has a key, which originates from itself if it is an external node, or is extracted from one of its children if it is an internal node. Each external node has a numerical datum as a key. External nodes can be regarded as participants of the tournament. A parent has the minimum of those keys of its children. We call this a minimum tournament. A maximum tournament is similarly defined. In other words a parent is the winner among its children. The external nodes form the leaves of the tree. We form a complete r-ary tree as far as internal nodes are concerned. Also a node maintains some identity information of the winner that reached this node, such as the original position of the winner, etc. The key and this kind of information eventually propagates to the root, and the winner is selected. The size of the tournament, defined by the number of nodes, is $O(n)$, if there are n external nodes.

If we use a binary tournament for sorting, the identity can be the position of the data item in the original array. We can build up a minimum tournament for n data items in $O(n)$ time. After that, successive k minima can be chosen in $O(k \log n)$ time. This can be done by replacing the key of the winning item at the bottom level, that is, in a leaf, by ∞ and performing matches along the winning path spending $O(\log n)$ time for the second winner, etc. Thus k minima can be chosen in $O(n + k \log n)$ time in sorted order. If $k = n$, this is a sorting process in $O(n \log n)$ time, called the tournament sort. We use a similar technique of tournament in the k-MSA problem.

The distance matrix multiplication is to compute the following distance product $C = AB$ in (1) for two (n, n)-matrices $A = [a_{ij}]$ and $B = [b_{ij}]$ whose elements are real numbers. We can define (1) with "max" also.

$$c_{ij} = min_{k=1}^{n}\{a_{ik} + b_{kj}\}, (i, j = 1, ..., n) \quad (1)$$

The operation in the right-hand side of (1) is called distance matrix multiplication of the min version, and A and B are called distance matrices in this

context. The index k that gives the minimum in (1) is called the witness for c_{ij}. If we use max instead we call it the max version.

Suppose we have a three layered acyclic graph for which A is a connection matrix from layer 1 to layer 2, and B is that from layer 2 to layer 3. Each layer has vertices $1, ..., n$, and the distance from i in layer 1 to j in layer 2 is a_{ij}, and that from layer 2 to layer 3 is b_{ij}. Then c_{ij} is the shortest distance from i in layer 1 to j in layer 3.

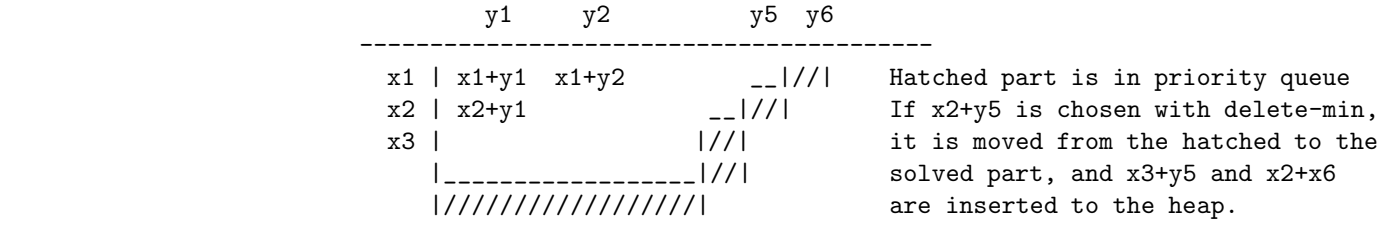
To solve the k-MSA problem, we want to find up to k shortest distances from layer 1 to layer 3 between any vertices. We use this version of extended DMM in this paper, whereas k-DMM in [2] computes k shortest paths for each pair (i, j) with i in layer 1 and j in layer 3, which is rather time consuming. If we solve DMM in $M(n)$ time in such a way that a tournament of some size becomes available for the extended DMM within the same time complexity, then k shortest distances can be found in $O(M(n) + k \log n)$ time for k up to $O(n^3)$, as shown in Sections 6 and 7.

We actually need at most k shortest distances in total for all DMMs used in our k-MSA algorithm, and our requirement is that the newly designed DMM algorithm return the next shortest distance for any pair (i, j), that is, i in layer 1 to j in layer 3, in $O(\log n)$ time.

3 X + Y problem

Let X and Y be lists of n numbers. We want to choose k smallest numbers from the set $Z = \{x + y | (x \in X) \wedge (y \in Y)\}$. We organize a tournament for each of X and Y in $O(n)$ time. Let the imaginary sorted lists be $X = (x_1, ..., x_n)$ and $Y = (y_1, ..., y_n)$. Actually they are extracted from the tournaments as the computation proceeds. We successively take elements from those sorted lists, one in $O(\log n)$ time. Obviously $x_1 + y_1$ is the smallest. The next smallest is $x_1 + y_2$ or $x_2 + y_1$. Let us have an imaginary two-dimensional array whose (i, j)-element is $x_i + y_j$. As the already selected elements occupy some portion of the top left corner, which we call the solved part, we can prepare a heap to represent the border elements adjacent to the solved region. By keeping selecting minima from the heap, and inserting new bordering elements, we can solve the problem in $O(n + k \log n)$ time.

See the figure below for illustration.



If we change the tournaments from minimum to maximum, we can find k maxima in the same amount of time. Also with similar arrangements, we can

select k largest or smallest from $X - Y = \{x - y | (x \in X) \wedge (y \in Y)\}$ in the same amount of time. We use this simple algorithm rather than sophisticated ones such as [13], since these two are equivalent in computing time for k minima in sorted order.

4 The maximum subarray problem

Now we proceed to the maximum subarray problem for an array of size (m, n). The cubic algorithm for this problem given by Bentley [7] was improved to sub-cubic by Tamaki and Tokuyama [22]. We review the simplified sub-cubic version in [19]. We give a two-dimensional array $a[1..m, 1..n]$ of real numbers as input data. The maximum subarray problem is to maximize the sum of the array portion $a[k..i, l..j]$, that is, to obtain the sum and such indices (k, l) and (i, j). We suppose the upper-left corner has co-ordinates (1,1). Bentley's algorithm finds the maximum subarray in $O(m^2 n)$ time, which is cubic $O(n^3)$ when $m = n$.

For simplicity, we assume the given array a is a square (n, n)-array. We compute the prefix sums $s[i, j]$ for array portions of $a[1..i, 1..j]$ for all i and j with boundary condition $s[i, 0] = s[0, j] = 0$. Obviously this can be done in $O(n^2)$ time for an (n, n)-array. The outer framework of the algorithm is given below. Note that the prefix sums once computed are used throughout recursion.

Algorithm M: Maximum subarray
1. If the array becomes one element, return its value.
2. Let A_{tl} be the solution for the top left quarter.
3. Let A_{tr} be the solution for the top right quarter.
4. Let A_{bl} be the solution for the bottom left quarter.
5. Let A_{br} be the solution for the bottom right quarter.
6. Let A_{column} be the solution for the column-centered problem.
7. Let $A_{left-row}$ be the solution for the row-centered problem for the left half.
8. Let $A_{right-row}$ be the solution for the row-centered problem for the right half.
9. Let the solution A be the maximum of those seven.

The *coverage* of a solution array is the smallest square region, defined by the above recursive calls, in which the solution is obtained. It is given by index pairs. The *scope* of a solution array is the index pairs $((k, l), (i, j))$ if the solution is $a[k..i, l..j]$. A coverage is also defined by the co-ordinates of the top-left corner, and those of the bottom-right corner. If we call the above algorithm for $a[1..n, 1..n]$, for example, the coverage of A is $((1, 1), (n, n))$, that of A_{tr} is $((1, n/2 + 1), (n/2, n))$, etc.

Here the column-centered problem is to obtain an array portion that crosses over the central vertical line with maximum sum, and can be solved in the following way. $A_{left-row}$ and $A_{right-row}$ can be computed similarly.

$$A_{column} = max_{k=0,l=0,i=1,j=n/2+1}^{i-1,n/2-1,n,n}\{s[i,j] - s[i,l] - s[k,j] + s[k,l]\}.$$

In the above we first fix i and k, and maximize the above by changing l and j. Then the above problem is equivalent to maximizing the following for $i = 1, ..., n$ and $k = 0, ..., i-1$.

$$A_{column}[i,k] = max_{l=0,j=n/2+1}^{n/2-1,n}\{-s[i,l] + s[k,l] + s[i,j] - s[k,j]\}$$

Let $s^*[i,j] = -s[j,i]$. Then the above problem can further be converted into

$$A_{column}[i,k] = -min_{l=0}^{n/2-1}\{s[i,l] + s^*[l,k]\} + max_{j=n/2+1}^{n}\{s[i,j] + s^*[j,k]\}$$

The first part in the above is distance matrix multiplication of the *min* version and the second part is of the *max* version. Let S_1 and S_2 be matrices whose (i,j) elements are $s[i,j-1]$ and $s[i,j+n/2]$. For an arbitrary matrix T, let T^* be that obtained by negating and transposing T. As the range of k is [0 .. $n-1$] in S_1^* and S_2^*, we shift it to [1..n]. Then the above can be computed by multiplying S_1 and S_1^* by the *min* version, multiplying S_2 and S_2^* by the *max* version, subtracting the former from the latter, that is, $S = S_2S_2^* - S_1S_1^*$, and finally taking the maximum from the lower triangle. We will re-organize this maximizing operation into a tournament later. We call the operations of extracting a triangle *triangulation*. This is effectively done by putting $-\infty$ in the upper triangle of S including the diagonal. We call this converted matrix S'.

For simplicity, we assume n is a power of 2. Then all size parameters appearing through recursion in Algorithm M are power of 2. We define the work of computing the three subarrays, A_{column}, $A_{left-row}$. and $A_{right-row}$, to be the work at level 0. The algorithm will split the array horizontally and vertically into four through the recursion to go to level 1.

Now let us analyze the time for the work at level 0. We can multiply $(n, n/2)$ and $(n/2, n)$ matrices by 4 multiplications of size $(n/2, n/2)$, and there are two such multiplications in $S = S_2S_2^* - S_1S_1^*$. We measure the time by the number of comparisons, as the rest is proportional to this. Let $M(n)$ be the time for multiplying two $(n/2, n/2)$ matrices. At level 0, we obtain an A_{column}, $A_{left-row}$, and $A_{right-row}$, spending $12M(n)$ comparisons. Thus we have the following recurrence for the total time $T(n)$. The following lemma [19] is obvious.

$$T(1) = 0,\ T(n) = 4T(n/2) + 12M(n).$$

Lemma 1. *Let c be an arbitrary constant such that $c > 0$. Suppose $M(n)$ satisfies the condition $M(n) \geq (4+c)M(n/2)$. Then the above $T(n)$ satisfies $T(n) \leq 12(1+4/c)M(n)$.*

Clearly the complexity of $O(n^3(\log\log n/\log n)^{1/2})$ for $M(n)$ satisfies the condition of the lemma with some constant $c > 0$. Thus the maximum subarray problem can be solved in $O(n^3(\log\log n/\log n)^{1/2})$ time. Since we take the maximum of several matrices component-wise in line 9 of our algorithm and maximum from S', we need an extra term of $O(n^2)$ in the recurrence to count the number of operations. This term can be absorbed by slightly increasing the constant 12 in front of $M(n)$ in the above recurrence.

5 The k-maximum subarray problem

When we solve the maximum subarray problem with Algorithm M, within the same asymptotic time complexity, we organize a four-ary tournament along the four-way recursion as internal nodes, and the three sub-problems; column centered, left-row centered, and right-row centered as external nodes, in Algorithm M. Those sub-problems are organized into tournaments in the next section. For now we regard them as leaves and assume they can respond to our request in our desired time. When we make the four-ary tournament along the execution of Algorithm M, we copy necessary portions of array a for the seven sub-problems from line 2 to 8. The total overhead time and space requirement for this part are $O(n^2 \log n)$.

Suppose the maximum subarray was returned at level 0, whose coverage and scope are $((K, L), (I, J))$ and $((k, l), (i, j))$. If this array is a single element, that is, returned at the bottom of recursion, i.e., line 1 of the algorithm, we put $-\infty$ at the leaf, and reorganize the tournament for the second maximum subarray towards the root along the winning path. The necessary time is $O(\log n)$.

If the maximum subarray is not from the bottom of recursion, it must be from one of A_{column}, $A_{left-row}$, and $A_{right-row}$ of some coverage at some level. Those three problems are organized into a tournament each, so that they can return the second maximum in $O(\log n)$ time. The coverage and scope information can identify which of the three produced the winner. We can reorganize the tournament along the winning path from this second maximum towards the root. Thus the k-maximum subarray problem can be solved in $O(M(n)+k \log n)$ time, where $M(n) = O(n^3 \sqrt{\log \log n / \log n})$.

Let us assume $K = 1$, $I = n$, $L = 1$, and $J = n$ without loss of generality. Also assume A was obtained from A_{column}, which is in turn obtained from S', that is, the lower triangle of $S = S_2 S_2^* - S_1 S_1^*$. We rewrite this equation as $S = Q - P$, where $P = S_1 S_1^*$ and $Q = S_2 S_2^*$. We assume that $S[i, k]$ for some $k < i$ gives A_{column} with the witnesses l and j for P and Q respectively. We need to find the next value for S'. To do so, we need to find the next minimum value for $P[i, k]$ and next maximum for $Q[i, k]$ with witnesses different from l and j. As is shown in the following sections, the next value for $P[i, k]$ and $Q[i, k]$ are returned in $O(\log n)$ time. Then using the X+Y algorithm, we can choose the next value for $S[i, k]$ in $O(\log n)$ time, which is delivered to the tournament for S' where other elements are intact. Thus the next value for the chosen one of the above three problems, A_{column}, $A_{left-row}$ and $A_{right-row}$, can be found in $O(\log n)$ time.

We observe at this stage that any DMM algorithm, that can deliver successive minimum distances from layer 1 to layer 3 in the context of Section 2 in $O(\log n)$ time, can be fitted into the framework of our algorithm.

6 Distance matrix multiplication by divide and conquer

We review the DMM algorithm of *min*-version in [18]. The *max*-version is similar. Matrices A, B, and C in DMM are divided into (m, m)-submatrices for $N = n/m$ as follows:

$$\begin{pmatrix} A_{1,1} & ... & A_{1,N} \\ ... & & \\ A_{N,1} & ... & A_{N,N} \end{pmatrix} \begin{pmatrix} B_{1,1} & ... & B_{1,N} \\ ... & & \\ B_{N,1} & ... & B_{N,N} \end{pmatrix} = \begin{pmatrix} C_{1,1} & ... & C_{1,N} \\ ... & & \\ C_{N,1} & ... & C_{N,N} \end{pmatrix}$$

Matrix C can be computed by

$$C = (C_{ij}), \text{ where } C_{ij} = min_{k=1}^{N}\{A_{ik}B_{kj}\}(i, j = 1, ...N). \quad (2)$$

Here the product of submatrices is defined similarly to (1) and the "min" operation is defined on submatrices. Since comparisons and additions of distances are performed in a pair, we measure the time complexity by the number of key comparisons, and omit counting the number of additions for measurement of the time complexity. We have N^3 multiplications of distance matrices in (2). Let us assume that each multiplication of (m, m)-submatrices can be done in $T(m)$ computing time, assuming precomputed tables are available. The time for constructing the tables is reasonable when m is small. The time for *min* operations in (2) is $O(n^3/m)$ in total. Thus the total time excluding table construction is given by $O(n^3/m + (n/m)^3 T(m))$. As shown below, it holds that $T(m) = O(m^2\sqrt{m})$. Thus the time becomes $O(n^3/\sqrt{m})$.

Now we further divide the small (m, m)-submatrices into rectangular matrices in the following way. We rename the matrices A_{ik} and B_{kj} in (2) by A and B. Let $M = m/l$, where $1 \leq l \leq m$. Matrix A is divided into M (m, l)-submatrices A_1, ..., A_M from left to right, and B is divided into M (l, m)-submatrices B_1, ..., B_M from top to bottom. Note that A_k are vertically rectangular and B_k are horizontally rectangular. Then the product $C = AB$ can be given by

$$C = min_{k=1}^{M} C_k, \text{ where } C_k = A_k B_k \quad (3)$$

As shown in the next section, $A_k B_k$ can be computed in $O(l^2 m)$ time. Thus the above C in (3) can be computed in $O(m^3/l + lm^2)$ time. Setting $l = \sqrt{m}$ yields $O(m^2\sqrt{m})$ time.

We define a u/l-tournament. Let us find k minima from (n, n)-matrices $X_1, ..., X_m$ for general m and n. The right-hand side of $X = min_{\ell=1}^{m} X_\ell$ is to take minimum values of matrices component-wise. For each (i, j) we organize (i, j) elements of those m matrices into a lower tournament through index ℓ. Then we organize the n^2 roots of those tournaments, which give X, into an upper tournament, We can draw k minima of those matrices from the root of the upper tournament. We call this tournament structure a u/l- tournament.

Now for the extended DMM algorithm, the "min" operation in (2) for each (i, j) is reorganized into a u/l-tournament within the same asymptotic complexity as that of DMM, by the substitution $X_k = A_{ik}B_{kj}$. As C in (2) is regarded as

an (N, N)-matrix of (m, m)-matrices, we organize a tournament of N^2 roots of these u/l-tournaments. We note that the matrix C in (3) can be updated by the next minimum in some $A_k B_k$ in $O(M) = O(m/l)$ time by sequential scanning, that is, without a tournament structure

From this construction, we can find the next minimum for the extended DMM in $O(\log n)$ time, since the next minimum in $A_k B_k$ in (3) can be found in $O(1)$ time, as is shown next.

7 How to multiply rectangular matrices

We rename again the matrices A_k and B_k in (3) by A and B. In this section we show how to compute AB, that is,

$$min_{r=1}^{l}\{a_{ir} + b_{rj}\}, \text{ for } i = 1, ..., m; j = 1, ..., m. \quad (4)$$

Note that we do not form tournaments for this "min" operation.

We assume that the lists of length m, $(a_{1r} - a_{1s}, ..., a_{mr} - a_{ms})$, and $(b_{s1} - b_{r1}, ..., b_{sm} - b_{rm})$ are already sorted for all r and s $(1 \leq r < s \leq l)$. The time for sorting will be mentioned later. Let E_{rs} and F_{rs} be the corresponding sorted lists. For each r and s, we merge lists E_{rs} and F_{rs} to form list G_{rs}. In case of a tie, we put an element from E_{rs} first into the merged list. Let H_{rs} be the list of ranks of $a_{ir} - a_{is}$ $(i = 1, ..., m)$ in G_{rs} and L_{rs} be the list of ranks of $b_{sj} - b_{rj}$ $(j = 1, ..., m)$ in G_{rs}. Let $H_{rs}[i]$ and $L_{rs}[j]$ be the ith and jth components of H_{rs} and L_{rs} respectively. Then we have $G_{rs}[H_{rs}[i]] = a_{ir} - a_{is}$ and $G_{rs}[L_{rs}[j]] = b_{sj} - b_{rj}$.

The lists H_{rs} and L_{rs} for all r and s can be made in $O(l^2 m)$ time, when the sorted lists are available. We have the following obvious equivalence for $r < s$.

$$a_{ir} + b_{rj} \leq a_{is} + b_{sj} \Longleftrightarrow a_{ir} - a_{is} \leq b_{sj} - b_{rj} \Longleftrightarrow H_{rs}[i] \leq L_{rs}[j]$$

Fredman [14] observed that the information of ordering for all i, j, r, and s in the rightmost side of the above formula is sufficient to determine the product AB by a precomputed table. This information is essentially packed in the three dimensional space of $H_{rs}[i](i = 1..m; r = 1..l; s = r + 1..l)$, and $L_{rs}[j](j = 1..m; r = 1..l; s = r + 1..l)$. This can be regarded as the three-dimensional packing.

In [18] it is observed that to compute each (i, j) element of AB, it is enough to know the above ordering for all r and s. This can be obtained from a precomputed table, which must be obtained within the total time requirement. This table is regarded as a two-dimensional packing, which allows a larger size of m. leading to a speed-up. In [20] and [21], a method by one-dimensional packing is described.

For simplicity, we omit i from $H_{rs}[i]$ and $L_{rs}[i]$, and define concatenated sequences $H[i]$ and $L[i]$ of length $l(l-1)/2$ by

$$H[i] = H_{1,2} \ldots H_{1,l} H_{2,3} \ldots H_{2,l} \ldots H_{l,l-1}$$
$$L[i] = L_{1,2} \ldots L_{1,l} L_{2,3} \ldots L_{2,l}, \ldots L_{l,l-1} \quad (5)$$

For integer sequence $(x_1, \ldots, x_p)$, let $h(x_1, \ldots, x_p) = x_1\mu^{p-1} + \ldots + x_{p-1}\mu + x_p$. Let $h(H[i])$ and $h(L[i])$ be encoded integer values for $H[i]$ and $L[i]$, where $p = l(l-1)/2$ and $\mu = 2m$. The computation of h for $H[i]$ and $L[i]$ for all i takes $O(l^2m)$ time. By consulting a precomputed table *table* with the values of $h(H[i])$ and $h(L[j])$, we can determine the value of r that gives the minimum for (4) in $O(1)$ time. For all i and j, it takes $O(m^2)$ time. Thus the time for one A_kB_k in (3) is $O(\ell^2 m)$, since $l^2 = m$. M such multiplications take $O(M\ell^2 m) = O(\ell m^2)$ time.

To compute $table[x][y]$ for any positive integers x and y, x and y are decoded into sequences H and L, which are expressed by the right-hand sides of (5). If $H_{s,r} > L_{s,r}$ for $s < r$ or $H_{r,s} < L_{r,s}$ for $r < s$, we can say r beats s in the sense that $a_{ir} + b_{rj} \leq a_{is} + b_{sj}$ if H and L represent $H[i]$ and $L[j]$. We first fix r and check this condition for all such s. We repeat this for all r. If r is not beaten by any s, it becomes the table entry, that is, $table[x][y] = r$. If there is no such r, the table entry is undefined. There are $O(((2m)^{l(l-1)/2})^2)$ possible values for all x and y, and one table entry takes $O(l(l-1)/2)$ time. Thus the table can be constructed in $O((l(l-1)/2)(2m)^{2l(l-1)/2}) = O(c^{m \log m})$ time for some constant c. Let us set $m = \log n/(\log c \log\log n)$. Then we can compute the table in $O(n)$ time.

If r is beaten by i participants, the rank of r becomes $i+1$. Let r_i be at rank i. Then we fill the (x, y) entry of $table'$, $table'[x, y]$, by $h(r_1, ..., r_l)$ with $p = l$. That is, using this function h, we encode not only the winner, but second winner, third winner, etc., into the table elements. This can also be done in $O(n)$ time, by a slight increase of constant c in the previous page.

To prepare for the extended DMM, we extend equation (4) in such a way that c_{ij} is the l-tuple of the imaginary sorted sequence, $(a_{ir_1} + b_{r_1 j}, ..., a_{ir_l} + b_{r_l j})$, of the set $\{a_{ir} + b_{rj} | 1 \leq r \leq l\}$. Note that we do not actually sort the set. The leftmost element of c_{ij}, that is, the minimum, participates in the tournament for "min" in (3). If $c_{ij} = (x_1, x_2, ..., x_l)$ and x_1 is chosen as the winner, c_{ij} is changed to $(x_2, ..., x_l, \infty)$, etc. As k can be up to $O(n^3)$, many of c_{ij} will be all infinity towards the end of computation.

This can be implemented by introducing an auxiliary matrix C'. When we compute DMM, we compute C', where $c'_{ij} = table'[h[H[i]), h(L[j])] = h(r_1, ..., r_l)$. Each r_k $(k = 1, ..., l)$ is obtained in $O(1)$ time. The elements of the sorted list of c_{ij} is delivered by decoding $C'[i, j]$ one-by-one when demanded from up-stream of the algorithm.

Example 1. $m = 5$, $2m = 10$, $h(H) = 456$, and $h(L) = 329$. Since $H_{1,2} > L_{1,2}$ and $H_{2,3} < L_{2,3}$, the winner is 2, that is, $table[456, 329] = 2$. Also we see $table'[453, 329] = 213$, since $H[1, 3] > L[1, 3]$.

$$H = \begin{bmatrix} - & 4 & 5 \\ - & - & 6 \\ - & - & - \end{bmatrix}, \quad L = \begin{bmatrix} - & 3 & 2 \\ - & - & 9 \\ - & - & - \end{bmatrix}$$

We note that the time for sorting to obtain the lists E_{rs} and F_{rs} for all k in (3) is $O(Ml^2 m \log m)$. This task of sorting, which we call presort, is done

for all A_{ij} and B_{ij} in advance, taking $O((n/m)^2(m/l)l^2 m \log m) = O(n^2 l \log m)$ time, which is absorbed in the main complexity. Thus we can compute k shortest distances in $O(M(n) + k \log n)$ time.

8 Concluding remarks

We showed an asymptotic improvement on the time complexity of the k-maximum subarray problem based on a fast algorithm for DMM. The time complexity is sub-cubic in n, when $k = o(n^3/\log n)$. If we use recent faster algorithms for DMM, it may be possible to have a better complexity bound for the k-MSA problem.

Another challenge is to use the same idea of tournament technique for the disjoint k-MSA problem. Once the maximum subarray is found, we need to exclude the occupied portion from further considerations. This was done by "hole creation" in [3], achieving a cubic time for $k = O(n/\log n)$. A "hole" causes many tournaments to be updated to offer the best subarrays to be chosen. It remains to be seen whether a similar technique can be used in the disjoint case to achieve a sub-cubic time for the same range of k.

The authors are very grateful to reviewers, whose constructive comments greatly helped us improve the description of this revised version.

References

1. Bae, S. E., and Takaoka, T., Mesh algorithms for the K maximum subarray problem, Proc. ISPAN 2004, pp 247-253, 2004
2. Bae, S. E., and Takaoka, T., Improved Algorithms for the K-Maximum Subarray Problem for Small K, COCOON 2005, LNCS 3595, pp 621–631. Also in Computer Journal, vol. 49, no. 3, pp 358–374, 2006.
3. Bae, S. E., and Takaoka, T., Algorithms for K Disjoint Maximum Subarrays, ICCS 2006, LNCS 3991, pp 595–602. Also in IJFCS, vol 18, no. 2, pp 310–339, 2007.
4. Bae, S. E., Sequential and Parallel Algorithms for the Generalized Maximum Subarray Problem, Ph. D Thesis submitted to University of Canterbury, April 2007.
5. Bengtsson, F. and J. Chen, Efficient Algorithms for the k Maximum Sums, ISAAC 2004, LNCS 3341, Springer, 137–148, 2004
6. Bengtsson, F. and J. Chen, A Note on Ranking k Maximum Sums, Technical Report Lulea University LTE-FR-0508, 2005.
7. Bentley, J, Programming Pearls - Perspective on Performance, Comm. ACM, 27 (1984) 1087-1092
8. Brodal, G. S. and Jorgensen, A. G., A Linear Time Algorithm for the k Maximal Sums Problem, private communication. Also MFCS 2007, to appear.
9. Chan, T. M., All pairs shortest paths with real weights in $O(n^3/\log n)$ time, Proc. 9th Workshop on Algorithms and Data Structures (WADS), LNCS 3608, pp 318-324, 2005
10. Chan, T. M., More algorithms for all-pairs shortest paths in weighted graphs, 39th ACM Symposium on Theory of Computing (STOC), Pages: 590 - 598, 2007
11. Cheng, C., Cheng, K., Tien, W., and Chao, K., Improved algorithms for the k maximum sums problem. Proc. ISAAC 2005, LNCS 3827 (2005) 799-808.

12. Dobosiewicz, A more efficient algorithm for min-plus multiplication, Internt. J. Comput. Math. 32 (1990) 49-60
13. Frederickson, G. N. and D. B. Johnson, The complexity of selection and ranking in X+Y and matrices with sorted rows and columns, JCSS vol. 24 (1982) 197-208
14. Fredman, M, New bounds on the complexity of the shortest path problem, SIAM Jour. Computing, vol. 5, pp 83-89, 1976
15. Han, Y, Improved algorithms for all pairs shortest paths, Info. Proc. Lett., 91 (2004) 245-250
16. Han, Y., An $O(n^3(\log\log n/\log n)^{5/4})$ time algorithm for all pairs shortest paths, Proc. 14th European Symposium on Algorithms (ESA), LNCS 4168, pp 411-417, 2006
17. Lin, T. C. and Lee, D. T., Randomized algorithm for the sum selection problem, ISAAC 2005, LNCS 3827 (2005) pp 515-523.
18. Takaoka, T., A New Upper Bound on the complexity of the all pairs shortest path problem, Info. Proc. Lett., 43 (1992) 195-199
19. Takaoka, T, Sub-cubic algorithms for the maximum subarray problem, Proc. Computing:Australasian Theory Symposium (CATS 2002), pp 189-198, 2002.
20. Takaoka, T., A Faster Algorithm for the All Pairs Shortest Path Problem and its Application, Proc. COCOON 2004, LNCS 3106, 278-289.
21. Takaoka, T., An $O(n^3 \log\log n/\log n)$ Time Algorithm for the All Pairs Shortest Path Problem, Info. Proc. Lett., vol. 96, pp 155–161, 2005.
22. Tamaki, H. and T. Tokuyama, Algorithms for the Maximum Subarray Problem Based on Matrix Multiplication, Proceedings of the 9th SODA (Symposium on Discrete Algorithms), (1998) 446-452
23. Zwick, U, A Slightly Improved Sub-Cubic Algorithm for the All Pairs Shortest Paths Problem, ISAAC 2004, LNCS 3341, pp 921-932, 2004