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

---

A thesis
submitted in partial fulfillment
of the requirements for the Degree
of
Master of Science in Computer Science
in the
University of Canterbury
By
Sang Myung Lee

University of Canterbury

2011

Table of Contents

Acknowledgement

The author wishes to express sincere appreciation to Professor Tadao Takaoka for his assistance and supervision in the preparation of this manuscript. In addition, special thanks to the staff of the college of science for their patience that allows me to finish this work.

Abstracts

The maximum subarray problem is to find the array portion that maximizes the sum of array elements in it. This problem was first introduced by Grenander and brought to computer science by Bentley in 1984. This problem has been branched out into other problems based on their characteristics. *k*-overlapping maximum subarray problem where the overlapping solutions are allowed, and *k*-disjoint maximum subarray problem where all the solutions are disjoint from each other are those. For *k*-overlapping maximum subarray problems, significant improvement have been made since the problem was first introduced. The best known complexities of this problem are $O(n^3 + klog\,n)$, 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 $k = O(\sqrt{log\,n / log\,log\,n})$.

For *k*-disjoint maximum subarrsy, Ruzzo and Tompa gave an $O(n)$ time solution for one-dimension. This solution is, however, difficult to extend to two-dimensions. While a trivial solution of $O(kn^3)$ time is easily obtainable for two-dimensions, little study has been undertaken to better this. This paper introduces a faster algorithm for the *k*-disjoint maximum sub-array problem under the conventional RAM model, based on distance matrix multiplication. Specifically $O(n^3\sqrt{\log\log n / \log n} + kn^2 log\,n)$ is achieved for the problem. This complexity is sub-cubic when $k < O(n/log\,n)$. Also, DMM reuse technique is introduced for the maximum subarray problem based on recursion for space optimization.

1. Introduction

Data mining is to extract useful information from a vast amount of data, typically from a large database. Here useful information means some interesting information that can be found only going through a large database with a computer, which a human can never scan through with bare eyes and hands. Suppose there is a record of monthly sales of smart phones for one year at a retail store in some city as a one-dimensional array. The maximum subarray (MSA) problem scans through the database to determine the array portions that sums to the maximum value with respect to all possible array portions within the input array, which gives seasonal trend of the sales. This sort of data mining methods are described in [1] and [2]. Since the array elements are all non-negative, the obvious solution is the whole array. If the mean value of the array elements is subtracted from each array element, and consider the modified maximum subarray problem, we can have more accurate estimation on the sales trends. When the input array is two-dimensional, we find a rectangular subarray portion with the largest possible sum. The two-dimensional maximum subarray problem can be used in digital video image where every frame in it is represented as a two dimensional array. If the mean value of the pixels of a frame is subtracted from the each pixel value in grey-scale video image, we can identify the brightest portion in it. Even further, if a well-established background model is subtracted from the current video frame then the maximum subarray problem can spot new objects in the frame and track them through the following frames.

The maximum subarray problem was first introduced by Grenander and brought to computer science by Bentley [3] in 1984 as an example to discuss the efficiency of computer programs for the two-dimensional problem with an algorithm of $O(n^3)$, and attracts attention from data mining point of view [4]. It was later improved by Tamaki and Tokuyama [5] to a sub-cubic time algorithm based on distance matrix multiplication (DMM), which is further simplified by Takaoka [6].

This problem has been branched out into other problems based on their characteristics. *k*-overlapping maximum subarray problem and *k*-disjoint maximum subarray problem are those, whose detailed descriptions and known algorithms with their results are discussed in chapter 4. For *k*-overlapping maximum subarray problems where overlapping is allowed for solution arrays, significant improvements have been made since the problem was first discussed in [7] and [8]. Recent development by Cheng et al. [9] and Bengtsson and Chen [10] established $O(n + k \log k)$ time algorithm, and Brodal, et. al achieved $O(n + k)$ [11] for one-dimensional problem. For two-dimensions, $O(n^3)$ is possible in [12] and [9], and lately a sub-cubic algorithm was developed by Bae and Takaoka [13].

The goal of the *k*-disjoint maximum subarray problem is to find *k*-maximum subarrays, which are disjoint from one another. Ruzzo and Tompa's algorithm [14] finds all disjoint maximum subarrays in $O(n)$ time for one-dimension. However, little study has been undertaken on this problem for higher dimensions. In this paper, a new $O(m^2 n\sqrt{\log\log n / \log n} + km^2 \log n)$ time solution for two-dimension is presented, where (m, n) is the size of the input array. This is sub-cubic time when $m = n$ and $k < n/\log n$.

In chapter 2, the basic definition of the maximum subarray problem and a divide-and-conquer algorithm for the problem are given. In chapter 3, a faster algorithm for the DMM using two-level divide-and-conquer and table-lookup method is explained in detail, since the new algorithm is based on it. In chapter 4, $X + Y$ problem is defined and its well-known algorithm is described. Also a sub-cubic algorithm for the *k*-overlapping maximum subarray problem by Bae and Takaoka [13] is explained with a brief introduction of an algorithm for the *k*-disjoint maximum subarray problem. In chapter 5, which is the main chapter of this paper, the new sub-cubic *k*-disjoint maximum algorithm is presented and finally chapter 6 concludes the paper, discussing possibilities for further speed-up.

The computational model in this paper is the conventional RAM, where only arithmetic operations, branching operations, and random accessibility with $k < O(log\, n)$ bits are allowed and the same name k is used in two different meanings; indexing in arrays, and the k for the k-maximum subarray (k-MSA) problem. Also note that the terms of array and matrix are used interchangeably in this paper.

2. Maximum Subarray Problem

2.1 Basic Definitions

As described above, the maximum subarray problem is to find the consecutive portion of an array that maximizes the sum of array elements in the portion.

Example 2.1 *Let a be given by*

$$[3 \quad 51 \quad -41 \quad -57 \quad \mathbf{|52 \quad 59 \quad -11 \quad 93|} \quad -55 \quad -71 \quad 21 \quad 21]$$

Then the maximum subarray is given by the portion from index 5 to index 8 with the maximum sum 193.

In most applications, one-dimensional and two-dimensional arrays are used. In two dimensional arrays, the MSA problem is to comput a rectangular portion in the given two-dimensional array that maximizes the sum of array elements in it.

Example 2.2 *Let b given by*

$$\begin{bmatrix} -1 & 2 & -3 & 5 & -4 & -8 & 3 & -3 \\ 2 & -4 & -6 & -8 & 2 & -5 & 4 & 1 \\ 3 & -2 & 9 & -9 & \mathbf{|3} & \mathbf{6|} & -5 & 2 \\ 1 & -3 & 5 & -7 & \mathbf{|8} & \mathbf{2|} & 2 & -6 \end{bmatrix}$$

Then the maximum subarray is given by the rectangle defined by the upper left corner (3, 5) and the lower right corner (4, 6) with the maximum sum 19.

For the one-dimensional case, there is an optimal $O(n)$ time sequential solution, known as Kadane's algorithm [3] and a simple extension of this solution can solve the two-dimensional problem in $O(m^2n)$ time for an (m, n)-array ($m \leq n$), which is cubic when $m = n$ [3]. This is done by separating the two-dimensional array into every possible row strips, which is called strip separation, and applying the one-dimensional Kadane's algorithm on each strip. Also, a sub-cubic time algorithm for two-dimentional case was obtained by Tamaki and Tokuyama [5] by reducing the problem to DMM and showing that the time complexities of the two problems are the same order. Takaoka simplified the algorithm later for implementation [6] which is explained in chapter 2.3.

2.2 Distance Matrix Multiplication

We review distance matrix multiplication since it is the engine for the algorithms in this paper.

Normally we multiply two (n, n)-matrices over real numbers using "+" and "*". Let $C = AB$ where A, B and C are all (n, n) matrices. Then

$$c_{ij} = \sum_{k=1}^{n} a_{ik} * b_{kj} \ (i, j = 1, \dots, n)$$

We can define distance matrix multiplication $C = AB$ by corresponding the above "+" to "*min*" and "*" to "+" as follows

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

Example 2.3 *Distance Matrix Multiplication*

$$\begin{bmatrix} 1 & -3 & 7 \\ \infty & 5 & \infty \\ 8 & 2 & -5 \end{bmatrix} * \begin{bmatrix} 8 & \infty & -4 \\ -3 & 0 & -7 \\ 5 & -2 & 1 \end{bmatrix} = \begin{bmatrix} -6 & -3 & -10 \\ 2 & 5 & -2 \\ -1 & -7 & -5 \end{bmatrix}$$

c_{11} is decided by min{1+8, -3+-3, 7+5}

The intuitive meaning of c_{ij} is the distance of the shortest path from vertex *i* in the first layer to vertex *j* in the third layer in the following graph. The distance from *i* in the first layer to *k* in the second is a_{ik} and that from the second layer to the third is b_{kj}. The index *k* that gives the minimum is called the witness for c_{ij}

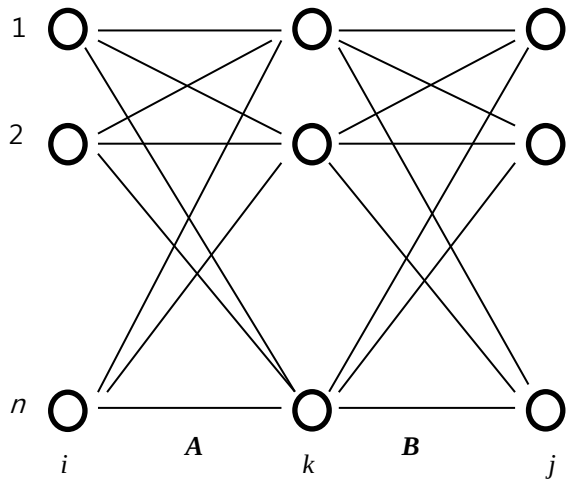


Figure 2.1 *All Pairs Shortest Paths from i to j*

In this figure 2.1, matrices *A* and *B* are to show the connection distance from layer to layer. We can define the max version by changing the symbol "*min*" to "*max*" in the above formula. This corresponds to longest paths from layer 1 to layer 3. The original version is called the min version and the other one is called max version in this paper.

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 subsequent shortest distances can be found in $O(M(n)+k \log n)$ time for *k* up to $O(n^3)$, as shown in Chapters 3.

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.

2.3 Prefix Sum Array and Maximum Subarray Problem by Distance Matrix Multiplication

The central algorithmic concept in the new algorithm is that of prefix sum array. The prefix sum of a one-dimensional array a at position i, denoted by $s[i]$, is the sum of $a[1], \ldots, a[i]$. The prefix sum array can be computed in linear time $O(n)$ by

$s[0] \leftarrow 0;$
for $i \leftarrow 1$ to n do $s[i] \leftarrow s[i-1] + a[i];$

as $s[x] = \sum_{i=1}^{x} a[i]$, the sum of $a[x \ldots y]$ is computed by the subtraction of these prefix sums as

$$\sum_{i=x}^{y} a[i] = s[y] - s[x-1]$$

To yield the maximum sum from a one-dimensional array, we have to find indices x, y that maximize $\sum_{i=x}^{y} a[i]$. In prefix sum array, $s[1, \ldots, n]$, the maximum subarray is defined by

For all $x, y \in [1, \ldots, n]$
Max Sum = $max_{\,1 \le y \le n}\, s[y] - min_{\,0 \le x \le y-1}\, s[x]$

Note that the notations *max* and *min* are used for operations.

The prefix sum array of a given two-dimensional array can be defined similarly. The prefix sum at position $s[i][j]$ of a two-dimensional (m, n)-array a is the sum of array portion $a[1, \ldots, i][1, \ldots, j]$ for all i and j with boundary condition $s[i][0] = s[0][j] = 0$, which can be calculated in $O(mn)$ time.

As $s[i][j] = \sum_{p=1,q=1}^{i,j} a[p][q]$, the sum of $a[k \ldots i][l \ldots j]$ is computed by the subtraction of these prefix sums as:

$$\sum_{p=k,q=l}^{i,j} a[p][\mathrm{q}] = s[i][j] - s[k][j] - s[i][l] + s[k][l]$$

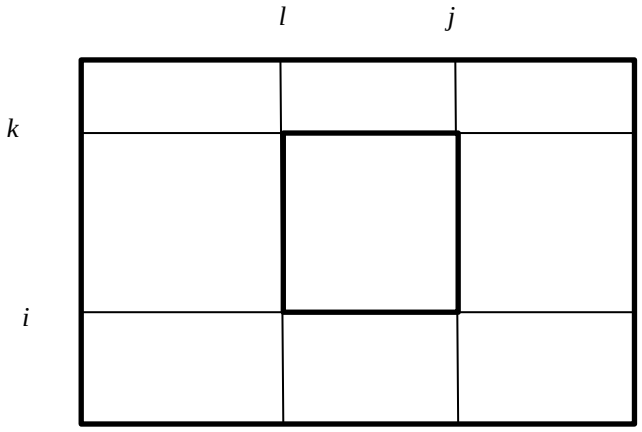


Figure 2.2 *Sum of Subarray in Prefix Sum Array in two-dimension*

To maximize the sum from a two-dimensional array, we have to find indices (k, l), (i, j) that maximize $\sum_{p=k,q=l}^{i,j} a[p][\mathrm{q}]$. In prefix sum array s, the maximum subarray is defined by

$$\text{Max Sum} = max_{k=0,l=0,i=1,j=1}^{m-1,n-1,m,n}\{s[i][j] - s[k][j] - s[i][l] + s[k][l]\}$$

$$= max_{i=1,j=1,k=0}^{m,n,i-1}\{s[i][j] - s[k][j]\} - min_{i=1,l=0,k=0}^{m,j-1,i-1}\{s[i][l] - s[k][l]\}$$

Let $s^*[j][k] = -s[k][j]$ and $s^*[l][k] = -s[k][l]$, then the above problem can further be converted into

$$\text{Max Sum} = max_{i=1,j=1,k=0}^{m,n,i-1}\{s[i][j] + s^*[j][k]\} - min_{i=1,l=0,k=0}^{m,j-1,i-1}\{s[i][l] + s^*[l][k]\}$$

The first part in the above is distance matrix multiplication of the max version and the second part is of the min version. Let S_1 and S_2 be matrices whose (i, j) elements are $s[i][j-1]$ and $s[i][j]$. For an arbitrary matrix T, let T^* be that obtained by negating and transposing T and the resulting matrix by DMM between T and T^* is called the DMM matrix of T in this paper. Then the above can be computed by multiplying S_1 and S_1^* by the min version, that is min DMM matrix of S_1, multiplying S_2 and S_2^* by the max version, that is max DMM matrix of S_2, and finally subtracting the former from the latter and taking the maximum.

Now we review the simplified sub-cubic version in [6], which is the starting point of the new algorithm. A two-dimensional (m, n)-array $a[1, \ldots, m][1, \ldots, n]$ of real numbers is given as input data. The maximum subarray problem is to maximize the sum of the array portion $a[k, \ldots, i][l, \ldots, j]$, 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).

For simplicity, the given array a is assumed to be a square (n, n)-array. We compute the prefix sums $s[i][j]$ for array portions of $a[1, \ldots, i][1, \ldots, j]$ 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_{row} be the solution for the row-centered problem.
8. Let the solution A be the maximum of those six.

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

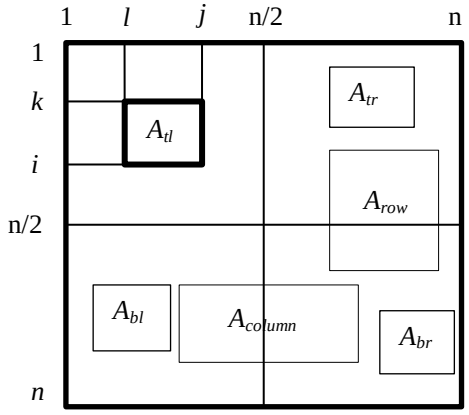


Figure 2.3 *Algorithm M. The solution of the coverage* ((1, 1), (*n*, *n*)) *is the maximum of the six solutions, that is* A_{tl} *in this figure.* A_{tl}, A_{tr}, A_{bl} *and* A_{br} *are the solutions of their own coverage that is (n/2, n/2)-submatrix.*

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

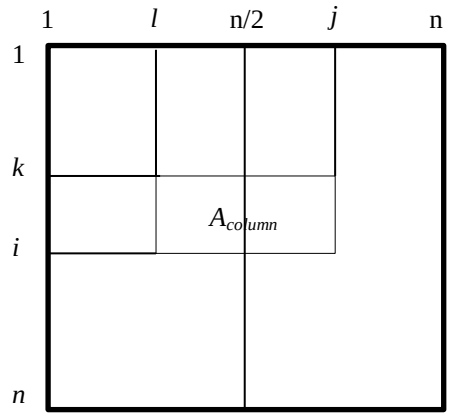


Figure 2.4 *The column-centered problem*

In the Figure 2.4, We first fix *k* and *i*, and maximize the column-centered problem by changing *l* and *j*. then the problem is equivalent to maximizing the following for $i = 1, \dots, n$ and $k = 1, \dots, i - 1$.

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

As described earlier in this section, the first part in the above is distance matrix multiplication of the max version and the second part is of the min version. Let S_1 and S_2 be matrices whose ranges are $s[1, \dots, n][1, \dots, n/2 - 1]$ and $s[1, \dots, n][n/2 + 1, \dots, n]$. As the range of *k* is $[0, \dots, n - 1]$ in S_1* and S_2*, we shift it to $[1, \dots, n]$. 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 = max S_2S_2* – min S_1S_1*, and finally taking the maximum value from the matrix *S*. We will re-organize this maximizing operation into a tournament later.

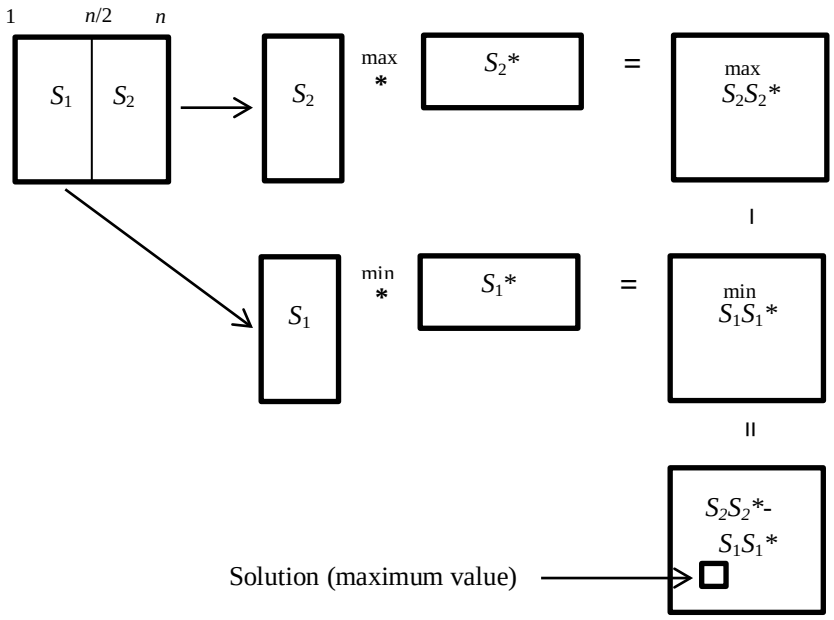
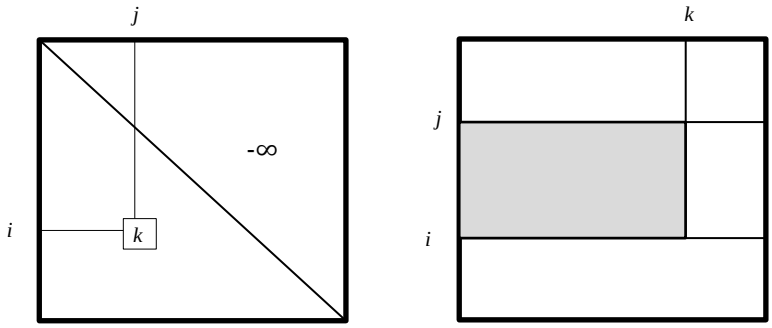


Figure 2.5 *How to solve the column-centered problem*

When we do DMM for the prefix sum matrix S of an arbitrary matrix A with its negated and transposed matrix S^*, the elements of the resulting matrix, that is called DMM matrix of S, represent the maximum or the minimum sums depending on the version of the DMM for each of the every possible horizontal strip on the original array A. the location of the solution is identified by the index of the element and its witness k. For example, an element in the resulting DMM matrix in min version whose index is (i, j) with witness k represents the minimum sum for the strip from row j to row i and it is bounded by column 1 and column k on the original array A. In other words, the sum of all the elements on the original array from $(j, 1)$ to (i, k) is the minimum sum for the (j, i)-strip. Note that the upper right triangles of the DMM matrices are not needed for calculating MSA problem where $i < j$, because starting row of a strip cannot be greater than its ending row. We call the operations of extracting a triangle triangulation and this is effectively done by putting $-\infty$ in the upper triangle of S. The converted matrix of S is now called S'.



DMM Matrix for *a* Original Matrix *a*

Figure 2.6 *The relationship between the min DMM matrix and its original matrix a. The value of the element (i, j) in the DMM matrix is the sum of the shaded portion of the original matrix a, that is the minimum sum for the strip. The index (i, j) with the witness k of the DMM matrix represent the top left corner (j, 1) and bottom right corner (i, k) of the solution in a. Note that the upper right triangle of the DMM matrix is filled with –∞.*

If n is assumed to be a power of 2 for simplicity, then all size parameters appearing through recursion in Algorithm M are power of 2. We define the work of computing two subarrays, A_{column} and A_{row} to be the work at level 0. The algorithm will split the array horizontally and vertically into four subarrays 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 matrices in size (n/2, n/2) and there are two such multiplications in $S = \max S_2S_2^* - \min S_1S_1^*$.

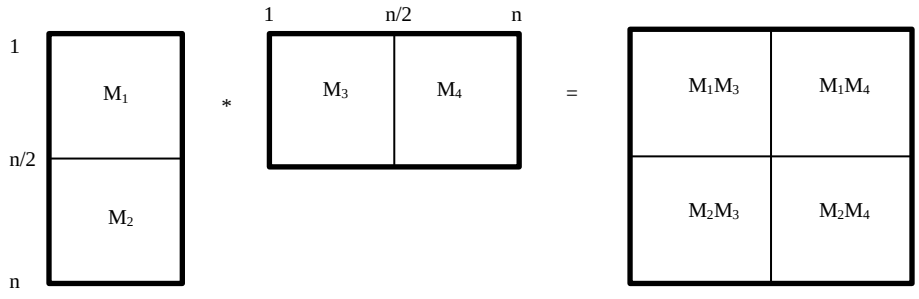


Figure 2.7 *We can multiply (n, n/2) and (n/2, n) matrices by 4 multiplications of matrices in size (n/2, n/2)*

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} and A_{row}, spending $16M(n)$ comparisons since we need $8M(n)$ each. Thus we have the following recurrence for the total time $T(n)$.

$$T(1) = 0$$
$$T(n) = 4T(n/2) + 16M(n).$$

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 $T(n) \leq 16(1 + 4/c)M(n)$.

Proof. The condition on M(n) means that its asymptotic growth ratio is more than n^2. If $M(n) \geq (4 + c)M(n/2)$ holds for $T(1)$ from the algorithm, we assume it also holds for $T(n/2)$ for induction. Then

$T(n) = 4T(n/2) + 16M(n)$ by definition
$T(n) \leq 64(1+ 4/c)M(n/2) + 4M(n)$ since $T(n/2) \leq 16(1+4/c)M(n/2)$
$T(n) \leq 64(1+4/c)(M(n)/(4+c)) + 4M(n)$ since $M(n) \geq (4 + c)M(n/2)$
$T(n) \leq 16(1 + 4/c)M(n)$

Now suppose one or both of m and n are not given by power of 2. By embedding the array a in the array of size (m', n') such that m' and/or n' are next powers of 2 and the gap is filled with 0, we can solve the original problem in the complexity of the same order.

3 A Faster Algorithm for Distance Matrix Multiplication

3.1 Distance Matrix Multiplication by Divide and Conquer

The engine for our problem is an efficient algorithm for DMM. Since a sub-cubic algorithm for DMM was achieved by Fredman [15], there have been several improvements [16], [17], [18], [19], [20], [21], [22], [23]. We review the DMM algorithm of min version in [16] whose complexity is $O(n^3\sqrt{log\ log\ n\ /\ log\ n})$, that is modified and extended to the new algorithm. The max version is similar. The recent improvements for DMM are slightly better than [16], and it may be possible that they can be tuned for speed-up of the *k*-MSA problem.

Let A and B be (*n, n*)-matrices whose compontes are nonnegative real numbers and we are to compute DMM matrix C between the two matrices in min version. Now the matrices *A* and *B* are divided into (*m, m*)-submatrices for *N* = *n*/*m*, then Matrix *C* can be computed as follows:

$$\begin{pmatrix} A_{11} & \dots & A_{1N} \\ \dots & \dots & \dots \\ A_{N1} & \dots & A_{NN} \end{pmatrix} \begin{pmatrix} B_{11} & \dots & B_{1N} \\ \dots & \dots & \dots \\ B_{N1} & \dots & B_{NN} \end{pmatrix} = \begin{pmatrix} C_{11} & \dots & C_{1N} \\ \dots & \dots & \dots \\ C_{N1} & \dots & C_{NN} \end{pmatrix}$$

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

The product of submatrices is DMM in min version as defined in (1) and the "*min*" operation in (2) is defined on the submatrices by taking the "*min*" operation component-wise. Since additions and comparisons 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 $O(n^3/m + (n/m)^3 T(m))$. In the next section it is shown that $T(m) = O(m^2\sqrt{m})$, which makes the time become $O(n^3/\sqrt{m})$.

3.2 Distance Matrix Multiplication by Table-Lookup

In this section it is explained how to multiply the (m, m)-submatrices A_{ik} and B_{kj} for C_{ij} in (2). Now the matrices A_{ik} and B_{kj} are renamed by A and B for simplicity of explanation and the maxrix A and B is further divided in the following way. 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.

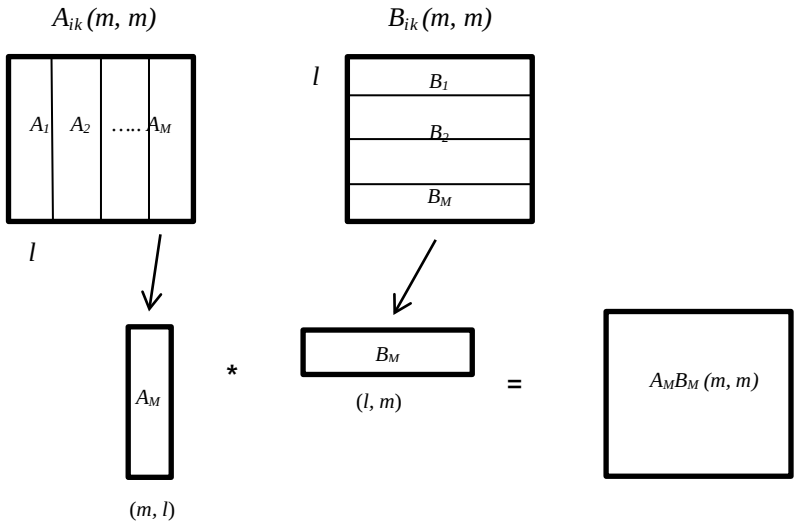


Figure 2.4 *matrix A_{ik} and B_{kj} are further divided into m/l submatrices.*

Then the product $C = AB$ can be given by

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

It is shown later, A_kB_k can be computed in $O(l^2m)$ time, assuming that a precomputed table is available. 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.

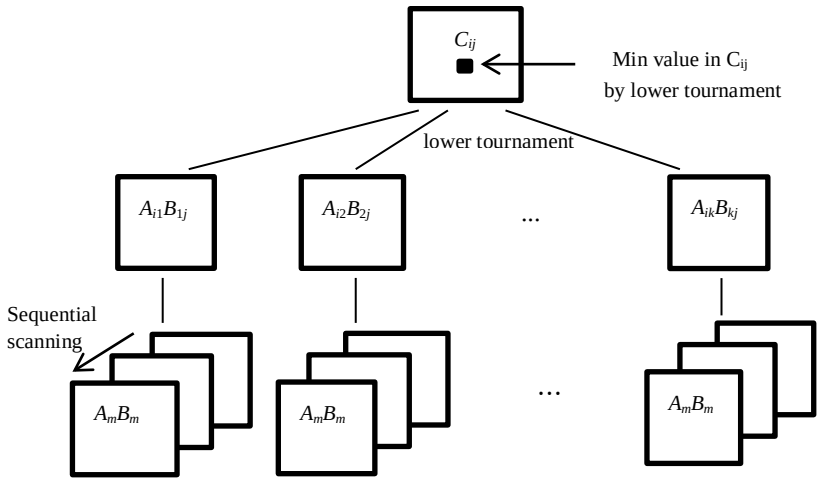


Figure 2.5 *DMM by Divide-and-Conquer*

We define a *u/l*-tournament. Let us find *k* minima from *m* (*n*, *n*)-matrices $X_1, \ldots, X_m$ for general *m* and *n*. The right-hand side of $X = min_{t=1}^{m} X_t$ is to take minimum values of matrices component-wise. The elements at the same index (*i*, *j*) in each matrix are organized into a lower tournament through the index t, then the n^2 roots of those tournaments, which give *X*, are also organized into an upper tournament. *k* minima of those matrices can be drawn from the root of the upper tournament in this structure. 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_kB_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_kB_k in (3) can be found in *O*(1) time, as is shown next. Note that *O*(*m/l*) is absorbed in *O*(*log n*).

Now the matrices A_k and B_k in (3) are renamed again by A and B to show how to compute AB, that is,

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

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

We assume that the lists of length m, $(a_{1r} - a_{1s}, \dots, a_{mr} - a_{ms})$, and $(b_{s1} - b_{r1}, \dots, b_{sm} - b_{rm})$ are already sorted for all r and s $(1 \le r < s \le l)$. 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 $a_{ir} - a_{is}$ $(i = 1, \dots, m)$ in G_{rs} and L_{rs} be the list of ranks of $b_{sj} - b_{rj}$ $(j = 1, \dots, m)$ 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 $O(l^2m)$ time, when the sorted lists are available. We have the following obvious equivalence for $r < s$.

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

Fredman [15] 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 $H_{rs}[i](i = 1, \dots, m;\quad r = 1, \dots, l;\quad s = r + 1, \dots, l)$, and $L_{rs}[j](j = 1, \dots, m;\quad r = 1, \dots, l;\quad s = r + 1, \dots, l)$. This can be regarded as the three-dimensional packing.

In [16] 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 [19] 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 $O(l^2m)$, since $l^2 = m$ and M such multiplications take $O(Ml^2m) = O(lm^2)$ time, since $M = m/l$.

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 $H_{sr} > L_{sr}$ for $s < r$ or $H_{rs} < L_{rs}$ 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 $h(r_1, \ldots, r_l)$ 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, $(a_{ir_1} + b_{r_1 j}, \ldots, a_{ir_l} + b_{r_l j})$, of the set $\{a_{ir} + b_{rj} \mid 1 \leq r \leq l\}$. 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 $c_{ij} = (x_1, x_2, \ldots, x_l)$ and x_1 is chosen as the winner, c_{ij} is changed to $(x_2, \ldots, x_l, \infty)$, 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′ $[h[H[i]), h(L[j])] = h(r_1, \ldots, r_l)$. 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 3.1

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

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'* [456, 329] = 231, since $H[1, 3] > L[1, 3]$.

We note that the time for sorting to obtain the lists E_{rs} and F_{rs} for all k in (3) is $O(Ml^2\ mlogm)$. 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.

4 *k*-Maximum Subarray Problem

Once the maximum sum is found, finding *k*-maximum sums is a natural extension. The *k*-maximum subarray (*k*-MSA) problem is to obtain the maximum subarray, the second maximum subarray, … , the *k*-th maximum subarray in sorted order for *k* up to $O(n^4)$. In many applications we need to find up to *k*-th maximum. For example, suppose the database is for a geographical distribution of customers, and we need to post flyers to the most loyal customers. The identified rectangle region for posting may not be very suitable due to road construction, etc. then we need the second or third alternative. This problem can be further defined by two such problems. One is the general case where physical overlapping of portions is allowed, and the other is only for disjoint portions. In this chapter, the *k*-maximum subarray problem where physical overlapping is allowed is called *k*-overlapping maximum subarray problem, and the other one where the overlapping is not allowed is called *k*-disjoint maximum subarray problem. We consider the general problem first.

Let $M(n)$ be the time complexity for DMM for an (n, n)-matrix. The problem is solved by Bae and Takaoka in $O(M(n)+k \log n)$ time for the general problem with an (n, n)-array, where $M(n) = O(n^3\sqrt{\log\log n / \log n})$ in [13], which is explained in detail in the following chapter.

Preceding results for the one-dimensional problem are $O(kn)$ by Bae and Takaoka [7], $O\left(\min\left(n\sqrt{k}, nlog^2 n\right)\right)$ by Bengtsson and Chen [8], $O(n \log k)$ by Bae and Takaoka [12], $O(n + k \log n)$ by Bae [24], Cheng, et. al. [9], Bengtsson, et. al. [10], $O(n \log n + k)$ expected time by Lin, et. al. [25], and $O(n + k)$ by Brodal, et. al. [11]. 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 $O(kn^3\sqrt{\log\log n / \log n})$ by [12] and $O(n^3 + k)$ by [11]. 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. The result of the algorithm in this chapter and [12] show an extension of an optimal algorithm in one dimension to two dimensions does not produce optimal solutions for the two-dimensional problem.

4.1 *k*-Overlapping Maximum Subarray Problem

We review the sub-cubic algorithm by Bae and Takaoka [13], with which the new algorithm for *k*-disjoint maximum subarray problem shares the same basic structure.

4.1.1 Tournament

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.

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 propagate 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.

4.1.2 $X + Y$ Problem

Let X and Y be lists of n numbers. We want to choose k smallest numbers from the set $Z = \{x + y \mid (x \in X) \cap (y \in Y)\}$. We organize a tournament for each of X and Y in $O(n)$ time. Let the imaginary sorted lists be $X = (x_1, \ldots, x_n)$ and $Y = (y_1, \ldots, y_n)$. 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 $X - Y = \{x - y \mid (x \in X) \cap (y \in Y)\}$ in the same amount of time. We use this simple algorithm rather than sophisticated ones such as [26], since these two are equivalent in computing time for k minima in sorted order.

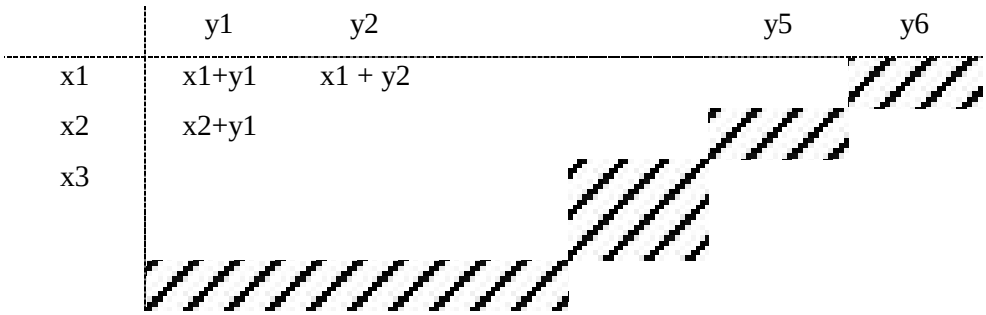


Figure 4.1 *Hatched part is in priority queue. If x2+y5 is chosen with delete-min, it is moved from the hatched to the solved part, and x3+y5 and x2+y6 are inserted to the heap*

4.1.3 The Main Algorithm

When we solve the maximum subarray problem of a two-dimensional (n, n)-array with Algorithm M in chapter 2.3, within the same asymptotic time complexity, we organize a four-ary tournament along the four-way recursion as internal nodes, and the two sub-problems; column-centered and row-centered as external nodes, in Algorithm M.

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_{row} be the solution for the row-centered problem.
8. Let the solution A be the maximum of those six.

When we make the four-ary tournament along the execution of Algorithm M, we copy necessary portions of the given prefix sum array s for the six sub-problems from line 2 to 7 and while solving the column-centered and row-centered problems by DMM for the maximum sum, the two sub-problems are organized into tournaments as described in chapter 3. 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 location 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} or A_{row} of some coverage at some level. As those two problems are organized into a tournament each, they can return the second maximum in $O(log\ n)$ time. The coverage and location information can identify which of the two 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 $M(n) = O(n^3\sqrt{\log\log n / \log n})$.

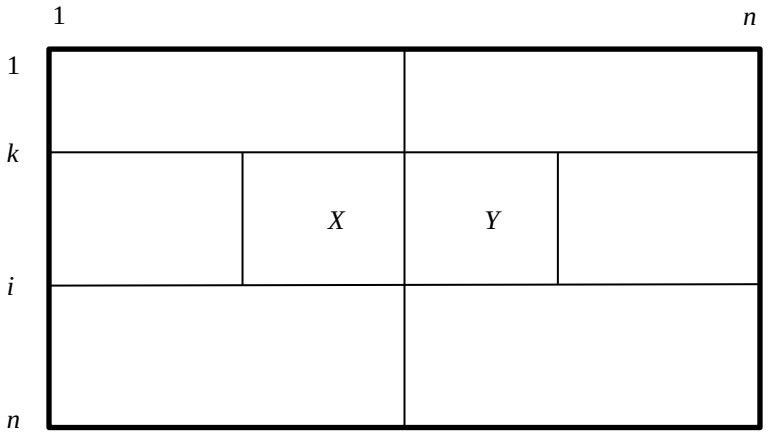


Figure 4.2 *Column Centered Problem*

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 $S = \max S_2S_2^* - \min S_1S_1^*$. We rewrite this equation as $S = Q - P$, where $P = \min S_1S_1^*$ and $Q = \max S_2S_2^*$. 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 chapters 3, 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. See Figure 4.2 for illustration. Thus the next value for the chosen one of the above two problems, A_{column} and A_{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 Chapter 2.2 in $O(log\ n)$ time, can be fitted into the framework of our algorithm

4.2 *k*-Disjoint Maximum Subarray Problem

k-disjoint maximum subarray Problem is to find the maximum subarray portion up to *k*-th without overlapping each other. 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 [27] for larger k. The new algorithm presented in the following chapter achieves sub-cubic time for this problem.

5 *k*-Disjoint Maximum Subarray Problem

5.1 Building Up Tournaments

After we find the maximum subarray, we want to find the next maximum subarray from the remianing portion. That is, we want to find *k*-disjoint maximum subarrays. As the general problem, in which the overlapping is allowed, Algorithm M and tournaments play the main roles in this algorithm for *k*-disjoint maximum subarray problem. While it is solving the maximum subarray problem with Algorithm M, a four-ary tournament are organized along the four-way recursion as internal nodes, with the two local *u*/*l*-tournaments for the column-centered and row-centered problem for each internal node as external nodes. Once the maximum sum is found and the tournaments are organized, the $X + Y$ problem takes part in the algorithm for the next maximum.

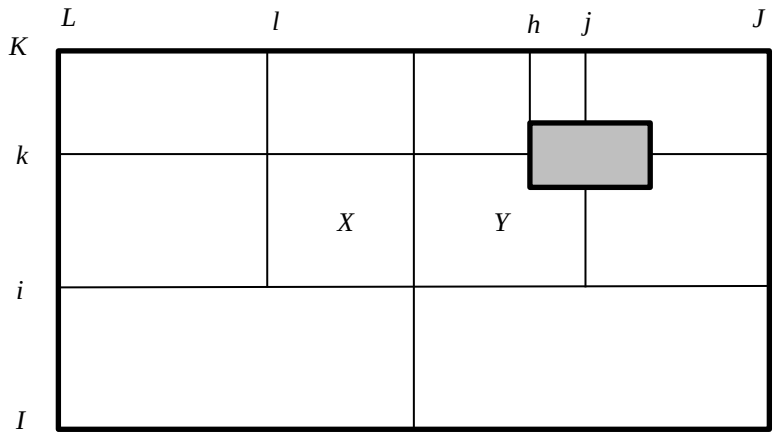


Figure 5.1 *Column centered problem with a hole over Y*

In the overlapping problem, successive minima and maxima for X and Y are selected in $O(log\ n)$ time for the next value of $X + Y$. In other word, the $X + Y$ problem played the role of glue between the four-way recursion tree of upper tournament structure and the *u*/*l*-tournament trees of the lower DMM tournament structures. When we require successive maximum subarrys to be disjoint, the previous choice becomes a hole in the array *a*, which affect various nodes in the tournament tree upward and downward.

Suppose the column centered solution at location $((k, l), (i, j))$ in the coverage $((K, L), (I, J))$ is chosen as the maximum subarray. Then the DMM solutions for the strips that intersect with the solution strip in the same coverage are all removed from further consideration. If a hole is created in another coverage like in Figure 3, the tournaments in affected coverages need to be modified. In the figure, the tournament for Y at position h (beginning of the hole) and to the right must be removed. After all modifications on the tournament trees are done, we can finalize the next winner of the whole tournament tree in a bottom up fashion. Note that the total number of affected strips in all layers on the four-way recursion is $O(n^2)$. In the following sections, we show that a modification on a strip can be done in $O(log\ n)$ time so that the selection of the next winner can be done in $O(n^2\ log\ n)$ time.

5.2 Modified *X* + *Y* Problem

We extend the *X* + *Y* problem to a modified *X* + *Y* problem, where some portion of *X* or *Y* is removed each time the maximum or minimum is found. More specifically *X* and *Y* are organized into tournaments, keeping the original order from left to right in which *X* and *Y* are given. Each node of either tree has two key values; one by which a winner is chosen and the other the position index of the key. We express those two key values by *key*(*v*) and *pos*(*v*) of node *v*, and also its content of node *v*, *c*(*v*), is defined by *c*(*v*) = (*key*(*v*), *pos*(*v*)). The tree for *X* is cut from left, whereas that for *Y* is cut from right. *X* is organized into a min-tournament (*Y* is into a max-tournament), where the winner at each node is determined by the *min* (*max*) key. A left cut at *i* is to cut the tree from position *i* to the left and a right cut is similarly defined. We perform a left cut on the tree for *X* and right cut on the tree for *Y*. The trees are a mixture of heap and binary search tree; vertically *key* follows the heap property and horizontally *pos* follows the binary search tree property. An example of a maximum tournament tree is shown in Figure 5.2 and 5.3.

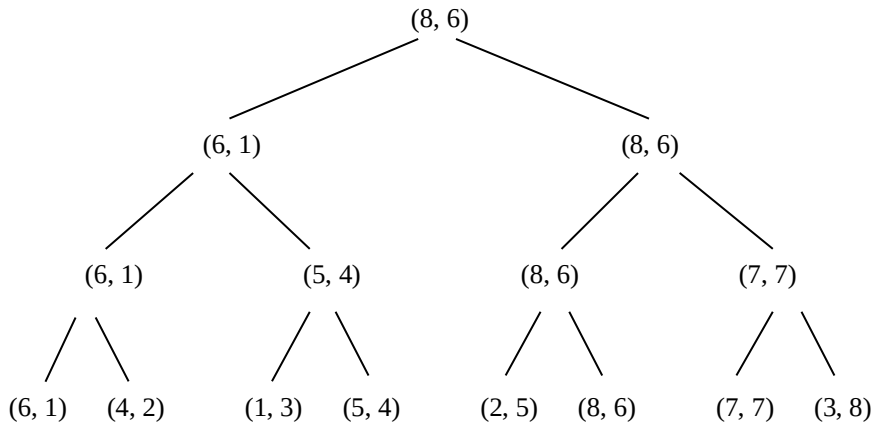


Figure 5.2 *A max-tournament tree. Key 8 at position (index) 6 is the winner.*

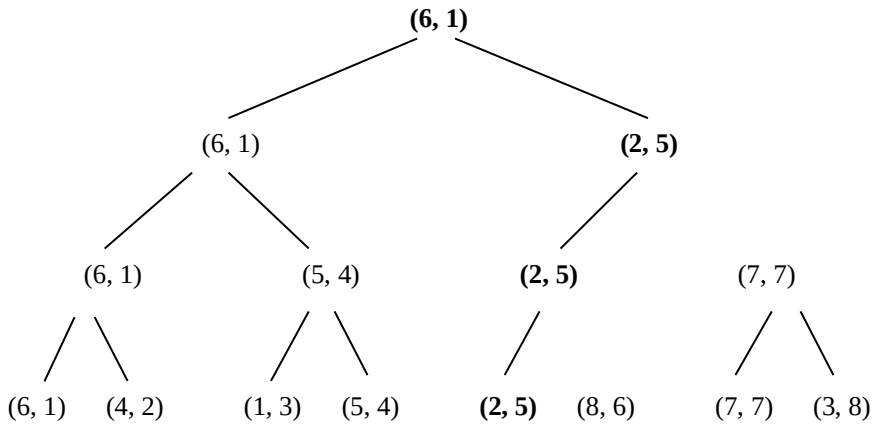


Figure 5.3 *After a right cut performed after the index 5. Key 6 at position (index) 1 is the new winner. The changed keys and indices are written in bold.*

A rightcut at position i = 6 results in the following tree. The key value 6 at position 1 is the next winner. See Figure 5. A node in the tree consists of two key values, pointer to the parent, the left child and that to right child. The left child and right child of v are denoted by $left(v)$ and $right(v)$. Cutting the left (right) child is to set the pointer to the left (right) child to nil. The parent of node v is denoted by $parent(v)$. The update of the tree takes place along a zig zag path from the bottom towards the root.

Algorithm : right cut //Left cut can be defined symmetrically

Perform binary search from the root for position of i, $pos(i)$, at the bottom;

v = the leaf node representing position i;

if v is the right child of its parent then let v go to the left child;

while v is not the root **do begin**

  while v is a left child of its parent **do begin**

    $c(parent(v)) = c(v)$;

    $v = parent(v)$;

    Cut the right child of v;

  end;

  while v is the right child of its parent **do begin**

    $v = parent(v)$;

    $c(v) = c(max(left(v); right(v)))$;

end

end

/** function max(u, v) is to take the node with maximum *key* **/

If there are n elements in the tree, this algorithm runs in $O(log\ n)$ time and $O(n^2)$ strips are affected. If we use the naive DMM algorithm based on the definition of $min_{k=1}^{n}\{a_{ik} + b_{kj}\}$, we can maintain two tournements for X and Y in each strip, and we can achieve $O(n^3 + kn^2\ log\ n)$ time for our problem. Our final goal is to organize a multi-level tournament based on the sub-cubic algorithm for DMM.

5.3 Modified DMM

Modification of tournament trees for equations (2) and (3) can be done with the technique in the previous section. We describe the modification for equation (4) in this section. We describe the right cut with table look-up. In chapter 3.2, the sorted order of participants are encoded in *table'*. Instead we encode the ranks of participants. We fill the (x, y) entry of *table'*, table'[x, y], by $h(w_1, \dots, w_l)$ with $p = l$, where w_i is the rank of participant i. Note that $(w_1, \dots, w_l)$ is the inverse permutation of $(r_1, \dots, r_l)$, that is, using [.] for indexing, $r[w[i]] = i$. The right cut of $(w_1, \dots, w_l)$ at position k is to give $(w_1, \dots, w_{k-1})$, that is, items at position to the right of k inclusive of k are cut off. The result is returned from the enhanced table with another dimension of k, $T[h(H_1, \dots, H_l), h(L_1, \dots, L_l), k] = h(w_1, \dots, w_{k-1})$, in $O(1)$ time.

Example 2 $m = 5$, $2m = 10$, $h(H) = 456$, 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 saw table'[456, 329] = 231, since H[1, 3] < L[1, 3] in Example 1. Now by the new definition table0[456, 329] = 312, that is. the rank of 1 is 3, etc. If we cut at position 3, we have table'[456, 329, 3] = 21, from which we can retrieve winners 2 and 1 in this order. A simple cut would give 31, which is normalized to 21.

With the right cut operations in Chapter 5.2 and Chapter 5.3, we can perform the modification of the tournaments of X and Y at any coverage ((K, L), (I, J)). Let

the index h be the leftmost index of the hole (not h-function), be given by $h = (L + J)/2 + (i - 1)m + (j - 1)l + k$. Then the portion from position k to the right in the bottom part is cut, the portion from position j to the right is cut and the portion from position i to the right is cut to update Y in this coverage. The left cut operations for X are similarly defined. The cut off operations for one coverage takes $O(log\ n)$ time. After one hole is created by deleting the next maximum subarray, there are $O(n^2)$ strips over affected levels of recursion,. Thus the time for updating tournaments for a next subarray, we spend $O(n^2\ log\ n)$ time, resulting in $O(kn^2\ log\ n)$ time for the k-maximum subarrays.

5.4 Sub-cubic Algorithm

By Algorithm M with modified $X + Y$ problem and modified DMM, we can achieve time complexity for k-disjoint maximum subarray problem of $O(M(n) + kn^2\ log\ n)$ where the $M(n)$ is the time complexity of DMM. If we go down to the single elements by recursion to build the DMM tree in a bottom up fashion ,and naive DMM algorithm is used to build the tree, where $M(\mathrm{n}) = n^3$, then the time complexity becomes $O(n^3 + kn^2\ log\ n)$ for k-disjoint maximum problem. This is the same time complexity in [27].

To use the sub-cubic time algorithm for DMM, we stop dividing s by the four way recursion when the desired submatrix size is achieved. When we do the DMM by divide-and-conquer with table-lookup method, the matrices are divided into (m, m)-submatrices where the size of m is determined by

$$m = log\, n/(\log c\ log\ log\, n) \quad \textit{where c is a constant}$$

Then the time for making the table I of this size is easily shown to be $O(n)$ as described in chapter 3.2 which can be absorbed in the main computing time. Substituting this value of m for $O(n^3/\sqrt{m})$, the time complexity of the DMM, we have the overall computing time for the maximum subarray problem with tree building $O(n^3\sqrt{\log\log n / \log n})$ as claimed. When the submatrix size matches the pre-determined size of m while dividing the matrix s by recursion in

a top-down fashion, we stop the recursion and start to find the maximum subarray for the array *s* building the tournament tree. Then we achieve *k*-disjoint maximum subarray algorithm in time complexity of $O(n^3\sqrt{\log\log n / \log n} + kn^2 log\ n)$ which is sub-cubic time.

5.5 Modified Algorithm M with Space Optimization

The maximum subarray problems described in this paper have the Algorithm M in their cores. It divides the prefix sum array *s* by four-way recursion and calculates the column-centered and row-centered problem in each recursive call by DMM, resulting that the coverage of each DMM matrix for the sub-problems is heavily overlapped. As we see in LEMMA 1 in chapter 2.3, an (*n*, *n*)-matrix takes 16 DMMs on its four (*n*/2, *n*/2)-submatrices to get the column-centered and row centered problems solved in Algorithm M. Now we explain how to reduce this constant by using DMM reuse technique.In the modified Algorithm M, the column-centered and row-centered problems are divided further into three sub-problems as described in [5].

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_{row} be the solution for the row-centered problem.
8. Let the solution A be the maximum of those seven.

Algorithm for the row-centered problem

1. Divide the array into two parts by the vertical center line.
2. Let A_{left} be the solution for the left row-centered problem.
3. Let A_{right} be the solution for the right row-centered problem.
4. Let A_{center} be the solution for the center problem.
5. Let the solution be the maximum of those three.

Algorithm for the column-centered problem

1. Divide the array into two parts by the horizontal center line.
2. Let A_{upper} be the solution for the left row-centered problem.
3. Let A_{lower} be the solution for the right row-centered problem.
4. Let A_{center} be the solution for the center problem.
5. Let the solution be the maximum of those three.

Note that A_{center} is the maximum subarray that crosses over the center point.

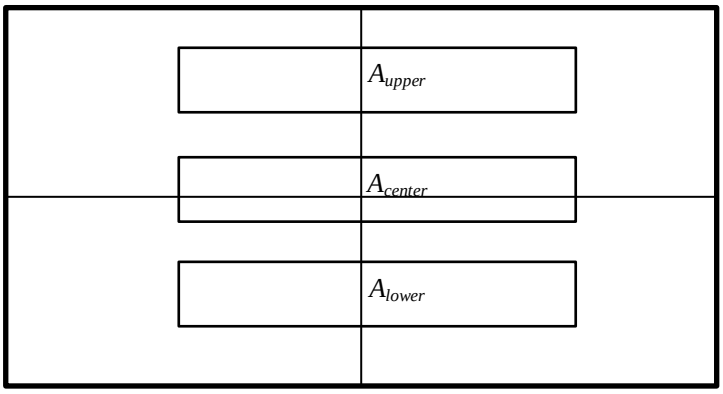


Figure 5.2 *Column-centered problem divided into three sub-problems*

Since the Algorithm M is based on recursion, the column-centered problem and the row-centered problem can be solved more efficiently in this way with DMM reuse technique that is explained in the following section. This optimization is applicable to all maximum subarray problems based on Algorithm M including *k*-overlapping maximum subarray problem and *k*-disjoint maximum subarray problem.

5.6 Reuse of DMM

When we do DMM by divide-and-conquer, if matrices *A*, *B*, and *C* in DMM are divided into (*m*, *m*)-submatrices for $N = n/m$ as follows:

$$\begin{pmatrix} A_{11} & \cdots & A_{1N} \\ \cdots & \cdots & \cdots \\ A_{N1} & \cdots & A_{NN} \end{pmatrix} \begin{pmatrix} B_{11} & \cdots & B_{1N} \\ \cdots & \cdots & \cdots \\ B_{N1} & \cdots & B_{NN} \end{pmatrix} = \begin{pmatrix} C_{11} & \cdots & C_{1N} \\ \cdots & \cdots & \cdots \\ C_{N1} & \cdots & C_{NN} \end{pmatrix}$$

Matrix C can be computed by N^3 multiplications of distance matrices on the submatrices. If there are some of C_{ij} are already calculated, obviously the total number for DMM to get C can be reduced.

When we do DMM at the level 0 in recursive calls on a given (n, n)-matrix in Algorithm M, the DMM matrices for its four (n/2, n/2)-submatrices are already calculated in the previously ending recursive call at level 1. Instead of calculating the DMM matrices for the level 0, the already calculated DMM matrices for its submatrices can be reused by merging them as described below.

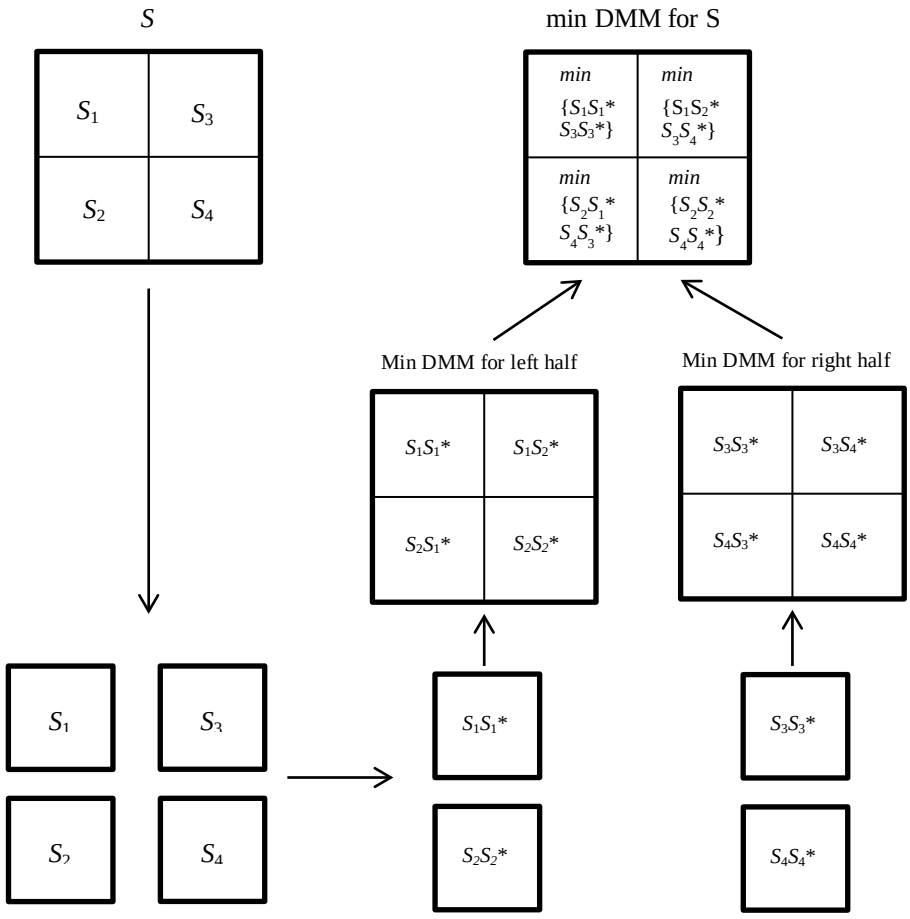


Figure 5.3 *Merging four DMM matrices into one for the upper level. The prefix sum array S is divided into four submatrices by recursion in a top down fashion and the DMM matrices are built up in a bottom-up fashion*

We explain the merging operation for DMM matrices in min version for the column–centered problem. The merging operations for the other matrices are similar. In each level of recursive call, the matrix has two versions of DMM matrix for the column centered-problem of its own coverage. One is min version and the other is max version. Each DMM matrix is built from the DMM matrices of its four submatrices as described in Figure 5.3. The prefix sum matrix S is divided into four submatrices S_1, S_2, S_3 and S_4 by recursion in a top-down fashion and min version of DMM matrices for the column-centered problem are calculated for each of those. For the min DMM Matrix for the upper level's, the precalculated min DMM matrices for S_1 and S_2 are merged first to form the min DMM matrix for the left half of s. The upper left portion and lower right portion of the matrix just copy the corresponding part of the DMM matrix of the children and only the rest parts are newly calculated. For the maximum subarray problem, upper right triangle of DMM matrix is not necessary, only lower left part are desired to be newly calculated. The right half DMM matrix is calculated in the same way. Finally, take the element-wise minimum between the two halves to get the final min DMM matrix for the matrix S.

To solve the column-centered problem for S, min DMM for S_1 is subtracted from the max DMM for S_3 and the take the maximum from the resulting matrix, such as $A_{upper} = max\{S_3S_3^* - S_1S_1^*\}$ where the *max* operation find the maximum value of the matrix. Since the coverage of the resulting matrix is ((1, 1), (n/2, n)), the solution should be bounded in the upper half of s. similarly min DMM for S_2 is subtracted from the max DMM for S_4 to get the lower solution, such as $A_{lower} = max\{S_4S_4^* - S_2S_2^*\}$. Since the coverage of the resulting matrix is ((n/2+1, 1), (n, n)), the solution should be bounded in the lower half of s. Lastly, newly calculated min version of S_2S_1 is subtracted from the also newly calculated max version of S_4S_3 to get the center solution. The row coverage of S_1 and S_3 is [1, … n/2] and the one of S_2 and S_4 is [n/2+1, … , n], so those of $S_4S_3^*$ and $S_2S_1^*$ should be [k, i] where $1 \leq k \leq n/2$, $n/2+1 \leq i \leq n$, which means that the solution $A_{center} = max\{S_4S_3^* - S_2S_1^*\}$ should across the center point. The final column-centered problem is the maximum of those three. The row-centered problem can

be solved in a similar way.

Now let us analyze the time for the work at level 0 as we do in chapter 2.3. We can multiply (n, $n/2$) and ($n/2$, n) matrices by 2 multiplications of size ($n/2$, $n/2$), if we reuse DMM matrices from the children. Furthermore, if we do not calculate unnecessary part of the DMM matrix, the upper right triangle, only one multiplication is needed. 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} and A_{row}, spending $4M(n)$ comparisons. Thus we have the following recurrence for the total time $T(n)$.

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

Note that the constant in front of the $M(n)$ is 16 in chapter 2.3

LEMMA 2 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 $T(n) \leq 4(1 + 4/c)M(n)$.

Clearly the complexity of $O(n^3\sqrt{\log\log n / \log n})$ satisfies the condition of the LEMMA 2 with some constant $c > 0$. Thus the first maximum sum can be solved in $O(n^3\sqrt{\log\log n / \log n})$ time. Since we take the maximum of two matrices component-wise in line 8 of our algorithm and for the 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 4 in front of $M(n)$ in the above recurrence.

6 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. For the disjoint case, the time complexity is $O(M(n) + kn^2 \log n)$. This complexity is sub-cubic when $k = o(n/\log n)$. Since the maximum possibility of k is $O(n^2)$, there is some possibility of improving the bound for k so that the time complexity of the problem remains sub-cubic.

References

[1] R. Agrawal, et al., "Mining association rules between sets of items in large databases," presented at the Proceedings of the 1993 ACM SIGMOD international conference on Management of data, Washington, D.C., United States, 1993.

[2] T. Fukuda, et al., "Data Mining with optimized two-dimensional association rules," ACM Trans. Database Syst., vol. 26, pp. 179-213, 2001.

[3] J. Bentley, "Programming pearls: perspective on performance," Commun. ACM, vol. 27, pp. 1087-1092, 1984.

[4] T. Takaoka, "Efficient Algorithms for the Maximum Subarray Problem by Distance Matrix Multiplication," Electronic Notes in Theoretical Computer Science, vol. 61, pp. 191-200, 2002.

[5] H. Tamaki and T. Tokuyama, "Algorithms for the maximum subarray problem based on matrix multiplication," presented at the Proceedings of the ninth annual ACM-SIAM symposium on Discrete algorithms, San Francisco, California, United States, 1998.

[6] T. Takaoka, "Sub-cubic algorithms for the maximum subarray problem," Proc. Computing:Australasian Theory Sumposium (CATS 2002), pp. 189-198, 2002.

[7] S. E. Bae and T. Takaoka, "Mesh algorithms for the K maximum subarray problem," Proc. ISPAN, pp. 247-253, 2004.

[8] F. Bengtsson and J. Chen, "Efficient Algorithms for Maximum Sums," in Algorithms and Computation. vol. 3341, R. Fleischer and G. Trippen, Eds., ed: Springer Berlin / Heidelberg, 2005, pp. 137-148.

[9] C.-H. Cheng, et al., "Improved Algorithms for the Maximum-Sums Problems," in Algorithms and Computation. vol. 3827, X. Deng and D.-Z. Du, Eds., ed: Springer Berlin / Heidelberg, 2005, pp. 799-808.

[10] F. Bengtsson and J. Chen, "A note on ranking k maximum sums," Research Report 2005:8, 2005.

[11] G. S. Brodal and A. G. Jørgensen, "A Linear Time Algorithm for the k Maximal Sums Problem," private communication, 2007.

[12] S. Bae and T. Takaoka, "Improved Algorithms for the Maximum Subarray Problem for Small k," in Computing and Combinatorics. vol. 3595, L. Wang, Ed., ed: Springer Berlin / Heidelberg, 2005, pp. 621-631.

[13] S. E. Bae and T. Takaoka, "A sub-cubic time algorithm for the k-maximum subarray problem," presented at the Proceedings of the 18th international conference on Algorithms and computation, Sendai, Japan, 2007.

[14] W. L. Ruzzo and M. Tompa, "A Linear Time Algorithm for Finding All Maximal Scoring Subsequences," presented at the Proceedings of the Seventh International Conference on Intelligent Systems for Molecular Biology, 1999.

[15] M. L. Fredman, "New Bounds on the Complexity of the Shortest Path Problem," SIAM Journal on Computing, vol. 5, pp. 83-89, 1976.

[16] T. Takaoka, "A new upper bound on the complexity of the all pairs shortest path problem," Information Processing Letters, vol. 43, pp. 195-199, 1992.

[17] H. Yijie, "Improved algorithm for all pairs shortest paths," Information

Processing Letters, vol. 91, pp. 245-250, 2004.

[18] Y. Han, "An O(n3(loglogn/logn)5/4) Time Algorithm for All Pairs Shortest Path," Algorithmica, vol. 51, pp. 428-434, 2008.

[19] T. Takaoka, "A Faster Algorithm for the All-Pairs Shortest Path Problem and Its Application," in Computing and Combinatorics. vol. 3106, K.-Y. Chwa and J. Munro, Eds., ed: Springer Berlin / Heidelberg, 2004, pp. 278-289.

[20] U. Zwick, "A Slightly Improved Sub-cubic Algorithm for the All Pairs Shortest Paths Problem with Real Edge Lengths," in Algorithms and Computation. vol. 3341, R. Fleischer and G. Trippen, Eds., ed: Springer Berlin / Heidelberg, 2005, pp. 841-843.

[21] T. Takaoka, "An O(n3 log log n/log n) time algorithm for the all-pairs shortest path problem," Information Processing Letters, vol. 96, pp. 155-161, 2005.

[22] T. Chan, "All-Pairs Shortest Paths with Real Weights in O(n3/logn) Time," in Algorithms and Data Structure. vol. 3608, F. Dehne, et al., Eds., ed: Springer Berlin / Heidelberg, 2005, pp. 318-324.

[23] T. M. Chan, "More algorithms for all-pairs shortest paths in weighted graphs," presented at the Proceedings of the thirty-ninth annual ACM symposium on Theory of computing, San Diego, California, USA, 2007.

[24] S. E. Bae, "Sequential and Parallel Algorithms for the Generalized Maximum Subarray Problem," Ph.D, Computer Science and Software Engineering, University of Canterbury, Christchurch, 2007.

[25] T.-C. Lin and D. Lee, "Randomized Algorithm for the Sum Selection Problem," in Algorithms and Computation. vol. 3827, X. Deng and D.-Z.

Du, Eds., ed: Springer Berlin / Heidelberg, 2005, pp. 515-523.

[26] G. N. Frederickson and D. B. Johnson, "The complexity of selection and ranking in X + Y and matrices with sorted columns," Journal of Computer and System Sciences, vol. 24, pp. 197-208, 1982.

[27] S. Bae and T. Takaoka, "Algorithm for K Disjoint Maximum Subarrays," in Computational Science – ICCS 2006. vol. 3991, V. Alexandrov, et al., Eds., ed: Springer Berlin / Heidelberg, 2006, pp. 595-602.