Available online at www.sciencedirect.com

SciVerse ScienceDirect

Procedia Computer Science 9 (2012) 754 – 763

Procedia Computer Science

International Conference on Computational Science, ICCS 2012

Improved algorithms for the *K* overlapping maximum convex sum problem

Mohammed Thaher[a], Tadao Takaoka[a,b,*]

[a,b]*Department of Computer Science and Software Engineering, University of Canterbury, Christchurch 8140, New Zealand*

Abstract

This paper presents efficient algorithms that improve the time complexity of the *K-Overlapping Maximum Convex Sum Problem* (*K-OMCSP*). Previous research has solved this problem by using the *K-tuples* approach in a time complexity of $O(Kn^3)$. In this paper, efficient algorithms based on dynamic programming are derived to improve the time complexity for the overlapping case. The algorithms find the first, second and third maximum convex sum, and up to the K^{th} maximum convex sum in the time complexity of $O(n^3+Kn^2)$ applying a new method which we call *Active Trace Overlapping-Shape* (*ATOS*). In addition, efficient techniques have been developed for designing and implementing the derived algorithms. Moreover, experiments performed to compare the running time for the two methods. The experiments showed that the running time of *ATOS* was faster than the *K-tuples*.

Keywords: Maximum Subarry Problem; Maximum Convex Sum Problem; K-Overlapping Maximum Convex Sum Problem; K-tuples

1. Introduction

Historically, the problem of finding the maximum sums was initially introduced by J. Bentley [1,2]. This problem has two types: one-dimensional (1D) and two-dimensional (2D) versions [1]. The 1D version is called the maximum subsequence problem, and the 2D version is named the *Maximum Subarray Problem* (*MSP*). The *MSP* involves a selection of segments of consecutive array elements that has the largest possible sum compared with all other segments in presented data using the *rectangular shape* [1,2]. The *MSP* can also be a method that gives an accurate trend with respect to associated parameters in vast data.

Since the emergence of *MSP*, subsequent algorithms tackling the problem have been developed to improve the time complexity for 1D and 2D [2, 3, 4, 6]. In 2010, *MSP* took a new turn by using the *convex shape*; the new problem is called the *Maximum Convex Sum Problem (MCSP)*, which returns more precise results by increasing the gain compared to that of *MSP*. After developing *MCSP*, the problem was generalized to find *K* maximum convex sums for the *disjoint case* [7].

Following this, in 2011, an algorithm to find the *K* maximum convex sums by using the *Overlapping case* was developed. The present time complexity to find the *K-OMCSP* is $O(Kn^3)$[8]. Even though this algorithm has been considered an efficient algorithm for the *K-OMCSP*, improved algorithms with better time complexity will be presented in this paper.

MSP and *MCSP* have many potential applications, which include biological sequence analyses [3, 9-16], health sciences [17, 18] and medical applications [7]. An illustrative example is given to facilitate an understanding of *MSP*/*MCSP*. The example involves applying a record of transaction numbers classified by bank customers' annual balance and age. The bank returns and revenue can be increased by finding correlations between existing parameters extracted from the records, such as considering the relationship between annual balance and customers' ages when offering credit cards.

MSP and *MCSP* can be utilized in this example through the use of matrices, where the input parameters are age and annual balance (Figure 1). A matrix element at (i, j) is the number of customers with age i and annual income j. A normalisation of matrices is required to aid in computation. For example, negative and positive numbers can be obtained as a result from the normalisation. A normalisation value can be the overall mean subtracted from each array element. *MSP* and *MCSP* involve

* Corresponding author. Tel.: +64 3 364 2987; fax: +64 3 364 2569.
E-mail address: mohammed.thaher76@gmail.com, tad.takaoka@canterbury.ac.nz.

doi:10.1016/j.procs.2012.04.081

designing and implementing an algorithm to extract values that represent the potential maximum portions, which in the example's case is the group of customers who can be potentially offered credit cards.

Customers Records [*Age Group*] [*Annual Balance*] =

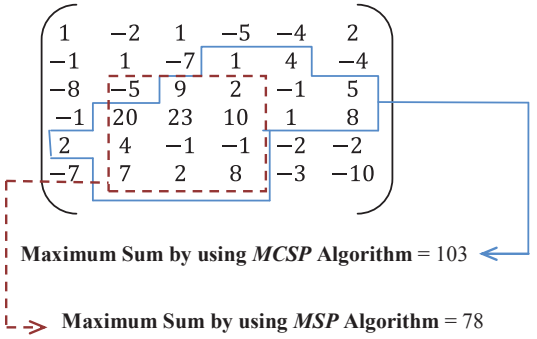


Fig. 1. This matrix has $m \times n$ positive and negative numbers generated randomly. The enclosed portions can be useful for bank marketing purposes.

The designed algorithms that are used to compute the maximum sums in Figure 1, utilised dynamic programming to extract the information portion.

The main contributions of this paper are:

(1) to improve the current time complexity for *K-OMCSP* .
(2) to speed up the running time of new algorithms by using efficient data structures.

The current paper covers the following sections: background; problem definition; implementation of the *convex shape*; improved algorithms to find the *K-OMCSP* for the *convex shapes* $O(n^3+Kn^2)$; results and analysis; conclusions.

2. Background

In 1977, Ulf Grenander from Brown University encountered a problem in pattern recognition [19]. This was answering a research question of finding the maximum sum over all the *rectangular regions* in $n \times n$ array of real numbers using dynamic programming [1, 2]. The *rectangular regions* of the maximum subarray were to be used as the maximum likelihood estimator of a certain kind of pattern in a digitised picture, such as the brightest spot. Grenander developed an $O(n^6)$ time algorithm for an array of size $n \times n$ and found his algorithm was extremely slow [19]. A series of algorithms in engineering design and methods was used to reduce the time complexity in the 1D and 2D versions of the problem [2, 3, 4, 5, 6].

In the field of sequential algorithms, up to 1984, the best solution for *MSP* in 1D array was of $O(n)$ time, whereas the time complexity for finding *MSP* in 2D was $O(n^3)$. Although the 2D's time complexity was best-known as the upper bound, Tamaki and Tokuyama devised a better time complexity that achieved a sub-cubic time of $O(n^3 (\log \log n / \log n)^{1/2})$. They used a divide-and-conquer technique. In addition, they applied the fastest known *Distance Matrix Multiplication* (*DMM*) algorithm by Takaoka [20]. Takaoka designed a simplified version of the algorithm and applied faster *DMM* algorithms to the *MSP* [21].

In 2007, Sung Eun Bae generalized the problem to find the ranking of the maximum sums. He classified the *K Maximum Subarray Problem* (*K-MSP*) into the *K-Overlapping Maximum Subarray Problem* (*K-OMSP*) and the *K-Disjoint Maximum Subarray Problem* (K-DMSP) [22-26]. Moreover, he presented methodologies and techniques to speed up computation time for these two categories. For example, he designed an algorithm of $O(Kn^3)$ for the 2D array and demonstrated a technique to improve the time complexity achieving $O(n^3+K\log n)$. In addition, he developed two mesh algorithms for the 2D *MSP* with $O(n)$ running time that required a network of size $O(n^2)$. However, the algorithms failed to increase the gain or sums because of the *rectangular shape* regions' lack of flexibility.

Fukuda and his colleagues discussed data mining based on association rules for two numeric attributes and one Boolean [27]. They proposed an efficient algorithm for computing the regions that give optimal association rules for gain, support, and confidence. The main aim of their algorithm was to generate 2D association rules that represent the dependence on a pair of numeric attributes. This algorithm was simplified later by Thaher and Takaoka by using the bi-directional approach [7]. In addition, Alan Sprague's research [28] investigated extracting optimal association rules over numerical attributes by using the anchored *convex shape*.

Following Fukuda and his colleagues' works [27], in 2010, Thaher and Takaoka [7] used concepts of the *MCSP* to optimise results obtained from conventional *MSP*. Thaher and Takaoka departed from the method of using the *rectangular region* in the *MSP* by using the *convex shape*. Using the *convex shape* to find the maximum sums gave an accurate result by increasing the sum. Furthermore, their work was extended to compute the maximum sums using the disjoint technique for finding the *K*

maximum convex sums in $O(Kn^3)$ time. In the most recent research, in 2011, they designed and implemented an algorithm to compute the *K-OMCSP* in $O(Kn^3)$ time by using the *K-tuples* approach[8].

3. Problem definition

3.1 Overview

The problem of the *K-OMSP* was first presented in 2004 [22]. Around the same time, Bengttson and Chen also studied the problem independently [29].

A rich collection of publications addressing the problem has been accumulated [22-26, 28-31] and the time complexity of the problem has been increasingly improved using the *rectangular shape* region. However, the *rectangular shape* region is not flexible enough to cover various data distributions.

3.2 Problem definition

We have started from the basic case of 1D array to provide background information in relation to the topic. Our approach is based on a 2D array. For a given array $a[1..n]$ containing positive and negative real numbers and 0, the maximum subarray is the consecutive array elements of the greatest sum. Let *MAX(K, L)* be the operation that selects the K largest elements in a list L in non-increasing order. The definition of K overlapping maximum subarrays is given as follows:

$$R = MAX(K, L), \text{ where } L=\left\{ \sum_{x=i}^{j} [a] \quad | \, 1 \le i \le j \le n \right\} \qquad (1)$$

Here, the K maximum sums are stored in $R[1..K]$. Note that the solution set R is in a sorted order. This sorting was not required, but all other literature on this problem unanimously assumed the order.

Example (1):

Let a = {3, 51, -41, -57, 52, 59, -11, 93, -55, -71, 21, 21}. For this array of size 12, a total of 78(= 12(12+1)/2) subarrays exist. Among them, the first maximum subarray is 193, $a[5]+a[6]+a[7]+a[8]$ if the first element is indexed '1'. We denote this by 193(5, 8). When overlapping is allowed, the second and third maximum subarrays are 149(1, 8) and 146(2, 8).The 78-th maximum subarray, (or the minimum subarray) is −126(9, 10).

In 2D, previous studies used the *rectangular shape* [1-6, 17-26, 28-31]. Latest research presents the problem from different perspective by using the *convex shape* which maximised the sum in [7,8]. The *convex shape* that is currently used is called the *WN-shape*. The best time complexity to find the *K-OMCSP* is $O(Kn^3)$. This is achieved by using the *K-tuples* method (Figure 2) [8].

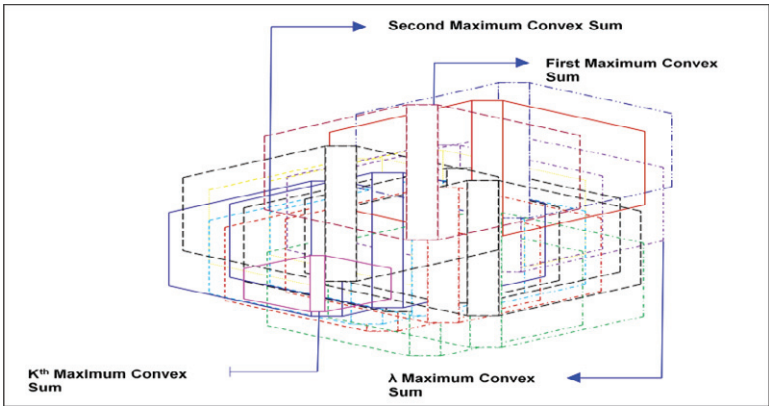


Fig.2. K overlapping convex maximum sums in two dimensions. λ represents one of the maxima before finding the K^{th} overlapping maximum sums

4. Implementation for the *convex shape*

Finding the maximum convex sums in a 2D array involves using a *WN-shape*. In the present paper, this is called the *convex shape* for simplicity. In [27] called rectilinear convex. The *convex shape* combines two shapes. These are the *W-shape* and the *N-shape* - Figure 3 [7].

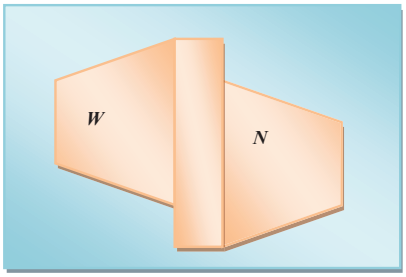


Fig.3. *WN- shape* (*Convex shape*).

Algorithm (1) finding *W-shapes* by using dynamic programming

Algorithm (1) demonstrates finding the *W-shapes*. The iterative formula in line 4 of this algorithm uses dynamic programming to find the solution.

Algorithm (1): finding *W- shape using dynamic programming*

1: $f_W(0, [i,j]) \leftarrow 0$ for all $i < j$
2: for $c \leftarrow 1$ to n do
3: for all intervals of $[i, j]$ in increasing order of j-i where $i < j$ do

4: $$f_W(c, [i,j]) \leftarrow \max \begin{cases} f_W(c\text{-}1, [i,j]) + \text{sum}[c,i,j] & \text{(case 1)} \\ f_W(c, [i\text{+}1,j]) + a[i,c], & \text{(case 2)} \\ f_W(c, [i, j\text{-}1]) + a[j,c] & \text{(case 3)} \end{cases}$$

5: end for
6: end for
// where f_W is a function to find *W-shape* such that $f_W(c,[i,j])$ = the maximum value of the sum of *W-shape* ending from position i to position j in column c.
//sum$[c,i,j]$ = the sum of the c^{th} column from position i to position j used in the first case, and, a$[i,c]$ and a$[j,c]$ = the values added in the second and third cases respectively.

The process of finding the *W- shapes* (Figure 4), which is the left portion of the *convex shape*, involves three cases. This process is conducted in order to select the maximum value from the three cases. The first case is that if column c is being processed, the algorithm needs to check the previous solution up to c-1 to observe if the extension is appropriate for maximising the sum. The second case involves adding the value at (i, c) to the best solution at column c with a narrower interval to obtain more gain to the sum. The third case includes adding the value at (j, c) to the best solution in the c^{th} column with a narrower interval to maximise the sum. The value of f_N is similarly computed from right to left.

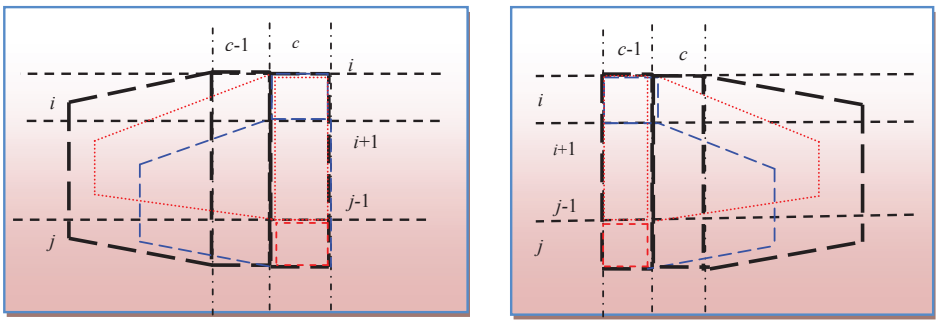


Fig.4. Three cases of *W- shape*; (b) three cases of *N-shape*

Algorithm (2) finding the *Convex Shape*

Based on the results obtained from Algorithm (1), and assuming that sum $[c, i, j]$ is the column sum from i to j of the c^{th} column, we implemented the following algorithm:

Let sum$[c, i, j]$ be the column sum from i to j of the c^{th} column.

Algorithm (2): finding the *Convex Shape*

1: Compute *W-shape* from left to right for each c, i and j in $f_{w,}$ using Algorithm (1).
2: Compute *N-shape* from right to left for each c, i and j, resulting in f_N, using Algorithm 1;
/** Finalisation by combination **/
For $c \leftarrow 1$ to n do
For $i \leftarrow 1$ to n do
For $j \leftarrow i$ to n do
$f_{WN}(c,[i,j]) = f_W(c,[i,j] + f_N(c,[i,j] - \text{sum}[c,i,j]$
3: Take the maximum of $f_{WN}(c,[i,j])$ for all c, i, j

The anchor column of $f_{WN}(c,[i,j])$ represents the array portion from index i to j in the c^{th} column. The involved time complexity for computing the finalisation part, line 3 Algorithm (2), is $O(n^3)$, due to the presence of a triply nested structure.

Algorithm (3): computing the *K-Overlapping Maximum Convex Sum Problem* (*K-OMCSP*) by using the *K-tuples* in $O(Kn^3)$

The *MCSP* has been extended to find the first, second and the third maximum convex sum, and up to the K^{th} maximum convex sum [8]. This extension is to compute the overlapping maximum convex sums up to K^{th}. The overlapping case can be computed by an Algorithm (3) that has a time complexity of $O(Kn^3)$ [8].

The previous work extended Algorithm (2) to include *K-tuples* [8]. In Algorithm (3), instead of returning single values for f_W and f_N the *K-tuples* were used and it was expressed by Fw and F_N. Algorithm (1) was implemented as follows: suppose L is a *K-tuple* $(a_1, a_2, \ldots, a_K)$, for a single value x, $L+x$ is defined by $L+x = (a_1+x, a_2+x, \ldots, a_K+x)$. For the sorted *K-tuples* $L_1, L_2, \ldots, L_m$, $\max\{L_1, L_2, \ldots, L_m\}$ are the largest K numbers in the merged list of $L_1, L_2, \ldots, L_m$ that are in sorted order. In the case of two *K-tuples* L_1 and L_2, L_1+L_2 is the set of the largest K numbers in sorted order from the Cartesian sum for the sums $\{x+y \mid x \text{ is in } L_1 \text{ and } y \text{ is in } L_2\}$[32].

Algorithm (3): computing the *K-Overlapping Maximum Convex Sum Problem* (*K-OMCSP*) by using the *K-tuples* in $O(Kn^3)$

1: $F_W(0, [i,j]) \leftarrow (0, -\infty\ldots, -\infty)$ for all $i < j$ // 0 followed by (K-1) $-\infty$
2: for $c \leftarrow 1$ to n do
3: for all intervals of $[i, j]$ in increasing order of j-i where $i < j$ do

$$F_W(c, [i,j]) \leftarrow \max \begin{cases} F_W(c\text{-}1, [i,j]) + \text{sum}[c,i,j] & \text{(extended case 1)} \\ F_W(c, [i\text{+}1,j]) + \text{a}[i, c], & \text{(extended case 2)} \\ F_W(c, [i, j\text{-}1]) + \text{a}[j, c] & \text{(extended case 3)} \end{cases}$$

F_N is similarly computed from right to left.
/** Finalisation **/
4: For $c \leftarrow 1$ to n do
5: For $i \leftarrow 1$ to n do
6: For $j \leftarrow i$ to n do
$F_{WN}(c,[i,j]) = (F_W(c,[i,j]) + F_N(c,[i,j]) - \text{sum}[c,i,j])$

The anchor column of $F_{WN}(c,[i,j])$ represents the array portion from index i to j in the c^{th} column. i is the top index , j is the bottom index, and c is the column number. F_W and F_N is computed in $O(Kn^3)$ time complexity; this is because the max operation is placed inside a triply nested structure, and the max operation takes $O(K)$ time. Furthermore, in Algorithm (3), finalisation process takes $O(Kn^3)$ time. This is because finding $F_{WN}(c,[i,j]) = (F_W(c,[i,j]) + F_N(c,[i,j]) - \text{sum}[c,i,j])$ is also placed in a triply nested structure and the addition operation, "+", of two *K-tuples* can be calculated in $O(K)$ time [8].

5. Improved algorithms to find the *K-OMCSP* in $O(n^3+Kn^2)$

The previous work to find the *K-OMCSP* achieved a time complexity of $O(Kn^3)$. This approach used the *K-tuples* method, which was previously discussed in detail [8]. The current paper's approach uses a new method, which we call *Active Trace Overlapping-Shape* (*ATOS*).

ATOS is an active search algorithm which searches for up to K possible overlapping *convex shapes* sharing the same anchor column. This algorithm uses data that was collected during the pre-processing of the maximum *convex shape*. The algorithm traces back all the possible overlapping *convex shapes* and sorts them using the tournament tree method, which is based on the rank of the overlapping sums of the *convex shapes*.

The main idea of our preliminary paper was to find the overlapping maximum sum using the *K-tuples* approach to find the *K-OMCSP*. In the *K-tuples*, we repeated the calculation of *K-tuples* every time we needed to find *K-OMCSP*. In Algorithm (3) [8], the finalisation takes $O(Kn^3)$ time. This is because finding $F_{WN}(c,[i,j])= (F_W(c,[i,j]) + F_N(c,[i,j]) - \text{sum}[c,i,j])$ is placed in a triply nested structure and the addition operation, "+", of two *K-tuples* can be calculated in $O(K)$ time. The disadvantage of using the *K-tuples* in *K-OMCSP* is that the *K-tuples* method repeats the calculation of *K-tuples* every time we need to find *K-OMCSP*. This results in an increased running time.

In our new approach, computing *K-OMCSP* has been decomposed into the following steps: firstly, finding the first *convex shape* costs $O(n^3)$. A tournament for the three cases that contribute to finding the final solution is used to select the winner *convex shape* in the anchor column (i,j,c) . In the tournament tree, anchor columns are set up as nodes. Secondly, from the winning path amongst the three cases in the tree, we assume there is a branch from anchor column $[c,i,j]$ to anchor column $[c',i',j']$, where $[c',i',j']$ is determined by the winning case in Algorithm (1). If case (1) is the winning case, for example, $c'=c-1$, $i'=i$ and $j'=j$ (Figure 5), we start by tracing back the winning path of the solution until we reach the source that the solution came from. Thirdly, we set $-\infty$ to the bottom of the winning path solution and updated both the children and parents of the tournament by pushing the winner upwards along the path (Figure 6). In this approach, we keep tracing the previous winning path to find the next winner. Fourthly, we repeated the same process to find the second, third, fourth, and up to the K^{th} sums. Finally, we run a top tournament between different anchor columns and ranking them. The advantage of this approach is that it omits unnecessary repetition, which was used in the previous research, in the process of computing K sums. This approach improves the time complexity for the overlapping case to $O(n^3+Kn^2)$.

5.1. The process of tracing-down and tracing-up

When the first maximum *convex shape* is found, it returns the coordinates *i, j,* and *c* of the anchor column in the input matrix. Here *i* is the top index, *j* is the bottom index, and *c* is the column coordinate of the anchor column with the best combination for the *W-shapes* and *N-shapes*. This process takes $O(n^3)$. Once the coordinates for the first maximum convex sums are found in an anchor column (i,j,c), a node is created which contains four attributes: the position of this node *i, j,* and *c*; the summation up to this node sum; the children's node of this node (child_1, child_2, child_3); and the contributed value by the link of this node to its child (value_1, value_2, value_3). The first node is called the root.

Node Structure
1: *i*, *j*, *c* // the coordinates of this node (i,j,c). *i* is the top index, *j* is the bottom index, and *c* is the column index of the anchor column with the best combination for the *W-shapes* and *N-shapes.*
2: sum // sums up to this node, aw[*i*][*j*][*c*]
3: child_1 // children node based on case 1 from Algorithm (1) which have the same structure
4: child_2 // children node based on case 2 form Algorithm (1) which have the same structure
5: child_3 // children node based on case 3 from Algorithm (1) which have the same structure
6: value_1 // value_1 is aw[*i*][*j*][*c*-1] for *W-shape* , aw[*i*][*j*][*c*+1] for *N-shape*
7: value_2 // value_2 is aw[*i*+1][*j*][*c*]
8: value_3 // value_3 is aw[*i*][*j*-1][*c*]

The process of *trace-down* can be described in Algorithm (4). For the *W-shape*, *trace-up* goes left. For the *N -shape*, it goes right.

Algorithm (4): *trace-down* (node)
1: if node → *c* is not at the boundary
2: createChild(node) // create child
3: path ← getMaxChild(node)
4: // return the child which has the greatest summation
5: // path = child_1 if case 1,
6: // path = child_2 if case 2,
7: // path = child_3 if case 3,
8: path ← -1 if *i* = *j*
9: if path != -1 then trace down(node → path)
10: else (child_2 → sum) ← -∞,
11: (child_3 → sum) ← -∞,
12: trace up(node → parent)
13: end if
14: else (node → sum) ← -∞,
15: trace up(node → parent)
16: end if

Trace-down is a recursive function for any anchor column (*i*,*j*,*c*) which iterates until it traces down to the boundary of the matrix or the source from where the solution at (*i*,*j*,*c*) was generated (Figure 5).

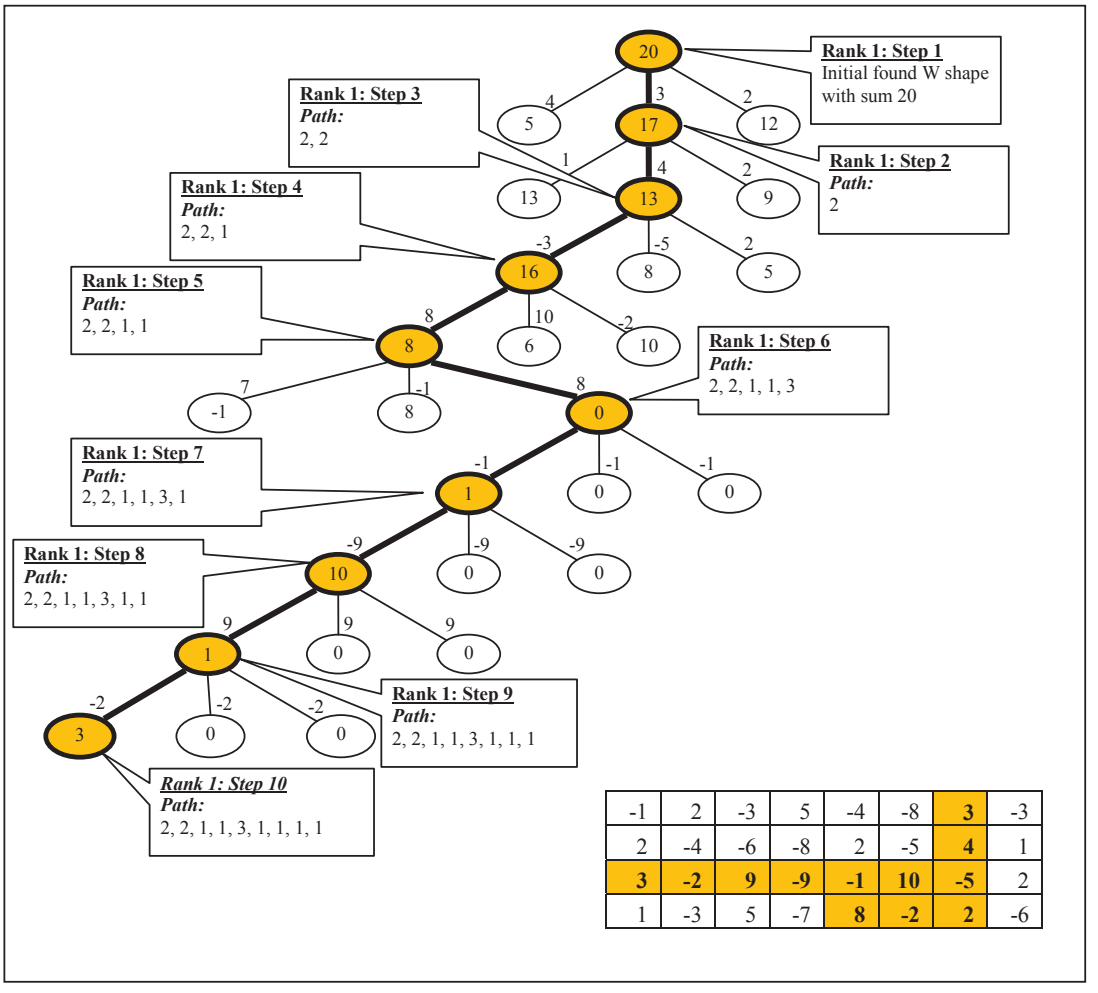


Fig. 5. An example for the *trace-down* process. It shows how we obtained the solution at an anchor column (*i*,*j*,*c*) for the *W-shape*.

The initial state of *trace-down,* which starts from the found anchor column at *i*, *j*, and *c* in Figure 5 is 20. This function traces down its the winning path to find all the choices of cases. The path length is *O(n)*. Thus, the time for *trace-down* is $O(n^2)$ as shape-check (described later) takes *O(n)* time.

Algorithm (5): *trace-up*(node)

1: path ← getMaxChild(node) // (a) return the child which has the greatest sum; (b) (path→ value) = value_1 if case 1; (c) (path→ value) = value_2 if case 2; (d) (path→ value) = value_3 if case 3,
2: path ← -1 if (case2 or case 3) and *i*= *j*
3: if path != -1 then
(node → sum)←(path→ sum)+(path→ value)
4: else (node → sum) ← (child 1 → sum) + value_1
5: if node → parent is not NULL then trace up(node → parent)
6: end if

Once the function traces down and reaches its end, the last node is assigned as negative infinity because the function does not need to use it again. After that, the function calls the *trace-up* function, which is a recursive function that iterates itself until it traces up to its initial state (*i*,*j*,*c*) Figure 6.

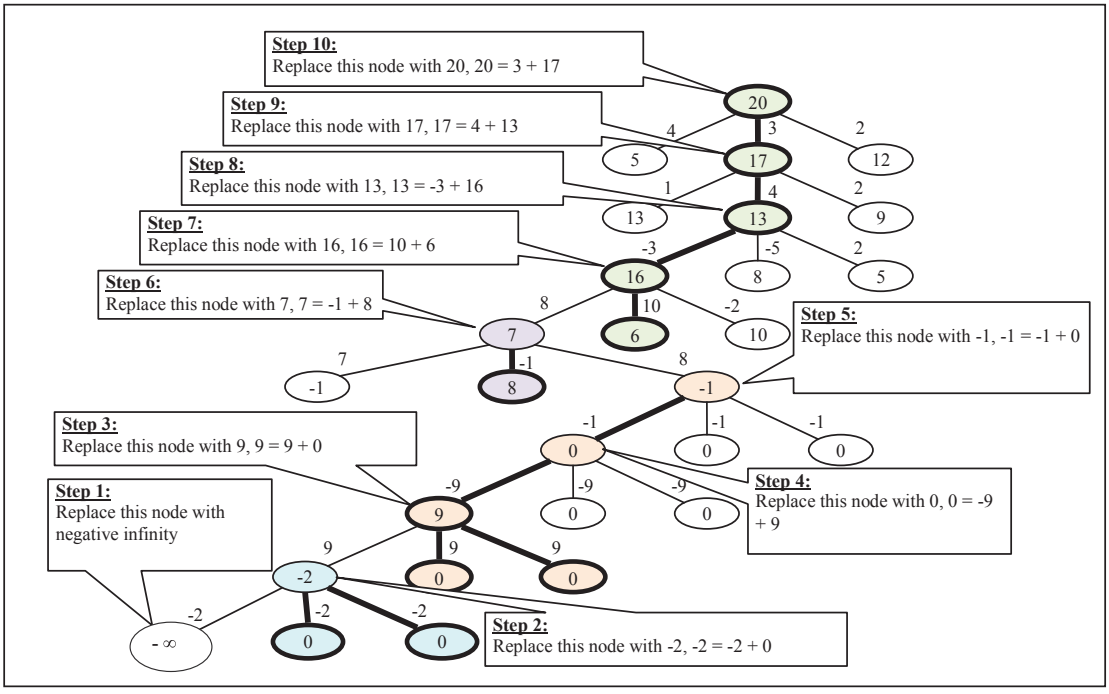


Fig. 6. An example for the *trace-up* process where the child with the greatest sum is pushed up; it updates the sum of the node during the iteration process

During the *trace-up*, the child with the greatest sum is pushed up; it updates the sum of the node during the iteration process. The final sum value is the next greatest sum of the overlapping shapes. After the second sum for the *W-shape* and that for the *N-shape* are obtained for the same anchor column (*i*,*j*,*c*), the second sum for that particular anchor column is obtained. This process is repeated up to the K^{th} sum.

5.2. Operation

All shapes have a unique path and sum. The *trace-down* function finds the path of the elements which contribute to the sum. The sum of the root, or first created node, is the anchor column of the shape which gives the largest sum. The *trace-down* function will find how the path of this sum was formed. In the second line of Algorithm (4), the created child node is retrieved. It creates the children of the node; that is if it does not already have children. The children nodes are the three cases which build-up the shape, and the values are the elements that are added along with the three cases to form the final shape. The *trace-down* process then finds which child node has the largest sum. The child that has the largest sum is the next visited node. This is the path selection where the child with the greatest sum contributes to the final shape.

The *trace-down* process is iterated until it reaches the boundary of the matrix or it finds itself at the sharp corner of the shape where the solution was originated (i.e. when *i* is equal to *j*). When the final node is reached, the sum at that node is replaced with negative infinity. The reason for such an action is to avoid the Algorithm visiting this node again in the future.

Subsequently, the algorithm needs to update the sum of the node for each level. It starts from the last found node and traces itself up to the root. The sum of the node is replaced by the sum of the child which is the greatest among the other children. The updated sum of the root is the next greatest sum of the overlapping *W-shape* based on that anchor column.

ATOS can search for *N*-overlapping shapes, where *N* is the mirror of the *W-shape*. When all the overlapping *W* and *N-shapes* are found, it is necessary to discard repeated shapes and shapes with redundant tails. Repeated shapes are found because the path selection of the *trace-down* function follows the logical decision making as follows:

Algorithm (6): logical decision making
If child_1 is greater or equal to child_2 and child_3 then child_1 is the greatest If child_2 is greater or equal to child_1 and child_3 then child_2 is the greatest If child_3 is greater or equal to child_1 and child_3 then child_3 is the greatest

If all children have the same summation and child_2 is identical to child_3, the algorithm will first process child_1 and then child_2 and then child_3. The program needs to follow the above logic for every greatest sum checking, because different shapes

can make the sum of the final solution. In addition, we can obtain the same sum from different paths. These similar shapes with the same sum need to be removed during the process.

The shape checking process is as follows: only the shapes with the same sum will be checked. The advantage of this technique is to avoid unnecessary checking with the solutions that have different sums. Furthermore, it is apparent that the element of a unique *convex shape* cannot be summed to have different summations. These shapes are sorted into similar categories.

All overlapping shapes are based on the anchor column which is located at *i, j,* and *c* of the matrix. Therefore, it is clear that the overlapping *convex shape* only exists between the *i* and *j* boundaries. Only the indices within the boundaries are checked. This starts from the column that is adjacent to the anchor column, and then checks if both shapes have the same element. If one element exists in one of the shapes, but not in the other, then those shapes are different. The check is done on boundary coordinates in $O(n)$ time. In the worst case scenario, where all overlapping shapes of the same category are unique, the shape needs to be compared with all the unique shapes in the same category to ensure its uniqueness. This process discards the repeated shapes.

5.3. Combining the convex shape (W-shape and N-shape)

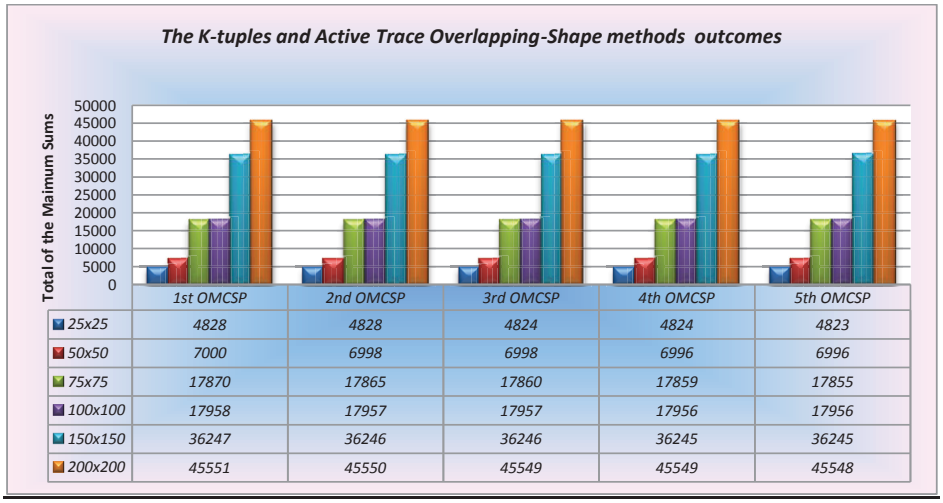
The combinations of the *W-shapes* and *N-shapes* are formed when the ranking for both shapes are computed. To obtain K^{th} rank of the *convex shape*, we process the K^{th} rank of the *W-shapes* and the K^{th} rank of the *N-shapes*. For example, to achieve the best five overlapping shapes, we need to find the best five shapes from the *W-shapes* and the best five from the *N-shapes*. Then, we combine both the *W* and *N-shapes*. This returns 25 combinations; that is, the best *W- shape* is connected with all of the five *N-shapes*.

We do not actually compute K^2 shapes. Each time the next maximum sum is computed, we can finalize the next maximum sum in conjunction with the upper tournament. The upper tournament between the different anchor columns is formed after the $O(n^3)$ maximum sums for all *WN-shapes* are computed. Using this efficient structure, we can compute the next maximum *WN-shape* in $O(n^2)$ time, with $O(\log n)$ for tournament absorbed.

6. Results and analysis

Previous research used *K-tuples* to solve the *K-OMCSP*. The best time complexity was $O(Kn^3)$ [8]. In this paper we showed new techniques to improve the time complexity, achieving $O(n^3+Kn^2)$. This paper's research adds to the existing knowledge in finding the *K-OMSCP* by using the *tracing-down* and *tracing-up* techniques with efficient data structures.

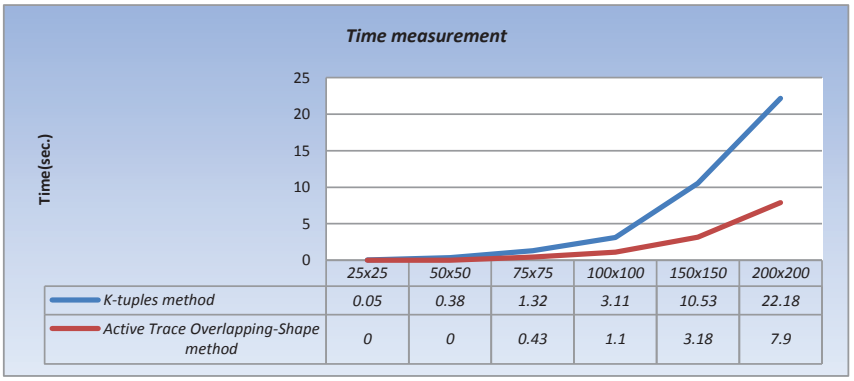
We conducted experiments to compare results obtained from *K-tuple* in algorithm (3) [8] and *tracing-up* and *tracing-down* - algorithms (1), (2), (4), (5) and (6). To compare the algorithms' outcomes for the maximum gain and the running time, the same input matrices were used. The matrix elements were generated by using a random number generator that provides positive and negative numbers. The matrices' sizes range between 25x25 to 200x200. Outcomes from the obtained output show that the new algorithms present the same sum with a better time complexity. These results are depicted in Figure 8.



	1st OMCSP	2nd OMCSP	3rd OMCSP	4th OMCSP	5th OMCSP
25x25	4828	4828	4824	4824	4823
50x50	7000	6998	6998	6996	6996
75x75	17870	17865	17860	17859	17855
100x100	17958	17957	17957	17956	17956
150x150	36247	36246	36246	36245	36245
200x200	45551	45550	45549	45549	45548

Fig. 8. This figure shows the total of the maximum sums in the first five overlapping by using the *K-tuples* method - Algorithm (3). The time complexity of this is $O(Kn^3)$. Whilst the same outcomes result from using the *Active Trace Overlapping-Shape* method - Algorithms (1, 2, 4, 5 and 6), the time complexity of this is less, $O(n^3+Kn^2)$.

The running time for the two methods was measured during the process of finding the first five overlapping convex maximum sums (Figure 9). The graph shows that the *Active Trace Overlapping-Shape* is faster than *K-tuples* when subjected to the same conditions.



	25x25	50x50	75x75	100x100	150x150	200x200
K-tuples method	0.05	0.38	1.32	3.11	10.53	22.18
Active Trace Overlapping-Shape method	0	0	0.43	1.1	3.18	7.9

Fig. 9. illustrates the running time for first five overlapping by using the *Active Trace Overlapping-Shape* and *K-tuples*. The *ATOS* method demonstrates faster algorithms compared with the *K-tuples* algorithm.

7. Conclusions

Recently, the maximum sum problem was extended to the *MCSP* to find the *K-OMCSP* by using the *K-tuples* approach in $O(Kn^3)$. This paper's research adds to the existing knowledge in implementing techniques to improve the time complexity for finding the *K-OMCSP*. The present paper achieved $O(n^3+kn^2)$ time by using *ATOS*. In addition, experiments were performed to compare the two approaches. Our experiments showed that the previous approach of the *K-tuples* and the new method *ATOS* gave the same sum with a better time complexity for *ATOS*. In addition, *ATOS* found the solution with a faster running time.

Acknowledgement

The work on this paper has been an inspiring, often exciting, sometimes challenging, but always interesting experience. It has been made possible by many people, who have supported us. Special thanks to Dr. Sung E. Bae for his valuable research.

References

1. J. Bentley, Programming pearls: algorithm design techniques. *Commun. ACM 27* No. 9 (1984) 865.
2. J. Bentley, Programming pearls: perspective on performance. *Commun. ACM* 27 No. 11 (1984) 1087.
3. S. E. Bae, Sequential and Parallel Algorithms for the Generalized Maximum Subarray Problem. *PhD Thesis*, University of Canterbury, 2007. http://hdl.handle.net/10092/1202.
4. D. Gries, A note on a standard strategy for developing loop invariants and loops. *Science of Computer Programming 2* (1982) 207.
5. D. R. Smith, Applications of a strategy for designing divide-and-conquer algorithms. *Science of Computer Programming 8* (1987) 213.
6. H. Tamaki and T. Tokuyama, Algorithms for the maximum subarray problem based on matrix multiplication. *In Proceedings of the ninth annual ACM-SIAM symposium on Discrete algorithms* (1998) 446.
7. M. Thaher and T. Takaoka, An efficient algorithm for the k maximum convex sums, *ICCS 2010*. *Elservier*, Amsterdam, (2010), 1469.
8. M. Thaher and T. Takaoka, An efficient algorithm for computing the *K*-Overlapping maximum convex sum problem, *ICCS 2011*. *Elservier*, Singapore, (2011), 1288.
9. K. Perumalla and N. Deo, Parallel algorithms for maximum subsequence and maximum subarray, Parallel Process. *Lett. 5* (1995) 367.
10. L. Allison, Longest biased interval and longest non-negative sum interval, *Bioinformatics 19*, (2003), 1294–1295.
11. K.-Y. Chen and K.-M. Chao, Optimal algorithms for locating the longest and shortest segments satisfying a sum or an average constraint, Inform.Process. *Lett. 96*, (2005) 197.
12. T.-H. Fan, S. Lee, H.-I. Lu, T.-S. Tsou, T.-C. Wang and A. Yao, An optimal algorithm for maximum-sum segment and its application in bioinformatics, *In Proceedings of eighth Internat. Conference on Implementation and Application of Automata*, Lecture Notes in Computer Science, vol. 2759, (2003) 251.
13. X. Huang, An algorithm for identifying regions of a DNA sequence that satisfy a content requirement, *Comput. Appl. Biosci*. 10, (1994) 219.
14. Y.-L. Lin, X. Huang, T. Jiang and K.-M. Chao, MAVG: locating non-overlapping maximum average segments in a given sequence, *Bioinformatics 19*, (2003) 151.
15. Y.-L. Lin, T. Jiang and K.-M. Chao, Efficient algorithms for locating the length-constrained heaviest segments with applications to biomolecular sequence analysis, J. *Comput. System Sci*. 65, (2002) 570.
16. L. Wang and Y. Xu, SEGID: identifying interesting segments in (multiple) sequence alignments, *Bioinformatics 19*, (2003) 297.
17. K. Fukuda and T. Takaoka, Analysis of air pollution (PM10) and respiratory morbidity rate using *K*-maximum sub-array (2-D) algorithm, *In Proceedings of the 2007 ACM symposium on Applied computing*, Korea, (2007), 153.
18. K. Fukuda, Computer-enhanced knowledge discovery in environmental science. *PhD Thesis*, University of Canterbury, 2009. http://hdl.handle.net/10092/2140
19. U. Grenander, Pattern Analysis, *Springer*, New York, 1978.
20. H. Tamaki and T. Tokuyama, Algorithms for the maximum subarray problem based on matrix multiplication. *In Proceedings of the ninth annual ACM-SIAM symposium on Discrete algorithms 9th ACM-SIAM* (1998) 446.
21. T. Takaoka, Efficient algorithms for the maximum subarray problem by distance matrix multiplication. *In Electronic Notes in Theoretical Computer Science* vol. 61, *Elsevier*, (2002).
22. S. Bae and T. Takaoka, Algorithms for the problem of k maximum sums and a vlsi algorithm for the k maximum subarrays problem.*In Proc. of the International Symposium on Parallel Architecture Algorithms, and Networks, ISPAN 2004* (2004) 247.
23. S. Bae and T. Takaoka, Improved algorithm for the k-maximum subarray problem for small k. *In Proc. of International Computing and Combinatorics Conference , COCOON 2005* , LNCS 3595 (2005) 621.
24. S. Bae and T. Takaoka. Algorithm for k disjoint maximum subarrays. *In Proc. of the International Conference on Computational Science ICCS 2006*, *Elservie,* Part I (2006) 595.
25. S. Bae and T. Takaoka. Improved algorithms for the k-maximum subarray problem. *Computer Journal 49*, 3 (2006) 358.
26. S. Bae and T. Takaoka. Algorithms for k-disjoint maximum subarrays. *International Journal of Foundations of Computer Science 18*, 2 (2007) 319.
27. T. Fukuda, Y. Morimoto, S. Morishita and T. Tokuyama, Data Mining with optimized two-dimensional association rules**,** *ACM Transactions on Database Systems* (*TODS*), (2001) 179. doi.acm.org/10.1145/383891.383893.
28. A. Sprague, Extracting Optimal Association Rules over Numerical Attributes. Department of Computer and Information Sciences, *University of Alabama at Birmingham*, (1999) 1.
29. F. Bengtsson and J. Chen. Efficient algorithms for the k maximum sums. *ISAAC 2004*, *LNCS*, *Springer* 3341 (2004) 137.
30. F. Bengtsson and J. Chen, Ranking k maximum sums. *Theoretical Computer Science*, 377(2007) 229.
31. C. Cheng, K. Chen, W. Tien, and K. Chao, Improved algorithms for the maximum-sums problems. *Theoretical Computer Science,* 362 (2006) 162.
32. G. Frederickson and D. Johnson. The complexity of selection and ranking in x+y and matrices with sorted rows and columns. *Journal of Computer and System Sciences 24* (1982)197.