Algorithm for K Disjoint Maximum Subarrays

Sung Eun Bae and Tadao Takaoka

Department of Computer Science and Software Engineering
University of Canterbury, Christchurch, New Zealand
{seb43, tad}@cosc.canterbury.ac.nz

Abstract. The maximum subarray problem is to find the array portion that maximizes the sum of array elements in it. For K disjoint maximum subarrays, 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. We first propose an $O(n + K \log K)$ time solution for one-dimension. This is equivalent to Ruzzo and Tompa's when order is considered. Based on this, we achieve $O(n^3 + Kn^2 \log n)$ time for two-dimensions. This is cubic time when $K \leq n/\log n$.

1 Introduction

The maximum subarray problem determines an contiguous array elements that sum to the maximum value with respect to all possible array portions within the input array. When the input array is two-dimensional, we find a rectangular subarray with the largest possible sum.

In the sales database, the maximum subarray problem may be applied to identify certain group of consumers most interested in a particular product. This is also used in the analysis of long genomic DNA sequences to identify a biologically significant portion. In graphics, we can find the brightest area within the image after subtracting the average pixel value from each pixel.

For the one-dimensional case, we have an optimal $O(n)$ time sequential solution, known as Kadane's algorithm [5]. A simple extension of this solution can solve the two-dimensional problem in $O(m^2n)$ time for an $m \times n$ array ($m \leq n$), which is cubic when $m = n$ [6]. The subcubic time algorithm is given by Tamaki and Tokuyama [10], which is further simplified by Takaoka [9].

Finding K maximum sums is a natural extension. We can define two categories depending on whether physical overlapping of solutions is allowed or not.

For K overlapping maximum subarrays, significant improvements have been made since the problem was first discussed in [1] and [3]. Recent development by Cheng et al. [7] and Bengtsson and Chen [4] established an optimal solution of $O(n + K \log K)$ time. For two-dimensions, $O(n^3)$ time is possible [2, 7].

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 [8] finds all disjoint maximum subarrays in $O(n)$ time for one-dimension.

V.N. Alexandrov et al. (Eds.): ICCS 2006, Part I, LNCS 3991, pp. 595–602, 2006.

To the best of Authors' knowledge, little study has been undertaken on this problem for higher dimensions. Particularly, an algorithm for the two-dimensional case may be used to select brightest spots in graphics, and such a technique may be also applied to motion detection and video compression.

In this paper, we discuss the difficulty involved in extending Ruzzo and Tompa's algorithm [8] to two-dimensions and design an alternative algorithm for one-dimension that is more flexible to extend to higher dimensions. Based on the new framework, we present an $O(m^2n + Km^2 \log n)$ time solution for two-dimensions where (m, n) is the size of the input array. This is cubic time when $m = n$ and $K \leq n/\log n$.

2 Problem Definition and Difficulty in Two-Dimensions

2.1 Problem Definition

For a given array $a[1..n]$ containing mixture of positive and negative real numbers, the maximum subarray is the consecutive array elements of the greatest sum. The definition of K disjoint maximum subarrays is given as follows.

Definition 1 (Ruzzo and Tompa [8]). *The k-th maximum subarray is the consecutive subarray that maximizes the sum of array elements* disjoint *from the $(k-1)$ maximum subarrays*

In addition, we impose sorted order on K maximum subarrays.

Definition 2. *The k-th maximum subarray is not greater than the $(k-1)$-th maximum subarray.*

It is possible for a subarray of zero sum adjacent to a subarray of positive sum to create an overlapping subarray with tied sums. To resolve this, we select the one with smaller area if there are subarrays of tied sums. Another subtle problem arises with the value of K. For $k < K$, it is possible that the k-th maximum subarray becomes non-positive. We may stop the process at this point even if the K-th maximum is yet to be found. A non-positive maximum subarray is essentially a single negative array element, which is trivial to find. Let $\bar{K}$ be the maximum number of positive disjoint maximum subarrays. Theoretically $1 \leq \bar{K} \leq n/2$ and is data dependent. Throughout this paper, we assume that K, the number of maximum subarrays we wish to find, is not greater than $\bar{K}$.

Example 1. a ={3,51,-41,-57,52,59,-11,93,-55,-71,21,21}. From the array a, the maximum subarray is 193, $a[5] + a[6] + a[7] + a[8]$ if the index of first element is 1. We denote this by $192(5, 8)$. The second and third maximum subarrays are $54(1, 2)$ and $42(11, 12)$. The fourth is $-41(3, 3)$, so $\bar{K} = 3$.

A trivial solution may be repeated application of Kadane's algorithm [5, 6]. When the first maximum subarray is found in $O(n)$ time, we replace the element values within the solution with $-\infty$. The second and subsequent maximum subarrays are found by repeating this process. This is $O(Kn)$ time. Ruzzo and Tompa's algorithm [8] takes $O(n)$ time for $\bar{K}$ disjoint maximum subarrays, but requires sorting if Definition 2 needs to be met.

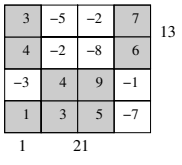
Algorithm 1. Maximum subarray for one-dimension

1: If the array becomes one element, return its value.
2: Let M_{left} be the solution for the left half.
3: Let M_{right} be the solution for the right half.
4: Let M_{center} be the solution for the center problem.
5: $M \leftarrow \max\{M_{left}, M_{right}, M_{center}\}$.

2.2 Problems in Two-Dimensions

For an (m, n) array, $a[1..m][1..n]$, we wish to find K disjoint maximum subarrays which are in rectangular shape. We denote a subarray of sum x with coordinates of top-left corner (r_1, c_1) and bottom-right corner (r_2, c_2) by $x(r_1, c_1)|(r_2, c_2)$. In the following example, we compute $K = 4$ disjoint maximum subarrays.

Example 2.



For $K = 4$, K disjoint maximum subarrays are 21(3,2)|(4,3), 13(1,4)|(2,4), 7(1,1)|(2,1) and 1(4,1)|(4,1).

In this example, $\bar{K} = 4$. When $K > 4$, the subsequent subarrays will be comprised of a single negative array element such as $-1(3,4)|(3,4)$, $-2(1,3)|(1,3)$ etc.

As is in one-dimension, a trivial solution for finding K disjoint maximum subarrays is repeated application of Kadane's algorithm. This is $O(Km^2n)$ time or $O(Kn^3)$ time for $m = n$. In the worst case, where $\bar{K} = n^2/2$, we have $O(n^5)$ time for $K = \bar{K}$.

For more efficient solution, it is natural to consider extending Ruzzo and Tompa's algorithm [8]. While we omit the details of this algorithm, it seems difficult to extend to two-dimensions as we have to organize the scanning in two directions such that the rectangular subarray may be found.

In the following section, we present another algorithm for one-dimension. This algorithm provides solid framework to extend to the two-dimensional case.

3 One-Dimensional Case

For a one-dimensional array $a[1..n]$, we compute the prefix sum s such that $s[x] = \sum_{i=1}^{x} a[i]$. We assume $s[0] = 0$.

Algorithm 1 shows the outer framework. In this algorithm, the center problem is to obtain an array portion that crosses over the central point with maximum sum, and can be solved in the following way. Note that the prefix sums once computed are used throughout recursion. We assume that n is power of 2 without loss of generality.

$$M_{center} = \max_{\substack{n/2 \le i \le n \\ 0 \le j < n/2}} \{s[i] - s[j]\} = \max_{n/2 \le i \le n} \{s[i]\} - \min_{0 \le j < n/2} \{s[j]\} \qquad (1)$$

The recursive computation of this algorithm can be conceived as a tournament-like selection process, which we describe in the following.

Algorithm 2. Build tournament for $a[f..t]$

procedure buildtree($node, f, t$) **begin**
1: $(from, to) \leftarrow (f, t)$
2: **if** $from = to$ **then**
3: $(L, M, G) \leftarrow (s[f-1], a[f], s[f])$ // *node* is a leaf
4: **else** // *node* is an internal node
5: create two children $left$ and $right$
6: buildtree($left$,f,$\frac{f+t}{2} - 1$) //build left subtree
7: buildtree($right$, $\frac{f+t}{2}$,t)) //build right subtree
8: $L \leftarrow \min\{L_{left}, L_{right}\}$, $R \leftarrow \max\{G_{left}, G_{right}\}$
9: $M \leftarrow \max\{M_{left}, M_{center}, M_{right}\}$ where $M_{center} \leftarrow G_{right} - L_{left}$
end

3.1 Tournament

We construct a binary tree bottom-up where each node contains the following attributes.

- ($from$,to): the coverage, i.e., the range of array elements covered by this node.
- L: the minimum prefix sum within $(from - 1, to - 1)$. Abbreviates "least".
- G: the maximum prefix sum within the coverage. Abbreviates "greatest".
- M: the maximum sum found within the coverage. Refer to Lemma 1.
- (noL, noG): boolean values initially both $false$. Discussed in Section 3.2.

We denote the left child of an internal node x by x_{left} and the right child by x_{right}. Variables of the child node are given with subscript such as L_{left}, meaning that L of x_{left}. Throughout this paper, we call this tree the *tournament*, or simply T. The root of T will be referred to as $root(T)$.

Based on Algorithm 1, we design Algorithm 2 that recursively builds T. Note that the computation of M_{center} at line 9 is due to Eq. 1. After *buildtree(root,1,n)* is processed, the value of M at $root(T)$ is the maximum sum in the array $a[1..n]$.

We let each L, G and M carry two indices such as $M.from$ and $M.to$ to indicate that M is the sum of array elements $a[M.from]..a[M.to]$. If M_{center} is chosen for M, $M.from = L_{left}.to + 1$ and $M.to = G_{right}.to$. The following two facts are easily observed. Proofs are omitted.

Lemma 1. *When a node x has coverage $(from, to)$, its M is the maximum subarray inside this coverage. i.e., $from \leq M.from \leq M.to \leq to$.*

Lemma 2. *The maximum subarray of $a[1..n]$ is M at $root(T)$.*

When there is a tie during computation, such as $L_{left} = L_{right}$, we select the one that will result in M with smaller physical size. For example, when $L_{left} = L_{right}$, we select L_{right} as M_{center} will have smaller physical size by subtracting L_{right}. Similarly, when $G_{left} = G_{right}$, we select G_{left}. For the same reason, we choose the one with smaller physical size in computing $M = \max\{M_{left}, M_{right}, M_{center}\}$.

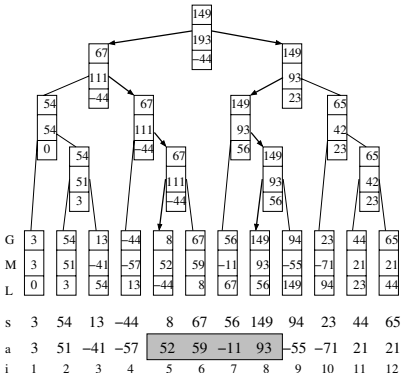


Fig. 1. Tournament T

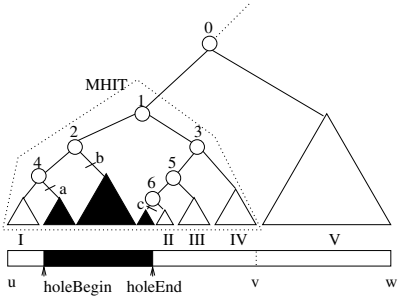


Fig. 2. Subarray deletion

Fig. 1 shows the example in Section 2.1 computed by the tournament. Each node shows a 3-tuple of (L, M, G). The value of M at root, 193 represents the maximum sum. The figure omits the location ($M.from$,$M.to$) which is $(5, 8)$.

3.2 Delete a Subarray

We discuss how we delete a subarray from the tournament, so that the tree will produce a maximum subarray that is disjoint from the deleted portion. In the following description, we use a term *hole* to refer to the portion to be deleted or has been deleted.

With the index *holeBegin* and *holeEnd*, the location of the hole, we trace the tree from the root to find subtrees whose coverage is inside the hole. In Fig. 2, dark subtrees are those to be deleted. These subtrees can be deleted by removing the link *a*,*b* and *c*. Since we only delete the link, deleting each subtree takes $O(1)$ time. Actual memory deallocation will be done during the post-processing.

After deletion is done, there are nodes that no longer have two children. Such nodes include 2,4,6. When a node has one child missing, we assume that this node receives a 3-tuple $(\infty, -\infty, -\infty)$ from the missing child.

The maximum subarray in the range of (u, v) is determined by node 1. We want it to be disjoint from the hole. If M_{center} becomes M at node 1, we have $(M.from, M.to) = (L_{left}.to + 1, G_{right}.to)$. This M is a "superarray" of the hole as it covers the hole. In general, a node with coverage that encompasses the whole range of the hole can "potentially" produce M_{center} overlapping the hole as M_{center} becomes a superarray of the hole. Node 0 is another node that has such potential, however, if L_{left} comes from region II,III or IV, M_{center} at node 0 can be disjoint from the hole. So we can not simply disable computing M_{center} at such nodes. We resolve this issue by Algorithm 3. The objective of the following algorithm is to ensure that M_{center} is a subarray disjoint from the hole. If no subarray disjoint from the hole can be obtained for M_{center}, we set $M_{center} = -\infty$ to represent no value.

Algorithm 3. Update tournament T

Recursively trace from $root(T)$ downwards the hole, and update each node x.

1: **if** *holeEnd* is in left subtree **then** $noG \leftarrow$true
2: **if** *holeBegin* is in right subtree **then** $noL \leftarrow$true
//Recursively update (L, M, G)
3: **if** x_{left} was deleted **then** $(L_{left}, M_{left}, G_{left}) \leftarrow (\infty, -\infty, -\infty)$
4: **if** x_{right} was deleted **then** $(L_{right}, M_{right}, G_{right}) \leftarrow (\infty, -\infty, -\infty)$
5: **if** noL **then** $L \leftarrow L_{right}$ **else** $L \leftarrow \min\{L_{left}, L_{right}\}$
6: **if** noG **then** $G \leftarrow G_{left}$ **else** $G \leftarrow \max\{G_{left}, G_{right}\}$
7: $M \leftarrow \max\{M_{left}, M_{right}, M_{center}\}$, where $M_{center} \leftarrow G_{right} - L_{left}$

When a node has the flag noL set, it means that this node will not use L_{left} for updating L. Similarly, noG means G_{right} is not used for updating G. However, L_{left} and G_{right} are still used to compute M_{center} regardless of the flags. Basically these flags block propagating L_{left} and G_{right} to the parent node.

If we delete the hole $(5, 8)$ from Fig. 1, and update T as described above, the second maximum subarray $54(1, 2)$ will be obtained from the root.

We propose the following lemma holds. If we use the *minimum hole inclusive tree* (MHIT), the smallest subtree that contains the hole (as shown in Fig. 2), as the basis, it can be inductively proved. We omit the proof due to limited space.

Lemma 3. *After deleting the hole and applying Algorithm 3, $root(T)$ produces maximum subarray M that is disjoint from the hole.*

3.3 Analysis

We find the first maximum subarray by building T in $O(n)$ time. To compute the next maximum subarray, we regard the previous solution as a hole and perform the deletion and flag updates as described in Section 3.2. Deleting nodes involves traversing two paths from the root to *holeBegin* and *holeEnd*. Since the height of the tree is $O(\log n)$, the time for the next maximum subarray is bounded by $O(\log n)$. To obtain K disjoint maximum subarrays in sorted order, the total time is therefore $O(n + K \log n)$. Note that $O(n + K \log n) = O(n + K \log K)$ for any integer K according to [4, 7]. For ranking K disjoint maximum subarrays, this is equivalent to Ruzzo and Tompa's algorithm [8] which requires extra time for sorting.

4 Two-Dimensional Case

In this section, we extend algorithm for one-dimension to two-dimensions.

4.1 Strip Separation

An easy way to extend an algorithm for the one-dimensional case to two-dimensions is *strip separation* technique, such that the two-dimensional array $a[1..m]$

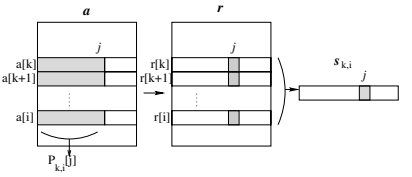


Fig. 3. Separating strips from a two-dimensions

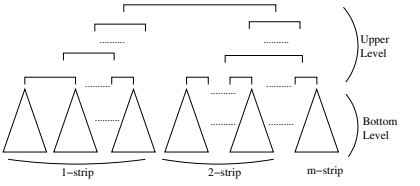


Fig. 4. Two-level tournament

$[1..n]$ is separated into $m(m+1)/2$ strips. As shown in Fig. 3, a strip $s_{k,i}$ is the prefix sum array of a portion $P_{k,i}[1..n]$. We call a strip consisting of x rows a $x-strip$.

We pre-process the row-wise prefix sum $r[1..m][1..n]$, such that $r[i][j] = a[i][1] + a[i][2] + .. + a[i][j]$ in $O(mn)$ time. Then $s_{k,i}[j]$ is computed by $r[k][j] + ..r[i][j]$. The time for strip separation is $O(m^2 n)$.

4.2 Two-Level Tournament

We organize a two-level tournament as shown in Fig. 4. The bottom-level is composed of $O(m^2)$ tournaments of each strip. A tournament of $s_{k,i}$ has $M_{k,i}$, the maximum sum of the strip, at the root. Since each strip is regarded as a one dimensional problem, $M_{k,i}$ is given as $x(l, j)$, from which we obtain the original rectangular subarray $x(k, l)|(i, j)$. The upper-level is a tournament where all $M_{k,i}$ of bottom-level tournaments participate. There are $O(m^2)$ participants. The winner of the upper-level is the maximum subarray for two-dimensions. As building a tournament is linear time, we spend $O(m^2 n)$ time for the bottom-level tournaments, and $O(m^2)$ time for the upper-level.

4.3 Next Maximum

Suppose $x^*(k^*, l^*)|(i^*, j^*)$ is selected for the first maximum by the upper-level. Let us consider a strip $s_{k,i}$. If its row-wise coverage (k, i) is disjoint from (k^*, i^*), this strip definitely produces $M_{k,i}$ disjoint from the first maximum. Otherwise, it is possible that this strip produces an overlapping $M_{k,i}$. There are $O(m^2)$ such strips. By creating a hole (l^*, j^*) in tournaments of such a strip and updating the tree as per Section 3.2, we ensure that all $M_{k,i}$ are disjoint from $(k^*, l^*)|(i^*, j^*)$.

Now that all winners of each bottom-level tournament have become disjoint from the first maximum, we are safe to re-build the upper-level to select the second maximum. Subsequent disjoint maximum subarrays are found by repeating these steps. Each maximum subarray is computed by $O(m^2 \log n)$ time overlap removal and $O(m^2)$ time upper-level re-build. The latter time is absorbed into the former. For K maxima, it is $O(Km^2 \log n)$ time.

Including the time for initial setting, the total time is $O(m^2 n + Km^2 \log n)$. For $K \leq \frac{n}{\log n}$, we have a cubic time $O(m^2 n)$.

5 Concluding Remarks

In this paper, we established $O(n + K \log K)$ time for ranking K disjoint maximum subarrays in a one-dimensional array and extend this to two-dimensions to achieve $O(m^2 n)$ time for small K. To Authors' knowledge, this is the first improvement to the trivial $O(Km^2 n)$ time solution. Since $\bar{K}$, the maximum possible K, can be as large as $mn/2$ depending on the data, reduction of the factor K is significant. It will be an interesting question to determine $\bar{K}$ in advance.

In the current form of our algorithm, in fact, the upper-level does not require a tournament. Linear time maximum selection algorithm can be used instead without increasing the complexity. It is, however, expected that the two-level tournament may provide a solid structural platform for further improvement.

The second term of the complexity, $O(Km^2 \log n)$, seems possible to improve. Currently, we update $O(m^2)$ bottom-level tournament trees on computation of each maximum subarray. If $M_{k,i}$ of a bottom-level tournament is no longer positive, we may discard such a tree to reduce the size of the bottom-level. By doing so, if $\bar{K} = mn/2$, only $O(m)$ bottom-level tournaments of 1-strip will remain. Also if a hole (l^*, j^*) has been already created in a bottom-level tournament in the previous computation, we may skip creating it again. As no more than $O(n)$ holes can be made in each strip, this will result in the second term not exceeding $O(m^2 n \log n)$ even if $K > n$. If such an idea is incorporated, the overall complexity may be reduced to $O(m^2 n + \min(K, n) \cdot m^2 \log n)$.

References

1. Bae, S.E., Takaoka, T.: Algorithms for the problem of K maximum sums and a VLSI algorithm for the K maximum subarrays problem. ISPAN 2004, Hong Kong, 10-12 May, (2004) 247–253. IEEE Comp.Soc. Press.
2. Bae, S.E., Takaoka, T.: Improved algorithms for the K-maximum subarray problem for small K. COCOON 2005, Kunming,China, 16-19 Aug. (2005) 621–631
3. Bengtsson, F., Chen, J.: Efficient algorithms for the k maximum sums. ISAAC 2004, Hong Kong, 20-22 Dec.,(2004) 137–148. Springer-Verlag
4. Bengtsson, F., Chen, J.: A note on ranking k maximum sums. research report 2005:08 (2005) Luleå University of Technology
5. Bentley, J.: Programming pearls: algorithm design techniques. Commun. ACM, Vol.**27(9)** (1984) 865–873.
6. Bentley, J.: Programming pearls: perspective on performance. Commun. ACM, Vol.**27(11)** (1984) 1087–1092.
7. Cheng, C.H., Chen, K.Y., Tien, W.C., Chao, K.M.: Improved algorithms for the k maximum sums problems. ISAAC 2005, Sanya, Hainam, China, 19-21 Dec.(2005) 799–808.
8. Ruzzo, W.L., Tompa, M.: A linear time algorithm for finding all maximal scoring subsequences. ISMB'99, Heidelberg, Germany, 6-10 Aug.(1999) 234–241.
9. Takaoka, T.: Efficient algorithms for the maximum subarray problem by distance matrix multiplication. Elec. Notes in Theoretical Comp. Sci., Vol.**61** (2002) Elsevier
10. Tamaki, H., Tokuyama, T.: Algorithms for the maximum subarray problem based on matrix multiplication. SODA 1998, San Francisco, 25-27 Jan. (1998) 446–452.