CATS 2002 Preliminary Version

Efficient Algorithms for the Maximum Subarray Problem by Distance Matrix Multiplication

Tadao Takaoka

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

Abstract

We design an efficient algorithm that maximizes the sum of array elements of a subarray of a two-dimensional array. The solution can be used to find the most promising array portion that correlates two parameters involved in data, such as ages and income for the amount of sales per some period. The previous subcubic time algorithm is simplified, and the time complexity is improved for the worst case. We also give a more practical algorithm whose expected time is better than the worst case time.

1 Introduction

Suppose we have a CD-ROM which has the record of monthly sales of some commodity, classified by the ages and income levels of purchasers. Then the record items of sales amounts can be stretched over a two-dimensional array, where each row and column corresponds to an age and an income level. The maximum subarray problem is to find an array portion of rectangular shape that maximizes the sum of array elements in it. This sort of data mining methods are described in [6] and [1]. Since the array elements are all non-negative, the obvious solution is the whole array. If we subtract the mean of the array elements from each array element, and consider the modified maximum subarray problem, we can have more accurate estimation on the sales trends with respect to some age groups and some income levels. The same problem can be used in graphics as well. If we subtract the mean values from the each pixel value in a grey-scale graphic image, we can identify the brightest portion in the image.

Thus we deal with the maximum subarray problem where array elements take real numbers, positive, 0, or negative, with at least one positive and one negative, and the total sum is 0. This problem was originated by Bentley [4] and improved by Tamaki and Tokuyama [10]. Bentley's algorithm is cubic and the Tamaki-Tokuyama algorithm is sub-cubic for a nearly square array. Their algorithm [10] is heavily recursive and complicated. We simplify the latter algorithm [10], and achieve sub-cubic time for any rectangular array.

We also give a more practical algorithm whose expected time is close to quadratic for a wide range of random data.

2 Review of distance matrix multiplication

The distance matrix multiplication is to compute the following distance product $C = AB$ for two n-dimensional matrices $A = [a_{i,j}]$ and $B = [b_{i,j}]$ whose elements are real numbers.

$$c_{i,j} = min_{k=1}^{n}\{a_{i,k} + b_{k,j}\}$$

The best known algorithm for this problem is by Takaoka [8], the computing time of which is $O(n^3(\log\log n/\log n)^{1/2})$. The meaning of $c_{i,j}$ is the shortest distance from vertex i in the first layer to vertex j in the third layer in an acyclic directed graph consisting of three layers of vertices, which are labelled 1, ...,n in each layer, and the distance from i in the first layer to j in the second layer is $a_{i,j}$ and that from i in the second layer to j in the third layer is $b_{i,j}$. If we replace the above *min* operation by *max*, we can define a similar product, where we have longest distances in the above three layered graph. The algorithm can be tailored to this version easily by symmetric considerations. We refer to the original multiplication and the algorithm as the *min* version, and those with *max* as the *max* version.

Assume m divides n. Then we can multiply an (m, n) matrix A and (n, m) matrix B in the following way. Divide A and B into n/m square matrices of dimension m, A_1, ..., $A_{n/m}$ and B_1, ..., $B_{n/m}$. The matrix $C = AB$ is computed by

$$C = min\{A_1B_1, ..., A_{n/m}B_{n/m}\}$$

In the above *min* chooses the minimum component-wise over n/m product matrices. Then the time is

$$O(m^3(\log\log m/\log m)^{1/2})(n/m)) = O(m^2n(\log\log m/\log m)^{1/2})$$

3 Maximum subarray problem

We give a two-dimensional array $a[1..m, 1..n]$ as input data. The maximum subarray problem is to maximize the array portion $a[k..i, l..j]$, that is, to obtain such indices (k, l) and (i, j). We suppose the upper-left corner has co-ordinate (1,1).

Example 3.1 *Let a be given by*

```
-1   2  -3   5  -4  -8   3  -3
 2  -4  -6  -8   2  -5   4   1
 3  -2   9  -9  |3   6| -5   2
 1  -3   5  -7  |8  -2|  2  -6
```

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

We assume that $m \leq n$ without loss of generality. We also assume that m and n are powers of 2. We will mention the general case of m and n later. Bentley's algorithm finds the maximum subarray in $O(m^2n)$ time, which is defined to be cubic in this paper. He introduces Kadane's algorithm for the one-dimensional case, whose time is linear. In the following, s is the sum of a tentative maximum subarray $a[k..l]$. The algorithm accumulates a partial sum in t and replace s by t and updates the position (k, l), when t becomes better than s. If t becomes negative, we reset the accumulation.

```
**Kadane's algorithm** /* maximum subarray $a[k..l]$ of $a[1..n]$ */
$(k, l) := (0, 0); s := -\infty;$ $t := 0;$ $j := 1;$
**for** $i$:=1 **to** $n$ **do begin**
    $t := t + a[i];$
    **if** $t > s$ **then begin** $(k, l) := (j, i);$ $s := t$ **end**;
    **if** $t < 0$ **then begin** $t := 0;$ $j := i + 1$ **end**
**end**
```

Unfortunately Kadane's idea does not work for the two-dimensional case. Tamaki and Tokuyama's algorithm solves the problem in subcubic time when the given array is nearly square, that is, $m = O(n)$. We review their algorithm first. Let us divide the array into four parts by the central vertical and horizontal lines. We call the upper-left, upper-right, lower-left, and lower right part the NW (north-west), NE, SW, and SE parts. We define the three conditional solutions for the problem. The first is the maximum subarray that crosses over the center, denoted by A_{center}. This problem is called the the center problem. The second is to cross the horizontal center line, denoted by A_{row}. This problem is called the row-centered problem. The third is to cross the vertical center line, denoted by A_{column}. This problem is called the column-centered problem. The algorithm is roughly described in a recursive manner as follows:

Main algorithm

If the array becomes one dimensional, horizontal or vertical, solve the problem by Kadane's algorithm in linear time. Otherwise
Let A_{NW} be the solution for the NW part.
Let A_{NE} be the solution for the NE part.

Let A_{SW} be the solution for the SW part.
Let A_{SE} be the solution for the SE part.
Let A_{row} be the solution for the row-centered problem.
Let A_{column} be the solution for the column-centered problem.
Let the solution be the maximum subarray of those six.

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

Algorithm for the column-centered problem
Divide the array into two parts by the horizontal center line.
Let A_{upper} be the solution for the upper column-centered problem.
Let A_{lower} be the solution for the lower column-centered problem.
Let A_{center} be the solution for the center problem.
Let the solution be the maximum of those three.

Let $T(m,n)$ be the computing time for the whole problem. Let $T_{row}(m,n)$, $T_{column}(m,n)$, and $T_{center}(m,n)$ be the time for the row-centered, column-centered, and center problem respectively. We have the following recurrence for those time functions, where $T_{center}(m,n)$ is counted twice for simplicity.

$$T(m,1) = O(m),\ T(1,n) = O(n)$$
$$T(m,n) = 4T(m/2,n/2) + T_{row}(m,n) + T_{column}(m,n)$$
$$T_{row}(m,n) = T_{center}(m,n) + 2T_{row}(m/2,n)$$
$$T_{column}(m,n) = T_{center}(m,n) + 2T_{column}(m,n/2)$$

Now the center problem can be solved in the following way. Let the partial sums of array elements from the center point towards the north-west, north-east, south-west, and south-east directions be $S_{NW}[i,j]$, $S_{NE}[i,j]$, $S_{SW}[i,j]$, and $S_{SE}[i,j]$ respectively. For example, $S_{NW}[i,j]$ is the sum of the array portion $a[i..m/2, j..n/2]$. Those partial sums for all i and j can be computed in $O(mn)$ time. Then A_{center} can be computed by

$$max_{i=1,j=1,k=m/2+1,l=n/2+1}^{m/2,n/2,m,n}\{S_{NW}[i,j] + S_{SW}[k,j] + S_{NE}[i,l] + S_{SE}[k,l]\}$$

If we fix i and k, the maximum of the sums of the former two terms and that of the sums of the latter two terms with respect to the suffices j and l respectively can be characterized by distance matrix multiplication of max version. Note that we need to transpose S_{SW} and S_{SE} to fit them into distance matrix multiplication. Thus the problem can be solved in $O(M(m,n))$ time, where $M(m,n)$ is the time for multiplying distance matrices of size (m,n) and (n,m). From this we see the time $T_{center}(m,n)$ can be given by $T_{center}(m,n) =$

$O(M(m, n))$. It is shown that the solution of the recurrence equation is given by

$$T(m, n) = O(m^2 n(\log\log m/\log m)^{1/2} \log(n/m)).$$

We can assume $m \leq n$ for all sub-problems, as we can rotate the problem 90 degrees if this condition does not hold. Intuitively speaking, the last factor $\log(n/m)$ in the above comes from the recursion in the column-centered problem where the recursion proceeds until n becomes equal to m. The computation for the partial sums requires $O(mn \log n)$ time. We improve these complexities in the next section. Specifically, we will go with single recursion, rather than double recursion.

4 New algorithm

The central algorithmic concept in this section is that of prefix sum. The prefix sums $sum[1..n]$ of a one-dimensional array $a[1..n]$ is computed by

```
$sum[0] := 0$;
**for** $i := 1$ **to** $n$ **do** $sum[i] := sum[i-1] + a[i]$;
```

This algorithm can be extended to two dimensions with linear time, the details of which are omitted.

We use distance matrix multiplications of both *min* and *max* versions in this section. We compute the partial sums $s[i, j]$ for array portions of $a[1..i, 1..j]$ for all i and j with boundary condition $s[i, 0] = s[0, j] = 0$. These sums are often called the two-dimensional prefix sums of a. Obviously this can be done in $O(mn)$ time. We show that the column-centered problem can be solved without recursion. The outer framework of the algorithm is given below. Note that we need not handle the one-dimensional problem here, and that the prefix sums once computed are used throughout recursion.

Main algorithm
If the array becomes one element, return its value.
Otherwise, if $m > n$, rotate the array 90 degrees.
Thus we assume $m \leq n$.
Let A_{left} be the solution for the left half.
Let A_{right} be the solution for the right half.
Let A_{column} be the solution for the column-centered problem.
Let the solution be the maximum of those three.

Now the column-centered problem can be solved in the following way.

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

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

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

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

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

The first part in the above is distance matrix multiplication of the *min* version and the second part is of the *max* version. Let S_1 and S_2 be matrices whose (i,j) elements are $s[i,j-1]$ and $s[i,j+n/2]$. For an arbitrary matrix T, let T^* be that obtained by negating and transposing T. Then the above can be computed by multiplying S_1 and S_1^* by the *min* version and taking the lower triangle, multiplying S_2 and S_2^* by the *max* version and taking the lower triangle, and finally subtracting the former from the latter and taking the maximum from the triangle.

5 Analysis

Let us assume that m and n are each a power of 2, and $m \leq n$. As we saw in Section 2, we can go to the case where $m = n$ by chopping the array into squares. Thus we analyze the time $T(n)$ for the (n,n) array. We observe the algorithm splits the array vertically and then horizontally. We can multiply $(n, n/2)$ and $(n/2, n)$ matrices by 4 multiplications of size $(n/2, n/2)$. We analyze the number of comparisons. The rest is proportional to this. Let $M(n)$ be the time for multiplying two $(n/2, n/2)$ matrices. Thus we have the following recurrence.

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

Theorem 5.1 *Let c be an arbitrary constant such that $c > 0$. Suppose $M(n)$ satisfies the condition*

$$M(n) \geq (4+c)M(n/2).$$

Then the above $T(n)$ satisfies

$$T(n) \leq 12(1+4/c)M(n).$$

Proof. The condition on $M(n)$ means that its asymptotic growth ratio is more than n^2. Theorem holds for $T(1)$ from the algorithm. Now assume theorem holds for $T(n/2)$ for induction. Then

$$\begin{aligned} T(n) &= 4T(n/2) + 12M(n) \\ &= 48(1+4/c)M(n/2) + 12M(n) \\ &\leq 48(1+4/c)/(4+c)M(n) + 12M(n) \\ &= 12(1+4/c)M(n). \end{aligned}$$

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

Now suppose one or both of m and n are not given by powers 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.

6 Further speed-up on average

In this section we modify the Moffat-Takaoka algorithm [7] for the all pairs shortest path problem to be used as a fast engine for distance matrix multiplication. Let us use the three layered DAG described in section 2.

Fast distance matrix multiplication
Let $A = [a_{i,j}]$ and $B = [b_{i,j}]$ be the two distance matrices. Let $C = AB$ be the distance product. In the following, we represent suffices by brackets.

First sort the rows of B in increasing order. Using the sorted lists of indices $list[k]$, solve n single source problems by the Moffat-Takaoka algorithm, where the solution set, to which shortest paths from each source have been established, is defined on the vertices in the third layer. To solve the single source problem from source i in the first layer, let each vertex k in the second layer have its candidate $cand[k]$ in the third layer, which is the first element in $list[k]$ initially. Organize $\{k|k = 1, ..., n\}$ into a priority queue by the keys $d[k] = a[i, k] + b[k, cand[k]]$. We repeat deleting v with the minimum key from the queue, and put $cand[v]$ into the solution set. We scan $list[v]$ to get a clean candidate for v, that is, $cand[v]$ outside the solution set S. Then we put v back to the queue with the new key value. After the solution set is expanded by one, we scan the lists for other w such that $cand[w] = cand[v]$ to make their candidates $cand[w]$ clean. The key values are changed accordingly and candidates change their positions in the queue. We stop this expansion process of the solution set at the critical point where the size is $n - n/\log n$. Let U be the solution set at this stage and not changed thereafter. After the critical point, we further expand the solution set to n in a similar fashion, but will be satisfied with candidates outside U, that is, half clean. The algorithm follows.

```
Sort $n$ rows of $B$ and let the sorted list of indices be $list[1]$, ..., $list[n]$;
Let $V = \{1, ..., n\}$;
**for** $i := 1$ **to** $n$ **do begin**
```

```
  **for** $k := 1$ **to** $n$ **do begin**
    $cand[k]$:=first of $list[k]$;
    $d[k] := a[i,k] + b[k, cand[k]]$;
  **end**;
  Organize set $V$ into a priority queue with keys $d[1], ..., d[n]$;
  Let the solution set $S$ be empty;
  /* Phase 1 : Before the critical point */
  **while** $|S| \leq n - n \log n$ **do begin**
    Find $v$ with the minimum key from the queue;
    Put $cand[v]$ into $S$;
    $c[i, cand[v]] := d[v]$;
    Let $W = \{w | cand[w] = cand[v]\}$;
    **for** $w$ in $W$ **do**
      **while** $cand[w]$ is in $S$ **do** $cand[w]$:= next of $list[w]$;
    Reorganize the queue for $W$ with the new keys $d[w] = a[i,w]+b[w, cand[w]]$;
  **end**;
  $U := S$;
  /* Phase 2 : After the critical point */
  **while** $|S| < n$ **do begin**
    Find $v$ with the minimum key from the queue;
    **if** $cand[v]$ is not in $S$ **then begin**
      Put $cand[v]$ into $S$;
      $c[v, cand[v]] := d[v]$;
      Let $W = \{w | cand[w] = cand[v]\}$;
    **end else** $W = \{v\}$;
    **for** $w$ in $W$ **do**
      $cand[w]$:=next of $list[w]$;
      **while** $cand[w]$ is in $U$ **do** $cand[w]$:= next of $list[w]$;
    Reorganize the queue for $W$ with the new keys $d[w] = a[i,w]+b[w, cand[w]]$;
  **end**;
**end**.
```

In [7], a binary heap is used for the priority queue, and the reorganization of the heap is done for W in a bottom-up fashion. The expected time for reorganization is shown to be $O(n/(n-j) + \log n)$, when $|S| = j$. Then the time for reorganization is bounded by $O(\log n)$, since $|S| \leq n - n \log n$. Thus the effort for the queue reorganization in phase 1 is $O(n \log n)$ in total.

By the coupon collector's problem in [5], we need to collect $O(m \log m)$ coupons on average before we get m different kinds of coupons. The expected number of unsuccessful trials before we get $|S| = n$ after the critical point is bounded by $O(n)$ by setting $m = n/\log n$. The expected size of W in phase 2 is $O(\log n)$ when S is expanded, and 1 when it is not expanded. In phase 2, we slightly change the queue reorganization. That is, we perform increase-key separately for each w, spending $O(\log n)$ time per $cand(w)$. From these facts, the expected time for the queue reorganization in phase 2 can be shown to be $O(n \log n)$.

We focused on heap operations. Although we omit the details, we can show that the scanning efforts to get clean candidates in phase 1 and half clean candidates in phase 2 are both $O(n \log n)$. From these observations we conclude the complexities before and after the critical point are balanced to be $O(n \log n)$, resulting in the total expected tine of $O(n \log n)$.

The expected time for the n single source problems becomes $O(n^2 \log n)$, including the time for sorting.

Analysis: The endpoint independence is assumed on the lists $list[k]$, that is, when we scan the list, any vertex can appear with equal probability. Let T_1, ..., T_N be the times for all distance matrix multiplications used in the algorithm in this section. Then, ignoring some overhead time between distance matrix multiplications, we have for the expected value $E[T]$ of the total time T

$$E[T] = E[T_1 + ... + T_N] = E[T_1] + ... + E[T_N].$$

From the theorem of total expectation, we have $E[E[X|Y]] = E[X]$ where $X|Y$ is the conditional random variable of X conditioned by Y. The first E in the left hand side goes over the sample space of Y and the second over that of X. In our analysis, X can represent a particular T_i and Y the rest. Thus, if T_i is the time of distance matrix multiplication for (n, n) matrices, we have $E[T_i] = O(n^2 \log n)$. In the analysis of the last section, we chose c to be a constant. In this section, we set c to $4/\log n$. Then we can show $E[T(n)] = O(n^2 \log^2 n)$ for the square maximum subarray problem, and $E[T(m, n)] = O(mn \log^2 m)$ for the rectangular problem of size (m, n), where $T(n)$ and $T(m, n)$ are the computing times of the corresponding problems.

The above analysis hinges on the endpoint independence, which holds for the distance matrix multiplication with prefix sums for a wide variety of probability distribution on the original data array. For simplicity, let us take a one-dimensional array given by $a[1], ..., a[n]$. Let $s[i] = a[1] + ... + a[i]$. For $i < j$, we have $s[j] - s[i] = a[i + 1] + ... + a[j]$. Let us assume $a[i]$ are independent random variables with $prob\{a[i] > 0\} = 1/2$. Then we have $prob\{a[i + 1] + ... + a[j] > 0\} = 1/2$ and thus $prob\{s[i] < s[j]\} = 1/2$. Hence we have any permutation of $s[1], ..., s[n]$ with equal probability of $1/n!$, if we sort them in increasing or decreasing order. If we extend this argument to two dimensions, we can say we have the endpoint independence, if each $a[i, j]$ has independent distribution with average value of 0. In practice, the algorithm in this section is expected to run faster for large m and n in many applications.

7 Concluding remarks

The gap between the trivial lower bound of $\Omega(mn)$ and the worst-case complexity in this paper is still large, as the latter is close to cubic. The algorithms for the all pairs shortest paths for a graph with small integer edge costs by [2] and [9] cut deeply into the sub-cubic complexity. Unfortunately these algorithms can not be used here, since the magnitude of elements in the prefix sums or partial sums become too large. On the other hand, there is an efficient algorithm for the all pairs shortest path problem exists [7]. Using this algorithm, we showed the maximum subarray problem can be solved much faster on average. Note that any fast algorithm for distance matrix multiplication can be used in the maximum subarray problem for an accelerating engine. Finally the author expresses his thanks to reviewers whose comments greatly improved the quality of the paper.

References

[1] Agrawal, R, T. Imielinski, and A. Swami, Mining Association Rules between Sets of Items in Large Databases, Proc. SIGMOD Conf. on Management of Data (1993) 207-216

[2] Alon, N, Z. Galil, and O. Margalit, On the exponent of the all pairs shortest path problem, Proc. 32nd FOCS (1991) 569-575

[3] Bentley, J., Programming Pearls - Algorithm Design Techniques, Comm. ACM, 27-9 (1984) 865-871

[4] Bentley, J., Programming Pearls - Perspective on Performance, Comm. ACM, 27 (1984) 1087-1092

[5] Feller, W, An Introduction to Probability Theory and its Applications, Volume 1, John Wiley and Sons (1950)

[6] Fukuda, T., Y. Morimoto, S. Morishita, and T. Tokuyama, Data Mining Using Two-Dimensional Optimized Association rules: Scheme, Algorithms and Visualization, Proc. SIGMOD Conf. on Management of Data (1996) 13-23

[7] Moffat, A. and T. Takaoka, An all pairs shortest path algorithm with $O(n^2 \log n)$ expected time, SIAM Jour. Computing, 16 (1987) 1023-1031

[8] Takaoka, T., A New Upper Bound on the complexity of the all pairs shortest path problem, Info. Proc. Lett., 43 (1992) 195-199

[9] Takaoka, T, Subcubic cost algorithms for the all pairs shortest path problem, Algorithmica, 20 (1998) 309-318

[10] Tamaki, H. and T. Tokuyama, Algorithms for the Maximum Subarray Problem Based on Matrix Multiplication, Proceedings of the 9th SODA (Symposium on Discrete Algorithms) (1998) 446-452