An $O(n^3 \log\log n/\log n)$ Time Algorithm for the All-Pairs Shortest Path Problem

Tadao Takaoka
Department of Computer Science
University of Canterbury
Christchurch, New Zealand

August 2004

Abstract

We design a faster algorithm for the all-pairs shortest path problem under the conventional RAM model, based on distance matrix multiplication (DMM). Specifically we improve the best known time complexity of $O(n^3(\log\log n)^2/\log n)$ to $O(n^3 \log\log n/\log n)$. As an application, we show the k-maximum subarray problem can be solved in $O(kn^3 \log\log n/\log n)$ time for small k.

1 Introduction

In this paper we consider the all-pairs shortest path (APSP) problem, which computes shortest paths between all pairs of vertices of a directed graph with non-negative real numbers as edge costs. We present an algorithm that computes shortest distances between all pairs of vertices, since shortest paths can be computed as by-products in our algorithm. It is well known that the time complexity of (n, n)-distance matrix multiplication (DMM) is asymptotically equal to that of the APSP problem for a graph with n vertices. Thus we concentrate on DMM in this paper. The computational model in this paper is the conventional RAM, where only arithmetic operations, branching operations, and random accessibility with $O(\log n)$ bits are allowed.

Fredman [4] was the first to break the cubic complexity of $O(n^3)$ under RAM, giving $O(n^3(\log\log n/\log n)^{1/3})$. This complexity was improved to $O(n^3(\log\log n/\log n)^{1/2})$ by Takaoka [6], and recently to $O(n^3(\log\log n)^2/\log n)$ [9]. In this paper we improve the complexity further to $O(n^3 \log\log n/\log n)$. The above mentioned complexities are all in the worst case. If we go to the average case, we can solve the APSP problem in $O(n^2 \log n)$ time [5]. If edge costs are small integers, the complexity becomes more subcubic, i.e., $O(n^{3-\epsilon})$ for some $\epsilon > 0$, as shown in [7], [1], and [11].

We follow the same framework as those in [4], [6], and [9]. That is, we take a two level divide-and-conquer approach. To multiply the small matrices resulting from dividing the original matrices, we sort distance data, and use the ranks of those data in the sorted lists.

As the ranks are small integers, the multiplication can be done efficiently by looking at some precomputed tables.

In Section 2, we introduce basic techniques for computing the minimum of a list of small integers through simulated tournament packed in single words. In Section 3, we describe the two level divide-and-conquer approach. In Section 4, we show how to use ranks to compute DMM. In Section 5, we show how to compute the tables for the efficient multiplication. In Section 6, we show how to compute DMM by table look-up. In Section 7, we analyze the computing time. In Section 8, we generalize DMM to find k minima in the product. In Section 9, we show how to use DMM and its variant to the k-maximum subarray problem. Section 10 concludes the paper.

2 Minimum selection from a small list of small integers

To prepare for distance matrix multiplication, we introduce a basic technique for minimum selection in encoded form in this section. The technique is *parallelism in single words*.

A sequential algorithm takes $O(n)$ time to select the minimum of a list of integers where the size of the list is n and the magnitude of integers is up to the maximum that can be contained in single computer words. We show that it takes $O(\log m)$ time for the same problem where the size of the list is m and the magnitudes of integers are polynomial of m, and $m = O(\log n)$. We assume a pre-computed table of size $O(n)$ is available, which can be constructed in $O(n)$ time..

For simplicity we assume m is a power of 2. We can generalize our theory without changing the orders of complexities. We go through tournament for participants 1, 2, ..., m with corresponding distinct keys $k(1)$, ..., $k(m)$. If $k(i) < k(i+1)$ for even i, i is chosen as a winner for the next stage, and $i+1$, otherwise. We keep track of winners as well as their keys for later application. The size and magnitude are so small that the list of winners and the list of their keys can be encoded into single integers. The one-to-one mappings for encoding are denoted by h_1 and h_2 respectively. At the 0-th stage, all of (1, 2, ..., m) are winners, encoded as $h_1(1, 2, ..., m) = (1-1)m^{m-1} + (2-1)m^{m-2} + ... + (m-1)$. Their keys are encoded into $h_2(k(1), ..., k(m)) = k(1)\mu^{m-1} + ... + k(m)$, where $\mu = m^p$ for some $p > 0$. When there are l winners $(x_1, ..., x_l)$ in general, h_1 and h_2 are defined by $h_1(x_1, x_2, ..., x_l) = (x_1 - 1)m^{l-1} + (x_2 - 1)m^{l-2} + ... + (x_l - 1)$, and $h_2(k(x_1), ..., k(x_l)) = k(x_1)\mu^{l-1} + ... + k(x_l)$, where we omit superscripts to specify the number of arguments in h_1 and h_2. We assume those encoded values are stored in single words. The table T^l is constructed as follows:

$$T^l(h_1(x_1, ..., x_l), h_2(k(x_1), ..., k(x_l))) = h_1(y_1, ..., y_{l/2})$$
$$y_i = x_{2i} \text{ if } k(x_{2i}) < k(x_{2i+1}), y_i = x_{2i+1}, \text{ otherwise.}$$

We prepare another mapping $f^l(h_1(x_1, ..., x_l), h_2(k(1), ..., k(m))) = h_2(k(x_1), ..., k(x_l))$. For general discussions, we omit superscripts. T is a mapping from key values to winners, and f is a mapping from winners to key values, both in the form of encoded integers. Repeated use of T and f will finalize the winner for the whole tournament in $O(\log m)$ time. As winners and keys are distinct integers, mappings T and f are partial functions of the form of $Z \times Z \to Z$, where Z is the set of positive integers. The domain for which function values are not defined can be filled with any value.

EXAMPLE 1 *Participants and corrresponding keys are traced in unencoded form as follows:*
(1, 2, 3, 4, 5, 6, 7, 8) (12, 4, 7, 16, 5, 9, 8, 2)
(2, 3, 5, 8) (4, 7, 5, 2)
(2, 8) (4, 2)
(8) (2), 8 is the final winner with key 2

Let key values be bounded by m^p for some positive constant p. The size of the table T^l is bounded by $m^l(m^p)^l{=}m^l m^{pl}$ and it takes $O(m)$ time to compute each table entry, resulting in $O(m^{l(p+1)}m)$ time. Now we prepare mapping f. The size of the table is $m^l m^{pm}$, and the time for one entry is $O(m)$, resulting in $O(m^l m^{pm} m) \leq O(m^m m^{pm} m)$ time for the whole table. The times for table construction are further multiplied by m as there are m tables for each l. Those times are shown to be $O(c^{m \log m})$ for some constant $c > 0$, which is further shown to be $O(n)$ when $m{=}O(\log n/(\log c \log\log n))$.

Let $\mathbf{x} = (1, 2, ..., m)$, and $\mathbf{k} = (key(1), ..., key(m))$. If the tables T and f as shown above are precomputed in $O(n)$ time, and one problem instance is given by the encoded form of $h_1(\mathbf{x})$ and $h_2(\mathbf{k})$, we can choose the minimum of m keys in $O(\log m)$ time. We call this method the Tournament.

Now we show that we can compute the minimum of $m \log m$ keys in $O(\log m)$ time with tables precomputed in $O(n)$ time. Let $\mathbf{x}$ and $\mathbf{k}$ be $m \log m$ participants and their keys. We sample m participants and corresponding keys at $\log m$ intervals, and encode them. Specifically we prepare table W that maps two sets of m arguments, $\mathbf{x}$ and $\mathbf{y}$, and a set of keys $\mathbf{k}$ in encoded form to m arguments $\mathbf{z}$ in encoded form as follows:

$$W(h_1(\mathbf{x}), h_1(\mathbf{y}), h_2(\mathbf{k})) = h_1(\mathbf{z}), \text{ where}$$
$$z_i = x_i, \text{ if } key(x_i) < key(y_i),\ z_i = y_i, \text{ otherwise.}$$

By going through $\log m$ samples, using W and f alternately, we can compute the minima of m intervals of size $\log m$ in encoded form. We assume that those m samples are available in encoded form in advance. We call this part the preliminary part. From that point on, we can use the Tournament method described above to finalize the minimum. We call this method, that is, the preliminary part followed by the Tournament, the Better Tournament. The time for constructing W's can be shown to be $O(n)$ with $m = O(\log n/(\log c \log\log n))$ for different c. If we take smaller m, we have $O(\log m)$ time for the Better Tournament for $m \log m$ participants.

In the above descriptions, we ignored encoding time. In later sectons we will see this is a reasonable assumption as all the work for table construction and encoding are done at the beginning, which will be shared by subproblems, and absorbed in the main complexity of DMM.

3 Distance matrix multiplication by the Better Tournament

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

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

The operation in the right-hand side of (1) is called distance matrix multiplication of *min* version, and A and B are called distance matrices in this context. If we use *max* instead we call it *max* version.

Now we divide A, B, and C into (m, m)-submatrices for $N = n/m$ as follows:

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

Matrix C can be computed by

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

where the product of submatrices is defined similarly to (1) and the "min" operation is defined on submatrices by taking the "min" operation componentwise. Since comparisons and additions of distances are performed in a pair, we omit counting the number of additions for measurement of the 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. Then the total time excluding table construction is given by $O(n^3/m + (n/m)^3 T(m))$.

In [9], it is shown that $T(m) = O(m^2(m \log m)^{1/2})$ with $m = O(\log n/(\log \log m)^3)$ by the Tournament. Thus the time becomes $O(n^3(\log m/m)^{1/2})$. In the subsequent sections, we show m can be greater by the Better Tournament.

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

$$C = min_{k=1}^{M}\{A_k B_k\} \ (3)$$

The values of m and l were determined in [9], using the Tournament, to achieve the claimed complexity.

4 How to multiply rectangular matrices

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

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

We assume that the lists of length m, $(a_{1r} - a_{1s}, ..., a_{mr} - a_{ms}), (1 \leq r < s \leq l)$, and $(b_{s1} - b_{r1}, ..., b_{sm} - b_{rm}), (1 \leq r < s \leq l)$ are already sorted for all r and s such that $1 \leq r < s \leq l$. The time for sorting will be mentioned in Section 7. 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}. Let H_{rs} be the list of ranks of $a_{ir} - a_{is}$ $(i = 1, ..., m)$ in G_{rs} and L_{rs} be the list of ranks of $b_{sj} - b_{rj}$ $(j = 1, ..., m)$ in G_{rs}. Let $H_{rs}[i]$ and $L_{rs}[j]$ be the ith and jth components of H_{rs} and L_{rs} respectively. Then we have

$$G_{rs}[H_{rs}[i]] = a_{ir} - a_{is},\ G_{rs}[L_{rs}[j]] = b_{sj} - b_{rj}$$

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

We have the following obvious equivalence.

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

Fredman [4] observed that the information of ordering for all i, j, r, and s in the rightmost side of the above formula is sufficient to determine the product AB by a precomputed table. This information is essentially packed in the three dimensional space of $H_{rs}[i](i = 1, ..., m; r = 1, ..., l; s = r + 1, ..., l)$, and $L_{rs}[j](j = 1, ..., m; r = 1, ..., l; s = r + 1, ..., l)$. We call this the three-dimensional packing.

Takaoka [6] proposed that to compute each (i, j) element of AB, it is enough to know the above ordering for all r and s. We call this the two-dimensional packing. Note that the precomputed table must be obtained within the total time requirement. The two-dimensional packing will therefore allow a larger size of m, leading to a speed-up.

In [9], it is shown that a one-dimensional packing scheme is possible by the Tournament, resulting in the complexity of $O(n^3(\log\log n)^2/\log n)$.

We first describe the Tournament, and then proceed to the Better Tournament. We extend the tournament schemes in Section 2 with two sets of keys H's and L's. To choose the minimum of

$$x_r^0 = a_{ir} + b_{rj}, (r = 1, ..., l)$$

we go through the Tournament. Later we will initialize $x_r^0, (r = 1, ..., l)$ differently for the Better Tournamnet. We assume m and l are a power of 2. Our theory can be generalized into other cases easily. Specifically we compare in the following pairs. This is the 0th comparison stage.

$$(x_1^0, x_2^0), (x_3^0, x_4^0), ..., (x_{l-1}^0, x_l^0)$$

Suppose the minima of those pairs, that is, winners, are $x_1^1, x_2^1, ..., x_{l/2}^1$. Each x_j^1 has two possibilities of being x_{2j-1}^0 or x_{2j}^0. Then we compare in the following pairs. This is the 1st comparison stage.

$$(x_1^1, x_2^1), (x_3^1, x_4^1), ..., (x_{l/2-1}^1, x_{l/2}^1)$$

H	1	2	3	4	5	6	7	8
1		1	13*	5	6	7	3	4
2			12	4	5	8	2	5
3				5	6	1#	10	7
4					2	4	5	1
5						12	3	9
6							4*	8
7								3
8								

L	1	2	3	4	5	6	7	8
1		11	3*	2	7	4	5	6
2				4	1	5	6	7	8
3					15	3	8#	1	5
4						11	3	7	14
5							10	5	6
6								7*	9
7									10
8									

Figure 1: $H_{rs}[i]$ and $L_{rs}[j]$ for $l = 8$

The winner x_j^2 of (x_{2j-1}^1, x_{2j}^1) has four possiblities of being x_i^0 for $4j-3 \leq i \leq 4j$. By repeating these stages, we can finish in $\log l - 1$ stages. Comparing in the pair (x_r^0, x_{r+1}^0), that is, testing "$x_r^0 \leq x_{r+1}^0$?" is equivalent to comparing in the pair $(H_{r,r+1}[i], L_{r,r+1}[j])$. Thus if we pack two tuples $(H_{12}[i], H_{34}[i], ..., H_{l-1,l}[i])$ and $(L_{12}[j], L_{34}[j], ..., L_{l-1,l}[j])$ into single integers, we can know the $l/2$ winners in $O(1)$ time from a precomputed table in the form of an encoded integer. We can take a similar approach for stages 1, 2, ... Thus the time for computing (4) becomes $O(\log l)$. Since the ranks are between 1 and $2m$, and there are up to l remaining winners, the size of each table is bounded by $m^l(2m)^{l/2}(2m)^{l/2} = m^l(2m)^l$.

The time for computing those tables will be mentioned in Section 7. In the following, winners are specified by indices.

EXAMPLE 2 *Let $H_{rs}[i]$ and $L_{rs}[j]$ be given in Figure 1. First we have the two lists*

$$(H_{1,2}[i], H_{3,4}[i], H_{5,6}[i], H_{7,8}[i]) = (1, 5, 12, 3)$$

$$(L_{1,2}[j], L_{3,4}[j], L_{5,6}[j], L_{7,8}[j]) = (11, 15, 10, 10)$$

Since $1 < 11, 5 < 15, 12 > 10$, and $3 < 10$, the winners at the first stage are (1, 3, 6, 7). The next lists are thus

$$(H_{1,3}[i], H_{6,7}[i]) = (13, 4),\ (L_{1,3}[j], L_{6,7}[j]) = (3, 7),\ \textit{(shown by asterisks).}$$

Since $13 > 3$ and $4 < 7$, the winners at the second stage are (3, 6), (shown by #). The last lists are $(H_{3,6}[i]) = (1)$ and $(L_{3,6}[j]) = (8)$, from which we conclude $a_{i3} + b_{3j}$ is the minimum. All tuples above are encoded in single integers in actual computation. If we had different H and L, we might have other winners such as (2, 4, 5, 8) at the first stage, and (1, 7) at the second stage, etc. We have all preparations for those situations in the form of precomputed tables, so each step is done in $O(1)$ time.

5 How to compute the tables

In Section 2 we used h_1 for encoding winners given in indices, and h_2 for encoding keys. In this section, we use the same h_1 for winners, but modify h_2 by setting $\mu = 2m$, and encoding ranks $H's$ and $L's$, instead of the set of keys.

When $h_2^{-1}(\alpha) = (a_1, ..., , a_k)$, we express the jth element of $h^{-1}(\alpha)$ by $h_2^{-1}(\alpha)[j]$, that is, $h_2^{-1}(\alpha)[j] = a_j$.

Now we construct tables $T^l, T^{l/2}, ...$ The role of table T^r is to determine winners $(x_1, ..., x_{r/2})$ for the next stage when there are r winners. Let integers α and β represent encoded forms of $l/2$ ranks in H's and L's. That is, $\alpha = h_2(H_{1,2}, ..., H_{l-1,l})$, and $\beta = h_2(L_{1,2}, ..., L_{l-1,l})$. The role of z is to represent winners in an encoded form. At the beginning all are winners. The table T^l is defined by

$$T^l(z, \alpha, \beta) = h_1(x_1, x_2, ..., x_{l/2}),$$

where $x_j = 2j - 1$ if $h_2^{-1}(\alpha)[j] < h_2^{-1}(\beta)[j]$, $x_j = 2j$ otherwise, and $z = h_1(1, 2, ..., l)$. The value of z has just one possibility. Tables $T^{l/2}, T^{l/4}, ...$ can be defined similarly using $l/2$ winners, $l/4$ winners, ... as follows:

Let α and β be encoded forms of $r/2$ ranks each and $z = h_1(z_1, ..., z_r)$. T^r is defined by

$$T^r(z, \alpha, \beta) = h_1(x_1, x_2, ..., x_{r/2}),$$

where $x_j = z_{2j-1}$ if $h_2^{-1}(\alpha)[j] < h_2^{-1}(\beta)[j]$, $x_j = z_{2j}$ otherwise.

Let $z_1, z_2, ..., z_r$ be r winners where r is an even integer. We introduce two mappings f^r and g^r for $1 \leq r \leq l$ to determine which ranks to use next after we have those r winners. We omit the subscripts i and j from $H_{rs}[i]$ and $L_{rs}[j]$ for simplicity in the following.

$$f^r(h_1(z_1, z_2, ..., z_r), h_2(H_{1,2}, ..., H_{l-1,l})) = h_2(H_{z_1,z_2}, H_{z_3,z_4}, ..., H_{z_{r-1},z_r})$$

$$g^r(h_1(z_1, z_2, ..., z_r), h_2(L_{1,2}, ..., L_{l-1,l})) = h_2(L_{z_1,z_2}, L_{z_3,z_4}, ..., L_{z_{r-1},z_r})$$

Those mappings can be computed in $O(r)$ time.

To compute $T^{l/2}(z, \alpha, \beta) = h_1(x_1, x_2, ..., x_{l/4})$, for example, we decode z, α, and β in $O(l)$ time, then test $h_2^{-1}(\alpha)[j] < h_2^{-1}(\beta)[j]$ to get $x_1, ..., x_{l/4}$, and finally encode $x_1, ..., x_{l/4}$ spending $O(l)$ time. We do this for all possible z, α, and β. Other tables can be computed similarly in $O(l)$ time. Tables f^r and g^r can also be computed in $O(l)$ time. The total time for f's and g's are not greater than that for T's.

The total time for computing those tables is thus $O(m^l(2m)^l l)$. Observe

$$O(m^l(2m)^l l) = O(c^{l \log m}), \text{ for some constant } c > 0 \ (5).$$

6 Algorithm by table look-up

Using tables $T^0, T^1, ...$, we can compute $min_r\{a_{ir} + b_{rj}\}$ by repeating the following $\log l$ steps. We omit the second argument of f^r and g^r as they are fixed throughout the next computation. We start from $f^l(z^0) = h_2(H_{1,2}, H_{3,4}, ..., H_{l-1,l})$ and $g^l(z^0) = h_2(L_{1,2}, L_{3,4}, ..., L_{l-1,l})$. The last $z^{\log l}$ is the solution index.

$$z^0 = h_1(1, 2, ..., l) \ (z^0 \text{ was precomputed})$$
$$z^1 = T^0(z^0, f^l(z^0), g^l(z^0))$$
$$z^2 = T^1(z^1, f^{l/2}(z^1), g^{l/2}(z^1))$$
$$...$$
$$z^{\log l} = T^{\log l - 1}(z^{\log l - 1}, f^2(z^{\log l - 1}), g^2(z^{\log l - 1}))$$

Now we describe the preliminary part of the Better Tournament. We assume there are $l \log l$ participants 1, 2, ..., $l \log l$ at the beginning. Suppose participants and their ranks are encoded using h_1 and h_2 at $\log l$ intervals. The first are

$$h_1(1, \log l + 1, ..., (l-1)\log l + 1),$$
$$h_2(H_{1,2}, H_{\log l+1,\log l+2}, ..., H_{(l-1)\log l+1,(l-1)\log l+2}),$$
$$h_2(L_{1,2}, L_{\log l+1,\log l+2}, ..., L_{(l-1)\log l+1,(l-1)\log l+2}).$$

In the Tournament as applied to DMM, we extended table T with two arguments of α and β. We can similarly extend table W with α and β. By using W and f alternately, we can select m winners in encoded form, and pass them to the Tournament. This preliminary part takes $O(\log l)$ time. As in Section 2, the time for constructing W's can be estimated in the same formula with that for T's with diffrent c.

Now we resize l. Suppose we have l participants at the beginning. Then the value of l in the preliminary part and the Tournament is replaced by $l/\log l$. The time is still $O(\log l)$, but m can be larger as seen in the next section.

7 Analysis of computing time

The time for this algorithm to compute (4) by the Better Tournament is $O(m^2 \log l) = O(m^2 \log m)$.

Let us evaluate the time for (3). Since there are m/l products $A_k B_k$ in (3), we need $O((m/l)m^2 \log m)$ time. To compute minimum component-wise in $min_{k=1}^{m/l}\{A_k B_k\}$, we need $O((m/l)m^2)$ time. We also need $O(Ml^2 m) = O(lm^2)$ time for Ml^2 mergings as described in Section 3.

We set $l = (m \log m)^{1/2}$ to balance the first and the third of the above three complexities. Then to compute the product of (m, m)-matrices, we take $T(m) = O(m^2(m \log m)^{1/2})$ time.

We determine the size of submatrices in Section 3. Let m be given by

$$m = \log^2 n/(\log^2 c \log \log n).$$

Then we have $l \leq \log n/\log c$ for sufficiently large n. As we substitute $l/\log l$ for l in the $O(c^{l \log m})$ time for making the tables given in equation (5), the time for constructing tables is $O(c^{(l/\log l)\log m}) = O(n)$. Substituting the value of m for $O(n^3(\log m/m)^{1/2})$, we have the overall computing time for distance matrix multiplication as $O(n^3 \log \log n/\log n)$.

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

The encoding of winners and ranks for the preliminary part of the Better Tournament is $O(l)$, and encoding can be done only at the beginning of the Better Tournament. If we do this for all the submatrices, the time is not greater than that for the above described sorting.

8 Generalization of distance matrix multiplication

To prepare for the k-maximum subarray problem, we extend equation (1) in Section 3 in such a way that c_{ij} is the k-tuple of k minima of $\{a_{il} + b_{lj}\}$ in non-decreasing order. We changed k in (1) to l to avoid confusion. We call this definition k-distance matrix multiplication, or simply k-matrix multiplication. We generalize the min and max operations on distance matrices in this section. Let each element of a distance matrix be a k-tuple of real numbers such as $\mathbf{a} = (a_1, ..., a_k)$ in non-decreasing order. The min operation on the two k-tuples $\mathbf{a}$ and $\mathbf{b}$ is defined by $min(\mathbf{a}, \mathbf{b}) = (c_1, ..., c_k)$, where $(c_1, ..., c_{2k})$ is the merged list of $\mathbf{a}$ and $\mathbf{b}$. Similarly we can define $max(\mathbf{a}, \mathbf{b}) = (c_{k+1}, ..., c_{2k})$. In the following we mainly describe the min version. The max version can be defined symmetrically. If each element of distance matrices A_1 and A_2 is a k-tuple, the min operation on A_1 and A_2 is defined component-wise over corresponding k-tuples. To compute k minima for each element in (1), we can use the extended min operation in (2) and (3). Then at the bottom, we need to return k minima in (4). Recall that the size of the Better Tournament, l, is $O(\log n / \log \log n)$. To fit into the Better Tournament, we assume $l \leq \log n / \log \log n$. Let us choose the second minimum after the minimum is finalized. Let t be the winner with the minimum. Keeping other elements intact, we partially reset H and L as follows:

$$H_{t,s} = 2m + 1, \text{ for } s = t + 1, ..., l, \; L_{r,t} = 2m + 1, \text{ for } r = 1, ..., t - 1$$

In other words, we exclude t from the second tournmnet, as t will be the loser for all matches. If we parepare for all possible 2^l subsets of participants the above described modifed ranks and their encoded tables, we can select k minima in $O(k \log l)$ time, by changing the final winner to a loser one by one. The number of tables required must be multiplied by 2^l. By adjusting the constant c, we can show that the time for table construction is $O(n)$. We conclude that k minima in (1) can be computed in $O(kn^3 \log \log n / \log n)$ time, if $k \leq \log n / \log \log n$. Otherwise we can divide the given matrices into (k, k) ones, and compute

$$C_{ij} = min_{l=1}^{n/k} \{A_{il} B_{lj}\} \text{ (6)}.$$

Let us rename A_{il} and B_{lj} in the above by A and B, and consider the multiplication. This time we can return all $\{a_{i1} + b_{1j}, ..., a_{ik} + b_{kj}\}$ as candidate k-tuples, and use extended min operations in (6). Then the time is shown to be $O(n^3)$, when $k \leq n$.

9 Application to the k-maximum subarray problem

Now we proceed to the maximum subarray problem for an array of size (n, n). The cubic algorithm for this problem given by Bentley [3] was improved to subcubic by Tamaki and Tokuyama [10]. We review the simplified subcubic version in [8]. We give a two-dimensional array $a[1..m, 1..n]$ of real numbers as input data. The maximum subarray problem is to maximize the sum of the array portion $a[k..i, l..j]$, that is, to obtain such indices (k, l) and (i, j). We suppose the upper-left corner has co-ordinates (1,1).

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 $O(m^2 n)$ time, which is cubic when $m = n$.

The central algorithmic concept in this section is that of prefix sum. We use distance matrix multiplications of both *min* and *max* versions in this section. We compute the prefix sums $s[i,j]$ for array portions of $a[1..i,1..j]$ for all i and j with boundary condition $s[i,0] = s[0,j] = 0$. Obviously this can be done in $O(mn)$ time. 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. Otherwise, if $m > n$, rotate the array 90 degrees.
3. Thus we assume $m \leq n$.
4. Let A_{left} be the solution for the left half.
5. Let A_{right} be the solution for the right half.
6. Let A_{column} be the solution for the column-centered problem.
7. Let the solution be the maximum of those three.

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

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

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 $k = 0, ..., i-1$.

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

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

$$A_{column}[i,k] = -min_{l=0}^{n/2-1}\{s[i,l] + s^*[l,k]\} + max_{j=n/2+1}^{n}\{s[i,j] + s^*[j,k]\} \quad (7)$$

The first part in the above is distance matrix multiplication of the *min* version and the second part is of the *max* version. Let S_1 and S_2 be matrices whose (i,j) elements are $s[i,j-1]$ and $s[i,j+n/2]$. For an arbitrary matrix T, let T^* be that obtained by negating and transposing T. As the range of k is $[0\ ..\ m-1]$ in S_1^* and S_2^*, we shift it to $[1..m]$. 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.

For simplicity, we apply the algorithm on a square array of size (n,n), where n is a power of 2. Then all parameters m and n appearing through recursion in Algorithm M are power of 2, where $m = n$ or $m = n/2$. We observe the algorithm splits the array vertically and then horizontally. We define the work of computing the three A_{column}'s through this recursion of depth 2 to be the work at level 0. The algorithm will split the array horizontally and then vertically through the next recursion of depth 2. We call this level 1, etc.

Now let us analyze the time for the work at level 0. We can multiply $(n, n/2)$ and $(n/2, n)$ matrices by 4 multiplications of size $(n/2, n/2)$, and there are two such multiplications in

(7). 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 two smaller A_{column}'s, spending $12M(n)$ comparisons. Thus we have the following recurrence for the total time $T(n)$. The following lemma is obvious.

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

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

Clearly the complexity of $O(n^3 \log\log n/\log n)$ for $M(n)$ satisfies the condition of the lemma with some constant $c > 0$. Thus the maximum subarray problem can be solved in $O(n^3 \log\log n/\log n)$ 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 $M(n)$.

Suppose n is not given by powers of 2. By embedding the array a in an array of size (n',n') such that n' is the next power of 2 and the gap is filled with 0, we can solve the original problem in the complexity of the same order. Similar considerations can be made on k in the following.

Now we describe the k-maximum subarray problem. The k-maximum subarray problem is to work out k maximum subarrays. If we require only non-overlapping subarrays, it is straightforward. We consider general cases. An $O(kn^3)$ time algorithm is reported in [2], which we call Algorithm A. Suppose k is a power of 2. First we change line 1 in Algorithm M as follows:

1. If the array becomes $n^{1/2}$ by $n^{1/2}$, return the solution by Algorithm A.

Next we define $\mathbf{a} - \mathbf{b}$ for two k-tuples $\mathbf{a}$ in non-increasing order and $\mathbf{b}$ in non-decreasing order to be the maximum k values in non-increasing order among k^2 values that are made by subtracting elements of $\mathbf{b}$ from those of $\mathbf{a}$. To compute distance matrix multiplication in $S_2S_2^* - S_1S_1^*$ in (7), we use the k-matrix multiplication of max and min version described in Section 8. To compute subtraction, we follow the above operation of $\mathbf{a} - \mathbf{b}$ component-wise. As k is small, this complexity of subtraction and susequent triangulation is absorbed in the main complexity. The initial condition for T becomes: $T(n^{1/2}) = O(kn^{3/2})$. As there are $n^{1/2} \times n^{1/2} = n$ subarrays at the bottom of recursion, the total time spent by Algorithm A is $O(kn^{5/2})$. If we use the $O(n^3)$ time algorithm for the k maximum subarray problem, the total time before hitting the bottom of recursion is $O(n^3)$. Thus the total time is $O(n^3)$ when $k \leq n^{1/2}$.

Let us use the faster k-matrix multiplication algorithm. Then we can show that the k-maximum subarray problem can be solved in $O(kn^3 \log\log n/\log n)$ time when $k \leq (1/2)\log n/\log\log n$, and in $O(n^3)$ time for $(1/2)\log n/\log\log n < k \leq n^{1/2}$. For the complexity of $O(kn^3 \log\log n/\log n)$, we can relax this bound on k close to $\log n/\log\log n$ by changing $n^{1/2}$ in the modified line 1 to $n^{1-\epsilon}$ with small ϵ.

10 Concluding remarks

We showed an asymptotic improvemenet on the time complexity of the all-pairs shortest path problem. The results will have consequences for application areas where DMM or the APSP problem is used. As an example, we showed that the maximum subarray problem [8] can be solved in the same complexity as that of DMM. Also the k maximum subarray problem [2] can be solved in subcubic time in terms of n, and linear in terms of k for small k. There may be some room for improving the factor of k in the complexity.

References

[1] Alon, N, Galil, and Margalit, On the Exponent of the All Pairs Shortest Path Problem, Jour. Comp. Sys. Sci., vol 54, no. 2, pp 255-262, 1997

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

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

[4] Fredman, M, New bounds on the complexity of the shortest path problem, SIAM Jour. Computing, vol. 5, pp 83-89, 1976

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

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

[7] Takaoka, T, Subcubic algorithms for the all pairs shortest path problem, Algorithmica, vol. 20, 309-318, 1998

[8] Takaoka, T, Subcubic algorithms for the maximum subarray probelm, Proc. Computing:Australasian Theory Symposium (CATS 2002), pp 189-198, 2002.

[9] Takaoka, T., A Faster Algorithm for the All Pairs Shortest Path Problem and its Application, Proc. COCOON 2004, to appear in 2004

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

[11] Zwick, U, All pairs shortest paths in weighted directed graphs - exact and almost exact algorithms, 39th FOCS, pp 310-319, 1998.