An $O(n^3 \log\log n / \log^2 n)$ Time Algorithm for All Pairs Shortest Paths

Yijie Han[1] and Tadao Takaoka[2]

[1] School of Computing and Engineering, University of Missouri at Kansas City, Kansas City, MO 64110, USA
hanyij@umkc.edu,

[2] Department of Computer Science and Software Engineering, University of Canterbury, Christchurch, New Zealand
T.Takaoka@cosc.canterbury.ac.nz

Abstract. We present an $O(n^3 \log\log n / \log^2 n)$ time algorithm for all pairs shortest paths. This algorithm improves on the best previous result of $O(n^3 (\log\log n)^3 / \log^2 n)$ time.

Keywords: Algorithms, all pairs shortest paths, graph algorithms, upper bounds.

1 Introduction

Given an input directed graph $G = (V, E)$, the all pairs shortest path problem (APSP) is to compute the shortest paths between all pairs of vertices of G assuming that edge costs are real values. The APSP problem is a fundamental problem in computer science and has received considerable attention. Early algorithms such as Floyd's algorithm ([2], pp. 211-212) computes all pairs shortest paths in $O(n^3)$ time, where n is the number of vertices of the graph. Improved results show that all pairs shortest paths can be computed in $O(mn+n^2 \log n)$ time [9], where m is the number of edges of the graph. Pettie showed [14] an algorithm with time complexity $O(mn + n^2 \log\log n)$. See [15] for recent development. There are also results for all pairs shortest paths for graphs with integer weights[10, 16, 17, 21–23]. Fredman gave the first subcubic algorithm [8] for all pairs shortest paths. His algorithm runs in $O(n^3 (\log\log n / \log n)^{1/3})$ time. Fredman's algorithm can also run in $O(n^{2.5})$ time nonuniformly. Later Takaoka improved the upper bound for all pairs shortest paths to $O(n^3 (\log\log n / \log n)^{1/2})$ [19]. Dobosiewicz [7] gave an upper bound of $O(n^3/(\log n)^{1/2})$ with extended operations such as normalization capability of floating point numbers in $O(1)$ time. Earlier Han obtained an algorithm with time complexity $O(n^3 (\log\log n / \log n)^{5/7})$ [12]. Later Takaoka obtained an algorithm with time $O(n^3 \log\log n / \log n)$ [20] and Zwick gave an algorithm with time $O(n^3 \sqrt{\log\log n} / \log n)$ [24]. Chan gave an algorithm with time complexity of $O(n^3 / \log n)$ [6]. Chan's algorithm does not use tabulation and bit-wise parallelism. His algorithm also runs on a pointer machine.

What subsequently happening was very interesting. Takaoka thought that $O(n^3/\log n)$ could be the ultimate goal and raised the question [20] whether $O(n^3/\log n)$ can be achieved. Chan first achieved $O(n^3/\log n)$ time and also thought that $O(n^3/\log n)$ is a natural bound [6]. However, Han showed an algorithm with $O(n^3(\log\log n/\log n)^{5/4})$ time [11]. Because in [11] Han exhausted Takaoka's technique [19] Han thought that this result should be it and did not expect any improvements in the near future (see the reasoning Han gave in the paper [11] explaining why it should be difficult to further improve). However, Chan came up with an algorithm with time complexity $O(n^3(\log\log n)^3/\log^2 n)$ [5]. Chan [5] believes that $O(n^3/\log^2 n)$ is probably the final chapter. Our experience indicates that Chan may be correct. Here we present an algorithm with time complexity $O(n^3\log\log n/\log^2 n)$. Thus we further remove a factor of $(\log\log n)^2$ from the time complexity of the best previous result due to Chan.

We would like to point out the previous results which influenced the formation of our ideas presented in this paper. They are: Floyd's algorithm [2], Fredman's algorithm [8], Takaoka's algorithm [19], Han's algorithm [11], Chan's algorithm [5].

2 Preparation

Since it is well known that the all pairs shortest paths computation has the same time as computing the distance product of two matrices [1] ($C = AB$), we will concentrate on the computation of distance product.

We divide the first $n\times n$ matrix A into t_1 submatrices $A_1, A_2, , A_{t_1}$ each having dimension $n\times n/t_1$, where $t_1 = n^{1-r_1}$ and r_1 is a constant to be determined later. We divide the second $n \times n$ matrix B into t_1 submatrices $B_1, B_2, ..., B_{t_1}$ each having dimension $n/t_1 \times n$. Therefore $C = AB = A_1B_1 + A_2B_2 + ... + A_{t_1}B_{t_1}$, where $*$ is addition and $+$ is the minimum operation. In the following we consider the computation of the distance product of an $n\times n/t_1$ matrix E with a $n/t_1\times n$ matrix F. The reason we need to do the division for this level will be understood later in this paper.

We then divide the $n \times n/t_1$ matrix E into $t_2 = (n/t_1)/(r_2\log n/\log\log n)$ submatrices $E_1, E_2, ..., E_{t_2}$ each having dimension $n\times(r_2\log n/\log\log n)$, where r_2 is a constant to be determined later. Similarly we divide the $n/t_1\times n$ matrix F into t_2 submatrices each having dimension $(r_2\log n/\log\log n)\times n$. Now $EF = E_1F_1 + E_2F_2 + ... + E_{t_2}F_{t_2}$.

In the following we will first consider the computation of E_1F_1, and then the computation of EF (or A_1B_1). After that it takes $O(n^2t_1)$ time straightforwardly to get the all-pairs shortest path of the input graph.

Let $E_1 = (e_{ij})$ and $F_1 = (f_{ij})$. We will first, for each $1 \le i, j \le r_2\log n/\log\log n$, sort $f_{ik} - f_{jk}$, $k = 1, 2, ..., n$. After sorting each $f_{ik} - f_{jk}$ has a rank in $[1, n]$. We then give $f_{ik} - f_{jk}$ a label $l_f(i, j, k) = l$ if $f_{ik} - f_{jk}$ has rank in $(l\cdot n/\log^9 n, (l + 1)\cdot n/\log^9 n]$. $l_f(i, j, k)$ uses $9\log\log n$ bits. For each $e_{kj} - e_{ki}$ we will give it label $l_e(k, i, j) = l$ if $f_{ik_1} - f_{jk_1} < e_{kj} - e_{ki} \le f_{ik_2} - f_{jk_2}$, where $f_{ik_1} - f_{jk_1}$ has

rank (not label) $l \cdot n/\log^9 n$ and $f_{ik_2} - f_{jk_2}$ has rank $(l+1) \cdot n/\log^9 n$. $l_e(k,i,j)$ also uses $9 \log\log n$ bits.

According to Fredman [8] and Takaoka [19], if the labels of $e_{k_1 j} - e_{k_1 i}$ and $f_{ik_2} - f_{jk_2}$ are different then we can determine either $e_{k_1 i} + f_{ik_2} < e_{k_1 j} + f_{jk_2}$ or $e_{k_1 i} + f_{ik_2} > e_{k_1 j} + f_{jk_2}$. Say $e_{k_1 j} - e_{k_1 i}$ has label l_e and $f_{ik_2} - f_{jk_2}$ has label l_f. If $l_e < l_f$ ($l_f < l_e$) then $e_{k_1 j} - e_{k_1 i} < f_{ik_2} - f_{jk_2}$ ($e_{k_1 j} - e_{k_1 i} > f_{ik_2} - f_{jk_2}$) and therefore $e_{k_1 j} + f_{jk_2} < e_{k_1 i} + f_{ik_2}$ ($e_{k_1 j} + f_{jk_2} > e_{k_1 i} + f_{ik_2}$). However, when their labels are the same then we cannot determine this. However, only a fraction $1/\log^9 n$ of the total number of $(e_{k_1 j} - e_{k_1 i})$'s (one out of all lables) are undetermined for each $f_{ik_2} - f_{jk_2}$ and therefore overall a fraction of $1/\log^9 n$ of all pairs of $e_{k_1 j} - e_{k_1 i}$ and $f_{ik_2} - f_{jk_2}$ are undetermined. In case of indeterminacy for two indices i, j we will pick i over j (to include i in the computation) when $i < j$ and leave the j-th position (or index) to be computed separately. This separated computation can be done in brute force and it takes $O(n^3/\log^8 n)$ time for the whole computation, i.e. the computation of AB. The actual counting of complexity of this separate computation is as this: There are $w = O(n^3 \log n/\log\log n)$ pairs of $e_{k_1 j} - e_{k_1 i}$ and $f_{jk_2} - f_{ik_2}$ (k_1 and k_2 each take value in $[0..n-1]$ and thus have a factor of $x = n^2$, there are $y = n \log\log n/(r_2 \log n)$ pairs of E and F for each pair of k_1 and k_2 and for each pair of E and F there are $z = O((r_2 \log n/\log\log n)^2)$ pairs of $e_{k_1 j} - e_{k_1 i}$ and $f_{jk_2} - f_{ik_2}$. $w = xyz$. Because $1/\log^9 n$ of these pairs are in the separate computation the complexity of the separate computation takes $O((n^3 \log n/\log\log n) \cdot (1/\log^9 n)) = O(n^3/\log^8 n)$ time.

3 The Further Details

Now for fixed i and k we pack $l_e(k,i,j)$, $j = 1, 2, ..., r_2 \log n/\log\log n$, into one word and call it $l_e(k,i)$. This can be done when r_2 is small. Also for fixed i and k we pack $l_f(i,j,k)$, $j = 1, 2, ..., r_2 \log n/\log\log n$, into one word and call it $l_f(i,k)$. By comparing $l_e(k_1,i)$ and $l_f(i,k_2)$ we can let the result be 1 if index i should be chosen over all the other $r_2 \log n/\log\log n - 1$ indices, and let it be 0 otherwise. This computation of comparing one index with all the other $r_2 \log n/\log\log n$ indices is done in constant time by concatenating $l_e(k_1,i)$ and $l_f(i,k_2)$ into one word of less than $\log n$ bits and then index into a precomputed table to get the result of either 0 or 1.

Note that since $l_e(k,i)$ has $9r_2 \log n$ bits, and we will pick r_2 later such that $9r_2 \log n$ is much smaller than $\log n$ and therefore $l_e(k,i), k = 1, 2, ..., n$ can be sorted into $t_3 = 2^{9r_2 \log n} = n^{9r_2}$ blocks such that each block has the same word $(l_e(k,i))$ value.

For the purpose of computing $E_1 F_1$, there are $r_2 \log n/\log\log n$ i's (columns in E_1) and for each (column) of them we have n $l_e(k,i)$'s (one on each row) and these $l_e(k,i)$'s ($0 \le k < n$) form t_3 blocks and for each of these blocks we get a $1 \times n$ vector of bits (0's and 1's) that are the result of the above computation (i.e. indexing into a table to get a 0 or a 1). We need to compute these (a vector of n) 0's and 1's for one $l_e(k,i)$ in a block because all $l_e(k,i)$'s in a block have the same value. Thus we need to compute $r_2(\log n/\log\log n)t_3$ vectors (of n bits

each), and this can be done in $O(r_2(\log n/\log\log n)t_3 n)$ time (one step gets one bit by the above table lookup computation).

On the average for each such an n bit vector v there are only $n/(r_2 \log n/\log\log n)$ bits that are 1's and the remaining bits are 0's. This is so because of the way we formed $l_e(k_1, i)$ and $l_f(i, k_2)$ and the way that we stipulate that the result of comparison of $l_e(k_1, i)$ and $l_f(i, k_2)$ is 0 or 1. We take v and first divide it into $n/((r_2 \log n/\log\log n)(r_3 \log n/\log\log n))$ small vectors each with dimension $1 \times ((r_2 \log n/\log\log n)(r_3 \log n/\log\log n))$, where r_3 is a constant to be determined later. Now, on the average, each small vector has $r_3 \log n/\log\log n$ bits which are 1's. If a small vector v' has between $(t-1)r_3 \log n/\log\log n + 1$ and $tr_3 \log n/\log\log n$ bits of 1's we will make a set V of t small vectors each having $((r_2 \log n/\log\log n)(r_3 \log n/\log\log n))$ bits and containing a subset of no more than $r_3 \log n/\log\log n$ 1's from v'.

Because the multiplication of each row of E_1 with all columns of F_1 results in $r_2 \log n/\log\log n$ vectors having a total of n bits of 1's, they will result in $2n(r_2 \log n/\log\log n)/((r_2 \log n/\log\log n)(r_3 \log n/\log\log n)) = 2n/(r_3 \log n/\log\log n)$ small vectors, where factor 2 is because of some small vectors may have less than $r_3 \log n/\log\log n$ bits of 1's.

For fixed i (a row of F_1) (and therefore $l_f(i, k)$, $0 \le k < n$) and fixed value of $l_e(k, i)$'s (a block) we formed $2n/((r_2 \log n/\log\log n)(r_3 \log n/\log\log n))$ small vectors each having $(r_2 \log n/\log\log n)(r_3 \log n/\log\log n)$ bits with no more than $r_3 \log n/\log\log n$ bits are 1's. Therefore each small vector can be represented by a word (with no more than $\log n$ bits) when r_3 is small. This is so because $\sum_{t=0}^{r_3 \log n/\log\log n} \binom{(r_2 \log n/\log\log n)(r_3 \log n/\log\log n)}{t} < n$. We first form these $2n/((r_2 \log n/\log\log n)(r_3 \log n/\log\log n))$ words for each vector (on the average) and then duplicate these words for all rows in the block because all rows in the same block has the same $l_e(k, i)$ value. The reason we do this duplicating is to save time because small vectors with the same value need not to be computed into words repeatedly. Thus for the multiplication of $E_1 F_1$ we obtained $2n^2/(r_3 \log n/\log\log n)$ words. And for the multiplication of $A_1 B_1$ we obtained $2n^{2+r_1}/((r_2 \log n/\log\log n)(r_3 \log n/\log\log n))$ words. And therefore for the multiplication of AB we have obtained $O(n^3(\log\log n/\log n)^2)$ words and computation thus far takes $O(n^3(\log\log n/\log n)^2)$ time.

However these $O(n^3(\log\log n/\log n)^2)$ words contain more than $O(n^3(\log\log n/\log n)^2)$ indices because multiple indices are coded into one word. Thus we shall combined these words to cut the number of indices.

To further combine these words we need to consider only the words formed by $E_i[1, 1..(r_2 \log n/\log\log n)] \times F_i[1..(r_2 \log n/\log\log n),$
$1..(r_2 \log n/\log\log n)(r_3 \log n/\log\log n)]$ (there are $r_2 \log n/\log\log n$ resulting words out of this multiplication as we explained above),
$i = 1, 2, ..., n^{r_1}/(r_2 \log n/\log\log n)$. Thus there are n^{r_1} words. We need to reduce them to $n^{r_1}/(r_3 \log n/\log\log n)$ words in $O(n^{r_1})$ time and thereafter we can simply disassemble indices (for minimum) out of packed words and finish the remaining computation straightforwardly. Because each word w contains a set S_w of no more than $r_3 \log n/\log\log n$ columns (these columns have 1's and the other

columns have 0's) in F_i there are $\sum_{l=1}^{r_3 \log n/\log\log n} \binom{((r_2 \log n/\log\log n)(r_3 \log n/\log\log n))}{l}$ $\leq n^{cr_3}$ choices, where c is a constant. When r_3 is much smaller than r_1 there are many words among the n^{r_1} words having the same S_w sets. This is the fact we can take advantage of. In the following we will refer two of these words with the same S_w sets as w_1 and w_2, i.e., the two small vectors represented by w_1 and w_2 are the same (equal or identical).

4 Combining Words

The scheme for combining words is a little complicated. The basic idea is that, since we have indices gathered in $O(n^3(\log\log n/\log n)^2)$ words, we just need to do pair-wise comparisons between pairs of indices (paths) to reduce the number of indices by half. If we do this for $2\log\log n$ rounds we can reduce the number of indices to $n^3/(\log n)^2$ and then we can just disassemble indices from words and finish the computation straightforwardly in $O(n^3/(\log n)^2)$ time. Note that because we do pairing-off the time complexity will remain to be $O(n^3(\log\log n/\log n)^2)$.

The complication of our algorithm comes from the fact that indices are encoded in $O(n^3(\log\log n/\log n)^2)$ words. To deal with this encoding we have to design an algorithm that utilizes the special characteristics of the encoding.

We use a different labeling for the matrix $B_1 = (b_{ij})$ and $A_1 = (a_{ij})$. We will, for each $1 \leq i, j \leq n^{r_1}$, sort $b_{ik} - b_{jk}$ together with $a_{kj} - a_{ki}$, $k = 1, 2, ..., n$. For A and B the total time for sorting is $O(n^{2+r_1}\log n)$. This gives the rank of $b_{ik} - b_{jk}$ $(a_{kj} - a_{ki})$ which we denote by $l_{b_1}(i, j, k)$ $(l_{a_1}(k, i, j))$. These ranks take value from 1 to $2n$ and have $\log n + 1$ bits. There are n^{2r_1} choices of i, j pairs and for each of these choices (each pair of i and j) and for each set

$U_t = \{t(r_2 \log n/\log\log n)(r_3 \log n/\log\log n) + 1,$
$t(r_2 \log n/\log\log n)(r_3 \log n/\log\log n) + 2, ...,$
$(t+1)(r_2 \log n/\log\log n)(r_3 \log n/\log\log n)\},$

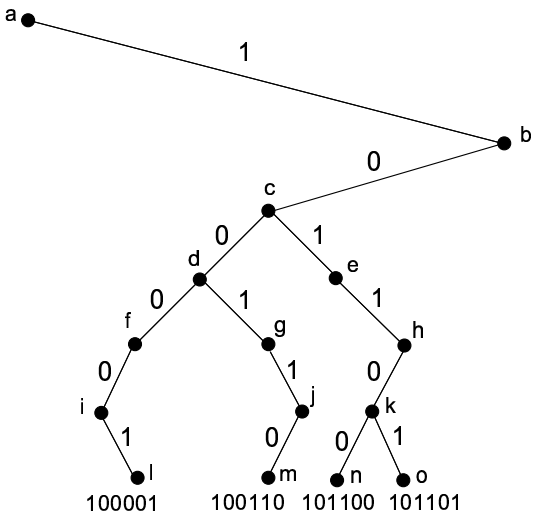
$t = 1, 2, ..., n/((r_2 \log n/\log\log n)(r_3 \log n/\log\log n))$, where values of k are taken, choose any subset of U_t containing no more than $r_3 \log n/\log\log n$ elements and there are no more than $(n/((r_2 \log n/\log\log n)(r_3 \log n/\log\log n))$ $* \sum_{l=1}^{r_3 \log n/\log\log n} \binom{((r_2 \log n/\log\log n)(r_3 \log n/\log\log n))}{l} \leq n^{1+cr_3}$ choices, where c is a constant, and thus there are a total of $n^{1+2r_1+cr_3}$ choices for all pairs of i and j. For each of these choices we use a word w to represent the total of (a) a choice of the pair i, j, (b) a choice (referred to as d) of the subset of U_t ($n^{2r_1+cr_3}$ choices) and (c) the packing of no more than $r_3 \log n/\log\log n$ ranks ($l_{b_1}(i, j, k)$'s) (where k take values over elements in the subset of U_t). Note that straightforward packing will not work because it will take $O(\log^2 n/\log\log n)$ bits (a subset of U_t has up to $O(\log n/\log\log n)$ elements and each element corresponds to $O(\log n)$ bits of an $l_{b_1}(i, j, k)$.) and cannot be stored in a word of $\log n$ bits. What we do is first to build a trie for the $r_3 \log n/\log\log n$ ranks. An example of such a trie is shown in Fig. 1(a). This trie is a binary tree with a node having two children

when there are ranks with the most significant bits differ at the node's level. Next we build a packed trie by removing nodes v with only one child except the root. The edge connecting this removed node and its child is also removed. This is shown in Fig. 1(b). Thus let $v_1, v_2, ..., v_t$ be such a chain with v_i being v_{i+1}'s parent, v_1 and v_t having two children and v_i, $i \neq 1, t$, having one child, and we will remove $v_2, v_3, ..., v_{t-1}$. Edges $(v_2, v_3), (v_3, v_4), ..., (v_{t-1}, v_t)$ are also removed. The edge originally connecting v_1 and v_2 are now made to connect v_1 and v_t. We will record on edge (v_1, v_t) that $t-2$ edges (bits) are removed. Also at leaves, we store only relative address of k (having value between 1 and $(r_2 \log n / \log\log n)(r_3 \log n / \log\log n)$) instead of the value of $l_{b_1}(i, j, k)$ (having value between 1 and $2n$). Such a packed trie is shown in Fig. 1(c) and it can be stored in a word w with $c \log n$ bits, where c is a constant less than 1.

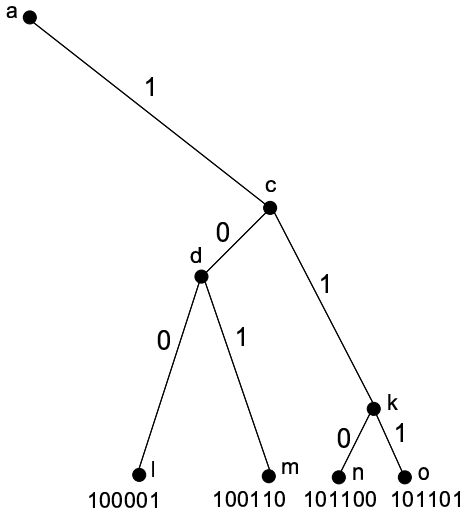
Now with $l_{a_1}(1, i, j)$, we can follow this packed trie in word w and reach a leaf l of the trie (by starting at the root, repeatedly deleting corresponding bits which has been removed from the packed trie and following the path in the packed trie). In Fig. 1(d) we show such situations. Therefore we can precompute a table T_1 and use the values of $l_{a_1}(1, i, j)$ and w to index into T_1 to get leaf l. (Note that $l_{a_1}(1, i, j)$ has $\log n + 1$ bits but this can be reduced to $c \log n$ bits with $c < 1$ by using multiple tables replacing T_1 and taking care of a fraction of $\log n + 1$ bits at a time). Here we will use l to represent both the leaf and the relative address of k mentioned in immediate previous paragraph. From l we get the value of k and we can then compare $l_{a_1}(1, i, j)$ and $l_{b_1}(i, j, k)$ to find the most significant bit where $l_{a_1}(1, i, j)$ and $l_{b_1}(i, j, k)$ differ (this can be done by exclusive-oring the two values and then finding the most significant bit which is 1 by indexing into a precomputed table). Say this bit is the b-th most significant bit. By using the values of b, $l_{a_1}(1, i, j)$ and w (indexing into another precomputed table T_2) we can then figure out at which chain C of the original trie $l_{a_1}(1, i, j)$ "branches out". Note that we need NOT to know the value of C. We need know only within C the branch happens and whether it branches to left or to right (these information can be figured out with b, $l_{a_1}(1, i, j)$ and w.) Thus the output of indexing into T_2 can be the columns where index i should be taken over index j.

We can store w as an array element in an array W as $W[i][j][t][d]$, here, i, j, t, d are the ones mentioned in the first paragraph of this section. We need not to pack $l_{a_1}(k, i, j)$'s because only one $l_{a_1}(k, i, j)$ is considered at a time.

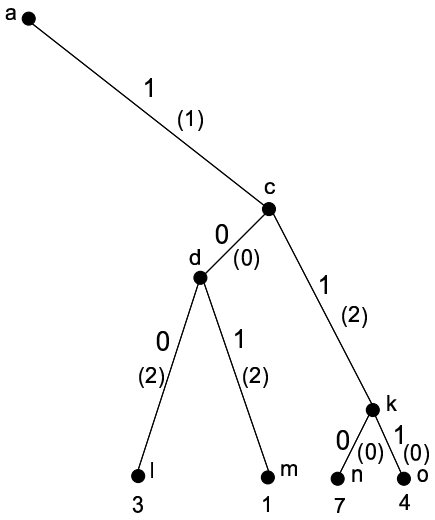
Now for the above mentioned words w_1 and w_2 (last paragraph of Section 3) we can get t and d value (both w_1 and w_2 are associated with the same t value because we put them together as explained above, w_1 and w_2 are associated with the same d value because as we mentioned above that we can get this by setting r_3 much smaller than r_1). We can also get i from w_1 and j from w_2. Thus we can get $W[i][j][t][d]$. Now we use the values of $W[i][j][t][d]$ and $l_{a_1}(1, i, j)$ to index into precomputed table T_1, T_2 to get rid of half of indices in w_1 plus w_2. Repeating this process we can then remove a factor of $r_3 \log n / \log\log n$ from the total number of indices. Thereafter we disassemble the indices from packed



(a). A trie built based on the ranks. These ranks are shown at leaves.



(b). The packed trie of the trie in (a).



(c). Packed trie. Numbers in parenthesis are the number of edges (bits) removed from the chain. The number at the leaves are relative address of columns of the matrix.

For rank 101000 we reach leaf n (first 1 go from a to c, remove 1 bit that is 0, next 1 go from c to k, remove 2 bits that are 00, next 0 go from k to n).

For rank 111011 we reach leaf o.

For rank 110100 we reach leaf m.

For rank 100101 we reach leaf m.

For rank 011001 we branch at a.

(d).

Fig. 1.

such that one word contains one index and the remaining computation can be carried out easily.

The precomputation of tables T_1, T_2 is not difficult and its complexity cannot dominate the overall computation. The reason is because all these computations have polynomial complexity and we can reduce the table size to n^ϵ with ϵ being an arbitrarily small constant.

The choice of r_1, r_2, r_3 should not be difficult now. The complexity of the algorithm as we described above is $O(n^3(\log\log n)^2/\log^2 n)$.

5 Removing Another Factor of $\log\log n$ From Complexity

In our algorithm we partitioned the number of rows of E_1 and the number of columns of F_1 by a factor of $O(\log^c n)$ at a time. This gives $O(\log n/\log\log n)$ depths of partitioning and results in the loss of a factor of $\log\log n$ in time complexity. If we partition the number of rows of E_1 and the number of columns of F_1 by a constant factor at a time our algorithm would have $O(\log n)$ depths of partitioning and thus can remove another factor of $\log\log n$ from time complexity. If we do this way the rows of E_1 and columns of F_1 are not partitioned uniformly. Such modification does not involve new ideas and does not require drastically different time analysis. The principle and the approach remain intact, only the parameter of the algorithm changed. The details of this modification is as follows.

Let $E_1 = (e_{ij})$ and $F_1 = (f_{ij})$. We will first, for each $1 \le i, j \le c_1 \log n$ ($c_1 < 1$ is a constant), sort $f_{ik} - f_{jk}$, $k = 1, 2, ..., n$. After sorting each $f_{ik} - f_{jk}$ has a rank in $[1, n]$. We then give $f_{ik} - f_{jk}$ a label $l_f(i, j, k) = l$ if $f_{ik} - f_{jk}$ has rank in $(l \cdot n/2, (l+1) \cdot n/2]$. $l_f(i, j, k)$ uses 1 bit. For each $e_{kj} - e_{ki}$ we will give it label $l_e(k, i, j) = l$ if $f_{ik_1} - f_{jk_1} < e_{kj} - e_{ki} \le f_{ik_2} - f_{jk_2}$, where $f_{ik_1} - f_{jk_1}$ has rank (not label) $l \cdot n/2$ and $f_{ik_2} - f_{jk_2}$ has rank $(l+1) \cdot n/2$.

Now if $l_e(k_1, i, j) \ne l_f(i, j, k_2)$ then we can decide to discard an index. If $l_e(k_1, i, j) = l_f(i, j, k_2)$ then we cannot make a decision. If we group elements of the same label together then the above labeling partitions the array $[0..n-1, 0..n-1]$ into 4 divisions and among them there are 2 divisions we can determine the index for the shorter path and discard the other index and for the other 2 divisions we cannot determine. The area for the determined divisions is $n^2/2$ and the area for the undetermined divisions is also $n^2/2$. Now for the undetermined divisions we sort and label elements again and further partitions. In this way when we partitioned to $c \log\log n$ levels for a constant c then the area of undetermined divisions is $n^2/\log^c n$. See Fig. 2 (Fig. 2 may looks like a Quad tree and actually it is not).

Built on top of the above partition we now do $l_e(k_1, i, j+1)$ and $l_f(i, j+1, k_2)$. This will further partition the divisions. However, once the undetermined divisions area reaches $n^2/\log^c n$ we will stop and not further partition it. Thus the undetermined divisions obtained when we worked on $l_e(k_1, i, j)$ and $l_f(i, j, k_2)$ are not further partitioned.

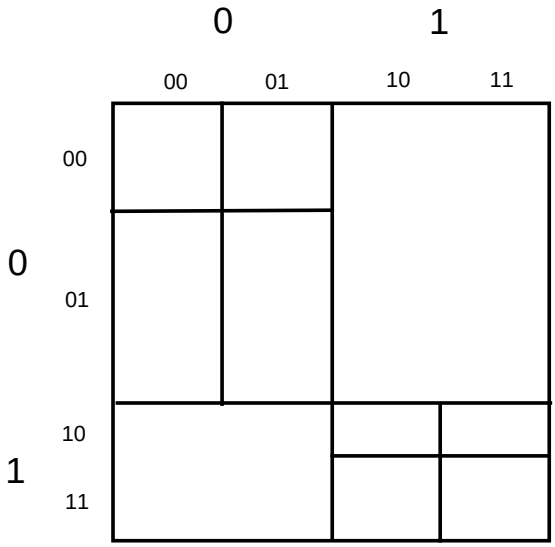


Fig. 2.

We can continue to work on $l_e(k_1, i, j+2)$ and $l_f(i, j+2, k_2)$, $l_e(k_1, i, j+3)$ and $l_f(i, j+3, k_2)$, ..., $l_e(k_1, i, j+c_1 \log n)$ and $l_f(i, j+c_1 \log n, k_2)$. Note the difference between here and the algorithm we gave before in the paper. Before we can only go to $l_e(k_1, i, j+r_2 \log n/\log\log n)$ and $l_f(i, j+r_2 \log n/\log\log n, k_2)$ (i.e. combining about $\log n/\log\log n$ columns (rows) of E_1 (F_1)), now we can go to $l_e(k_1, i, j+c_1 \log n)$ and $l_f(i, j+c_1 \log n, k_2)$ (i.e. combining about $\log n$ columns (rows) of E_1 (F_1)). This is the key for us to remove a factor of $\log\log n$ in time complexity.

In each partitioned division the winning index for the shortest paths is the same. The remaining computation basically follows the algorithm given before in the paper. First, for each row in E_1 and consecutive $r_3 \log n/\log\log n$ columns (consecutive in matrix F_1) we use a word w_1 to indicate the $r_3 \log n/\log\log n$ winning indices. Now compare words w_i (obtained for the same row in E_i and the same columns in F_i). If w_i and w_j are equal we then combine them into one word (removing half of the indices) by table lookup. We keep doing this until we combined $\log n/\log\log n$ words into one word. Thus now each word has $r_3 \log n/\log\log n$ winning indices and they are combined from $\log^3 n/(\log\log n)^2$ indices. Thus thereafter we can disassemble the indices from the word and the remaining computation shall take $O(n^3 \log\log n/\log^2 n)$ time.

Theorem: All pairs shortest paths can be computed in $O(n^3 \log\log n/\log^2 n)$ time.

As pointed by Chan that $O(n^3/\log^2 n)$ may be the final chapter and we are $\log\log n$ factor shy of this. We will leave this to future research. Also a referee

pointed out that if our scheme can be recursively exploited like in Strassen's algorithm [18] then $O(n^{2+\epsilon})$ time may be achieved.

Acknowledgment

The first author would like to thank Professor Timothy M. Chan for sending him the paper [5] upon his request. We are also thankful to referees' helpful comments on the paper.

References

1. A. V. Aho, J. E. Hopcroft, J. D. Ullman. The Design and Analysis of Computer Algorithms, Addison-Wesley, Reading, MA, 1974.
2. A. V. Aho, J. E. Hopcroft, J. D. Ullman. Data Structures and Algorithms, Addison-Wesley, Reading, MA, 1983.
3. S. Albers, T. Hagerup. Improved parallel integer sorting without concurrent writing. *Information and Computation* 136, 25-51(1997).
4. K.E. Batcher. Sorting networks and their applications. *Proc. 1968 AFIPS Spring Joint Summer Computer Conference*, 307-314(1968).
5. T.M. Chan. More algorithms for all-pairs shortest paths in weighted graphs. *Proc. 2007 ACM Symp. Theory of Computing*, 590-598(2007).
6. T.M. Chan. All-pairs shortest paths with real weights in $O(n^3/\log n)$ time. *Proc. 9th Workshop Algorithms Data Structures, Lecture Notes in Computer Science*, Vol. 3608, Spriner-Verlag, 318-324(2005).
7. W. Dobosiewicz. A more efficient algorithm for min-plus multiplication. *Inter. J. Comput. Math.* **32**, 49-60(1990).
8. M. L. Fredman. New bounds on the complexity of the shortest path problem. *SIAM J. Computing* **5**, 83-89(1976).
9. M. L. Fredman, R. Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *Journal of the ACM*, 34, 596-615, 1987.
10. Z. Galil, O. Margalit. All pairs shortest distances for graphs with small integer length edges. *Information and Computation*, 134, 103-139(1997).
11. Y. Han. An $O(n^3(\log\log n/\log n)^{5/4})$ time algorithm for all pairs shortest paths. *Algorithmica* 51, 4, 428-434(August 2008).
12. Y. Han. Improved algorithms for all pairs shortest paths. *Information Processing Letters*, 91, 245-250(2004).
13. Y. Han. A note of an $O(n^3/\log n)$ time algorithm for all pairs shortest paths. *Information Processing Letters*, 105, 114-116(2008)..
14. S. Pettie. A faster all-pairs shortest path algorithm for real-weighted sparse graphs. *Proceedings of 29th International Colloquium on Automata, Languages, and Programming (ICALP'02), LNCS Vol. 2380*, 85-97(2002).
15. S. Pettie, V. Ramachandran. A shortest path algorithm for real-weighted undirected graphs. *SIAM J. Comput.*, Vol. 34, No. 6, 1398-1431(2005).
16. P. Sankowski. Shortest paths in matrix multiplication time. *Proceedings of 13th Annual European Symposium on Algorithms.* Lecture Notes in Computer Science 3669, 770-778(2005).
17. R. Seidel. On the all-pairs-shortest-path problem in unweighted undirected graphs. *J. Comput. Syst. Sci.*, 51, 400-403(1995).

18. V. Strassen. Gaussian elimination is not optimal. *Numerische Mathematik*, 14(3):354-356(1969).
19. T. Takaoka. A new upper bound on the complexity of the all pairs shortest path problem. *Information Processing Letters* **43**, 195-199(1992).
20. T. Takaoka. An $O(n^3 \log\log n / \log n)$ time algorithm for the all-pairs shortest path problem. *Information Processing Letters* 96, 155-161(2005).
21. M. Thorup. Undirected single source shortest paths with positive integer weights in linear time. *Journal of ACM*, 46(3), 362-394(1999).
22. R. Yuster, U. Zwick. Answering distance queries in directed graphs using fast matrix multiplication. *46th Annual IEEE Symposium on Foundations of Computer Science . IEEE Comput. Soc. 2005*, pp. 389-96. Los Alamitos, CA, USA.
23. U. Zwick. All pairs shortest paths using bridging sets and rectangular matrix multiplication. *Journal of the ACM*, vol.49, no.3, May 2002, pp. 289-317.
24. U. Zwick. A slightly improved sub-cubic algorithm for the all pairs shortest paths problem. Proceedings of ISAAC 2004, Lecture Notes in Computer Science, Vol. 3341, Springer, Berlin, 921-932(2004).