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, 5] 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 [8], where m is the number of edges of the graph. Pettie showed [13] an algorithm with time complexity $O(mn + n^2 \log\log n)$. See [14] for recent development. There are also results for all pairs shortest paths for graphs with integer weights[9, 15, 16, 19–21]. Fredman gave the first subcubic algorithm [7] 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 $O(n^3 (\log\log n / \log n)^{1/2})$ [17]. Dobosiewicz [6] 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 $O(n^3(\log\log n/\log n)^{5/7})$ [11]. Later, Takaoka obtained an algorithm with time $O(n^3\log\log n/\log n)$ [18] and Zwick gave an algorithm with time $O(n^3\sqrt{\log\log n}/\log n)$ [22]. Chan gave an algorithm with time complexity of $O(n^3/\log n)$ [4]. Chan's algorithm does not use tabulation and bit-wise parallelism. His algorithm also runs on a pointer machine.

What subsequently happened was very interesting. Takaoka thought that $O(n^3/\log n)$ could be the ultimate goal and raised the question [18] 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 [4]. However, Han showed an algorithm with $O(n^3(\log\log n/\log n)^{5/4})$ time [10]. Because Han exhausted Takaoka's technique [17] in [10] Han thought that this result would be difficult to further improve on (see [10] for Han's reasoning). However, Chan came up with an algorithm with time complexity $O(n^3(\log\log n)^3/\log^2 n)$ [3]. Chan [3] believed that $O(n^3/\log^2 n)$ was 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 from 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 [7], Takaoka's algorithm [17], Han's algorithm [10], Chan's algorithm [3].

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 $A_0, A_1, , A_{t_1-1}$ each having dimension $n\times n/t_1$, where $t_1 = n^{1-\Delta_1}$ and Δ_1 is a constant to be determined later. We divide the second $n\times n$ matrix B into t_1 submatrices $B_0, B_1, ..., B_{t_1-1}$

each having dimension $n/t_1 \times n$. Therefore, $C = AB = A_0B_0 + A_1B_1 + ... + A_{t_1-1}B_{t_1-1}$, 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 an $n/t_1 \times n$ 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 $t_2 = (n/t_1)/(\Delta_2 \log n/\log\log n)$ submatrices $E_0, E_1, ..., E_{t_2-1}$ each having dimension $n \times (\Delta_2 \log n/\log\log n)$, where Δ_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 $(\Delta_2 \log n/\log\log n) \times n$. Now $EF = E_0F_0 + E_1F_1 + ... + E_{t_2-1}F_{t_2-1}$.

In the following, we will first consider the computation of E_0F_0 and then the computation of EF (or A_0B_0). Thereafter, it is straightforward to see that it takes $O(n^2t_1)$ time to get the all-pairs shortest path of the input graph.

Let $E_0 = (e_{ij})$ and $F_0 = (f_{ij})$. We will first, for each $0 \leq i, j < \Delta_2 \log n/\log\log n$, sort $f_{ik}-f_{jk}$, $k = 0, 1, ..., n-1$. After sorting, each $f_{ik}-f_{jk}$ has a rank in $[0, n-1]$. We then give $f_{ik} - f_{jk}$ a label $l_f(i, j, k) = l$ if $f_{ik} - f_{jk}$ has rank in interval $[ln/\log^9 n, (l + 1)n/\log^9 n)$. $l_f(i, j, k)$ uses $9 \log\log n$ bits. For each $e_{kj} - e_{ki}$, we will give it label $l_e(k, i, j) = l$ if $f_{ik_1} - f_{jk_1} \leq e_{kj} - e_{ki} < f_{ik_2} - f_{jk_2}$, where $f_{ik_1} - f_{jk_1}$ has rank (not label) $ln/\log^9 n$ and $f_{ik_2} - f_{jk_2}$ has rank $(l+1)n/\log^9 n$. $l_e(k, i, j)$ also uses $9 \log\log n$ bits.

According to Fredman [7] and Takaoka [17], if the labels of $e_{k_1j} - e_{k_1i}$ and $f_{ik_2} - f_{jk_2}$ are different, then we can determine either $e_{k_1i} + f_{ik_2} < e_{k_1j} + f_{jk_2}$ or $e_{k_1i} + f_{ik_2} > e_{k_1j} + f_{jk_2}$. Say $e_{k_1j} - e_{k_1i}$ 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 $e_{k_1j} - e_{k_1i} < f_{ik_2} - f_{jk_2}$ $(e_{k_1j} - e_{k_1i} > f_{ik_2} - f_{jk_2})$ and therefore $e_{k_1j} + f_{jk_2} < e_{k_1i} + f_{ik_2}$ $(e_{k_1j} + f_{jk_2} > e_{k_1i} + f_{ik_2})$. Note that when their labels are the same, we cannot determine this. However, for fixed i and j, only a fraction (i.e. $1/\log^9 n$) of the total number of $(f_{ik_2} - f_{jk_2})$'s, $0 \leq k_2 < n$, are undetermined for each $e_{k_1j} - e_{k_1i}$ for a fixed i, j, k_1. This is because for each $e_{k_1j} - e_{k_1i}$, its label $l_e(k_1, i, j)$ is fixed and only $n/\log^9 n$ of $f_{ik_2} - f_{jk_2}$'s, $0 \leq k_2 < n$, can have the same value $l_f(i, j, k_2)$ that is equal to $l_e(k_1, i, j)$. Note that these n comparisons of $l_f(i, j, k_2)$, $0 \leq k_2 < n$, with $l_e(k_1, i, j)$ for

fixed k_1, i, j are included in the n^3 comparisons in the straightforward distance matrix multiplication.

The above observation then generalizes to all comparisons of $l_e(k_1, i, j)$ and $l_f(i, j, k_2)$, $0 \leq k_1, k_2 < n$, for fixed i, j, and $1/\log^9 n$ of these pairs are undetermined. Then we further generalize it to all comparisons of $l_e(k_1, i, j)$ and $l_f(i, j, k_2)$, $0 \leq i, j < \Delta_2 \log n/\log\log n$ and $i \neq j$, $0 \leq k_1, k_2 < n$ and $1/\log^9 n$ of these pairs are undetermined. Note here we follow Fredman's algorithm [7] and compare every pair of distinct indices i and j. Thus, the total number of comparisons for multiplying E_0 and F_0 is $O(n^2(\Delta_2 \log n/\log\log n)^2)$ instead of the straightforward distance matrix multiplication approach that has only $n^2\Delta_2 \log n/\log\log n$ comparisons.

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 running time of this separate computation is as follows: There are $w = O(n^3 \log n/\log\log n)$ pairs of $l_e(k_1, i, j)$ and $l_f(i, j, k_2)$. Here k_1 and k_2 each take value in $[0..n-1]$ and thus have a factor of $x = n^2$. For each fixed pair of k_1 (a row in A) and k_2 (a column in B) there are $y = n\log\log n/(\Delta_2 \log n)$ pairs of E_i and F_i ($0 \leq i < y$) that contain the element in the k_1-th row of A and k_2-th column of B. Now first fixed k_1, k_2 and then fix a pair of E_i and F_i picked for k_1 and k_2, there are $z = O((\Delta_2 \log n/\log\log n)^2)$ pairs of $l_e(k_1, i, j)$ and $l_f(i, j, k_2)$. The distance multiplication of a row in E_0 with a column in F_0 is done by comparing every pair of distinct indices following Fredman's algorithm [7] and there are z pairs of them. This means $w = xyz = O(n^3 \log n/\log\log n)$. Because $1/\log^9 n$ of these pairs are in a separate computation, the computation takes $O((n^3 \log n/\log\log n) \cdot (1/\log^9 n)) = O(n^3/\log^8 n)$ time.

In the following, we use several precomputed tables. Each precomputed table has size of n^c for a constant $0 < c < 1$ and we can enforce this constant c to be as small as we wish. Thus, the precomputation of such a table takes $O(n)$ time because the computation takes a polynomial of n^c time.

3 Compute an $O(n^3(\log\log n/\log n)^2)$ Word Representation of the Shortest Paths

In this section, we discuss how to compute a representation of all shortest paths using $O(n^3(\log\log n/\log n)^2)$ computer words each having $\log n$ bits. These computer words tell us the indices of the shortest paths. We first do a pairwise comparison between the indices for a fixed row of E_0 and a fixed column of F_0. For each index, if it wins over all other indices we will give it 1, and if it loses to any other index we will give it 0. Although this can be done without doing pairwise comparisons between different indices, we do it because of the nature of our algorithm. This follows from the nature of Fredman's algorithm [7]. This gives $\Delta_2 \log n/\log\log n$ bits because there is one bit for each index. Note that among these $\Delta_2 \log n/\log\log n$ bits there is only one 1 and the other bits are 0's. For computing AB, we generated n^3 bits. Among these bits, $n^3/(\Delta_2 \log n/\log\log n)$ bits are 1's and the other bits are 0's. Then, we find ways to pack $O(\log n/\log\log n)$ 1's into a word and this will give $O(n^3/(\log n/\log\log n)^2)$ words. The problem is that during this computation and packing how could we avoid the n^3 bits of 0's and 1's before packing being spread to $\Omega(n^3/(\log n/\log\log n)^2)$ words for if this is to happen the time complexity will become $\Omega(n^3/(\log n/\log\log n)^2)$. As will be seen in this section we successfully avoid this problem and pack the 1's to $O(n^3/(\log n/\log\log n)^2)$ words. As will be seen, we will pack $L = (\Delta_2 \log n/\log\log n)(\epsilon \log n/\log\log n)$ bits of 0's and 1's, for a small constant ϵ, into one word. These L bits are the result of comparing a fixed index i with $\Delta_2 \log n/\log\log n - 1$ indices and get a bit 1 if index i wins over all these $\Delta_2 \log n/\log\log n - 1$ indices and get a bit 0 if index i loses to any one of these $\Delta_2 \log n/\log\log n - 1$ indices, for one row of E_0 and L columns of F_0.

For the $\binom{\Delta_2 \log n/\log\log n}{2}$ comparisons $l_e(k_1, i, j)$ and $l_f(i, j, k_2)$, for fixed k_1, k_2 and $0 \leq i, j < \Delta_2 \log n/\log\log n$ and $i \neq j$, that we are going to do, we partition them into $\Delta_2 \log n/\log\log n$ groups. The i-th group is the comparisons of $l_e(k_1, i, j)$ and $l_f(i, j, k_2)$, $0 \leq j < \Delta_2 \log n/\log\log n$ and $j \neq i$.

For fixed k_1 and i, we pack $l_e(k_1, i, j)$, $j = 0, 1, ..., \Delta_2 \log n / \log\log n - 1$ and $j \neq i$, into one word and call it $l_e(k, i)$. This can be done when $\Delta_2 < 1/9$ because each $l_e(k_1, i, j)$ takes only $9 \log\log n$ bits. Also for fixed i and k_2 we pack $l_f(i, j, k_2)$, $j = 0, 1, ..., \Delta_2 \log n / \log\log n - 1$ and $j \neq i$, into one word and call it $l_f(i, k_2)$. By comparing $l_e(k_1, i)$ and $l_f(i, k_2)$ we are in fact making $\Delta_2 \log n / \log\log n - 1$ comparisons of index i with indices $j = 0, 1, ..., \Delta_2 \log n / \log\log n - 1$ and $j \neq i$. If index i should be chosen over all the other $\Delta_2 \log n / \log\log n - 1$ indices we will let the result of comparing $l_e(k_1, i)$ and $l_f(i, k_2)$ to be 1, otherwise we will let it be 0. Therefore 1 means that index i wins and 0 says that index i loses.

The comparison of $l_e(k_1, i)$ and $l_f(i, k_2)$ can be 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. Each $l_e(k_1, i)$ ($l_f(i, k_2)$) contains $\Delta_2 \log n / \log\log n - 1$ $l_e(k_1, i, j)$'s ($l_f(i, j, k_2)$'s) with $0 \leq j < \Delta_2 \log n / \log\log n$ and $j \neq i$ and each of $l_e(k_1, i, j)$ ($l_f(i, j, k_2)$) takes $9 \log\log n$ bits. Here we need that $\Delta_2 < 1/18$. We then index this word into a precomputed table T to get the result of either 0 or 1. Table T can be precomputed in $O(n^{18\Delta_2}(\log n / \log\log n))$ time for $n^{18\Delta_2}$ is the size of the table and each table entry can be computed in time $O(\log n / \log\log n)$ by comparing $\Delta_2 \log n / \log\log n$ pairs of $l_e(k_1, i, j)$ and $l_f(i, j, k_2)$, $j = 0, 1, ..., \Delta_2 \log n / \log\log n - 1$. We can choose $\Delta_2 < 1/19$ so that the precomputation time for constructing T is $O(n)$.

Consider all the comparisons of $l_e(k_1, i)$ and $l_f(i, k_2)$, $0 \leq i < \Delta_2 \log n / \log\log n$, for fixed k_1, k_2. These comparisons produce $\Delta_2 \log n / \log\log n$ bits and among them only one bit is 1 and the other bits are 0's. This creates the sparseness which we will utilize later.

Note that $l_e(k, i)$ has $(\Delta_2 \log n / \log\log n) \cdot 9 \log\log n = 9\Delta_2 \log n < (9/19) \log n$ bits. $l_e(k, i), k = 1, 2, ..., n$, can then be sorted into $t_3 = 2^{9\Delta_2 \log n} = n^{9\Delta_2}$ blocks such that each block has the same $l_e(k, i)$ value.

We compare $l_e(k_1, i)$ for fixed k_1 and i and each of $l_f(i, k_2)$, $1 \leq k_2 \leq n$, and this produces n bits of 0's and 1's. We place these n bits into a $1 \times n$ vector with the k_2-th bit in the vector representing the result of comparing $l_e(k_1, i)$ and $l_f(i, k_2)$.

After sorting, all $l_e(k,i)$'s (for fixed i and $0 \leq k < n$) in a block have the same value. Thus, we can use one such $l_e(k,i)$ as the representative for all these same valued $l_e(k,i)$'s in the same block. Because there are $n^{9\Delta_2}$ blocks we have $n^{9\Delta_2}$ representative $l_e(k,i)$'s (fixed i and $n^{9\Delta_2}$ different k's). For each such representative $l_e(k,i)$, we compare it with $l_f(i,k_2)$, $0 \leq k_2 < n$, to produce a $1 \times n$ vector of binary bits as we stated above. For $t_3 = n^{9\Delta_2}$ representatives we get t_3 $1 \times n$ vectors.

Now let i vary from 0 to $\Delta_2 \log n / \log\log n - 1$ and for each value of i we get t_3 $1 \times n$ vectors, thus for all values of i we get $\Delta_2(\log n/\log\log n)t_3$ vectors. These vectors can be computed in $O(\Delta_2(\log n/\log\log n)t_3 n)$ time (one step gets one bit by the above table lookup computation).

On average, for each n bit vector v, there are only $n/(\Delta_2 \log n/\log\log n)$ bits that are 1's and the remaining bits are 0's due to our previous sparseness claim.

We take an above computed vector v and first divide it into n/L small vectors, where $L = (\Delta_2 \log n/\log\log n)(\epsilon \log n/\log\log n)$, each small vectors has dimension $1 \times L$, and ϵ is a constant to be determined later. On the average, each small vector has $\epsilon \log n/\log\log n$ bits which are 1's. If a small vector v' has between $(t-1)\epsilon \log n/\log\log n$ and $t\epsilon \log n/\log\log n$ bits of 1's, we will make a set V of t small vectors each having L bits and containing a subset of no more than $\epsilon \log n/\log\log n$ 1's from v' (among these t created small vectors $t-1$ of them each having $\epsilon \log n/\log\log n$ bits of 1's the last small vector has the remaining 1's). In doing so we at most double the number of small vectors because we can say that each small vector having $\epsilon \log n/\log\log n$ 1's is a filled small vector and each small vector having less than $\epsilon \log n/\log\log n$ 1's is an unfilled small vector. The $t_4 = ((\Delta_2(\log n/\log\log n)t_3) \cdot n/L$ (the number of vectors times the number of small vectors in each vector) small vectors convert to at most t_4 filled small vectors (because there are at most $t_4\epsilon \log n/\log\log n$ bits total that are 1's in all vectors or in all small vectors) and another (at most) t_4 unfilled small vectors (because each original small vector produces at most one unfilled small vector).

The multiplication of each row of E_0 with all columns of F_0 results in $\Delta_2 \log n/\log\log n$ vectors (Note here k_1 is fixed and we are not dealing with

blocks of $l_e(k,i)$'s. Each $l_e(k_1,i)$ for fixed k_1 and i and all $l_f(i,k)$'s, $0 \leq k < n$, result in one vector) having a total of $n\Delta_2 \log n/\log\log n$ bits with only n bits of 1's because of sparseness, they will result in $\Delta_2 \log n/\log\log n$ vectors and $2n(\Delta_2 \log n/\log\log n)/L = 2n/(\epsilon \log n/\log\log n)$ small vectors, where factor 2 is due to the reasoning of (at most) doubling of small vectors we explained before.

For fixed i (a row of F_0) (and therefore $l_f(i,k_2)$, $0 \leq k_2 < n$) and fixed value of $l_e(k,i)$'s (here fixed i) (a block obtained after sorting) we formed $2n/L$ small vectors each having L bits with no more than $\epsilon \log n/\log\log n$ bits are 1's. Therefore each small vector can be represented by a word (with no more than $\log n$ bits) when ϵ is small. This is so because $\sum_{t=0}^{\epsilon \log n/\log\log n} \binom{L}{t} < n^{4\epsilon} < n$ when we take $\epsilon < 1/4$. We first form these $2n/L$ words for each vector (on the average) and then duplicate these words for all rows in the block (all the same valued $l_e(k,i)$'s in the same 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. This duplicating is the key for us to avoid the spreading of bits into $\Omega(n^3/(\log n/\log\log n)^2)$ words.

Thus for the multiplication of E_0F_0 we obtained $2n^2/(\epsilon \log n/\log\log n)$ words because each row of E_0 compared with all columns of F_0 produced $abc = 2n/(\epsilon \log n/\log\log n)$ words, where $a = 2$ is the doubling factor, $b = n/L$ is the number of columns in F_0 divided by the number of bits in a small vector (i.e. the number of small vectors produced from each vector), $c = \Delta_2 \log n/\log\log n$ is the number of columns in E_0 which is also the number of rows in F_0. And for the multiplication of A_0B_0 we obtained $de = 2n^{2+\Delta_1}/L$ words, where $d = n^{\Delta_1}/(\Delta_2 \log n/\log\log n)$ is the number of E_i's in A_0 or the number of F_i's in B_0 and $e = 2n^2/(\epsilon \log n/\log\log n)$ is the number of words produced for E_0F_0. And therefore for the multiplication of AB we have obtained $fg = O(n^3(\log\log n/\log n)^2)$ words, where $f = t_1 = n^{1-\Delta_1}$ is the number of A_i's in A or the number of B_i's in B and $g = de = 2n^{2+\Delta_1}/L$ is the number of words produced for each A_iB_i, and computation thus far takes $O(n^3(\log\log n/\log n)^2)$ time.

4 Combining Words

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 (there are $n^3/(\Delta_2 \log n/\log\log n)$ indices encoded into these words because the information known to us so far is that there is one index remaining for each $E_i[k_1, *]F_i[*, k_2]$ for fixed k_1 and k_2). Thus we will combine these words to cut the number of words and the number of indices. If we reduce the number of words by a factor of $\log n$ then there will be only $O(n^3(\log\log n)^2/\log^3 n)$ words and $O(n^3(\log\log n/\log n)^2)$ indices remaining and we can therefore decode these remaining words to get these indices out and compare them straightforwardly.

Note that because of the way we create small vectors from vectors and we code each small vector into a word, a word contains no more than $\epsilon \log n/\log\log n$ indices (1's) for $\epsilon \log n/\log\log n$ columns of B. These indices in a word are for different columns of B and no two of them are for the same column of B. This is because a vector of n bits corresponding to n columns of B with one bit in the vector corresponding to one column of B. Note each bit in a word (and in a vector) indicates whether a particular index i wins (1 indicates win and 0 indicates lose) among $\Delta_2 \log n/\log\log n$ indices. An i index in a column of F_0 could win and another i index in a different column of F_0 could lose. This i is the same for all bits in the word and the vector.

We will combine two words into one word at a time. Each word corresponds to L bits (in which there are at most $\epsilon \log n/\log\log n$ 1's) and therefore corresponds to L columns of B. We need to ensure the following properties.

First property: we need to combine two words corresponding to the same columns of B (of course these words should be for the $l_e(k, i)$'s with fixed k and varying i's) because words corresponding to different columns of B cannot be combined.

Second property: we need to combine two words that have the same pattern. That is to say if word w_1 represents 01001 then we need to combine it with another word w_2 that also represents 01001. In this way the index for the first 1 in w_1 is compared with the index for the first 1 in w_2 and these two 1's represent

different indices for the same column of B (and one row of A). At the same time the index for the second 1 in w_1 is compared with the index of the second 1 in w_2 because they are for the same column of B. If we were to combine 01001 with 10010 there would be problems because theoretically we can have 11011 as the combined result but then the number of 1's in a word will increase and that will destroy our encoding restriction that no more than $\epsilon \log n / \log\log n$ 1's are encoded in a word. If we break this encoding restriction then we cannot encode L bits in a word. Thus we cannot combine 01001 with 10010. To deal with the problem of bringing the same patterned words together we look at all words for the comparison of $l_e(k_1, i)$ and $l_f(i, k_2)$ for fixed k_1, and $k_2 \in [tL, (t+1)L)$ for a fixed t (this ensures the first property) and $i \in [t'n^{\Delta_1}, (t'+1)n^{\Delta_1})$ for a fixed t' and thus they belong to $A_{t'}$ and $B_{t'}$. There are n^{Δ_1} words of them. If we choose ϵ to be much smaller than Δ_1 (although both are constants) we can ensure that on the average many of these n^{Δ_1} words have the same pattern and thus we will not be in a situation where we cannot find two words of the same pattern to be combined. This is because each word has $\sum_{t=0}^{\epsilon \log n/\log\log n} \binom{L}{t} < n^{4\epsilon}$ different values, and here we need $n^{4\epsilon} < n^{\Delta_1}$. Thus we may take $\epsilon < \Delta_1/4$.

Third property: because the way we ensure the second property holds by using n^{Δ_1} words, these words represent n^{Δ_1} different indices (columns of A_0 and rows of B_0) and thus our previously used labels $l_e(k, i, j), l_f(i, j, k), l_e(k, i), l_f(i, k)$ cannot be directly used because they are for indices across E_0 and F_0 ($\Delta_2 \log n/\log\log n$ indices). Thus we need to revise our labeling scheme.

Throughout this section we focus on the computation of $A_0[0, 0..n^{\Delta_1}-1]B_0[0..n^{\Delta_1}-1, 0..L-1]$. That is we fix one row of A_0 and L columns of B_0. This pre-fixing is due to first and the third properies.

To combine words we need to consider only the words formed by $E_i[0, 0..(\Delta_2 \log n/\log\log n)-1]F_i[0..(\Delta_2 \log n/\log\log n)-1, 0..L-1]$ (ensuring the first property) (there are $\Delta_2 \log n/\log\log n$ resulting words out of this multiplication because each $l_e(0, j)$ is compared with all $l_f(j, k)$, $0 \le k < L$), $i = 0$, 1, ..., $n^{\Delta_1}/(\Delta_2 \log n/\log\log n) - 1$ (help ensuring second property). Thus there are n^{Δ_1} words produced for $\min_{i=1}^{n^{\Delta_1}/(\Delta_2 \log n/\log\log n)} E_i[0, 0..(\Delta_2 \log n/\log\log n)-1]$ $F_i[0..(\Delta_2 \log n/\log\log n)-1, 0..L-1]$. We need to reduce them to $n^{\Delta_1}/\log n$

words in $O(n^{\Delta_1})$ time and thereafter we can simply disassemble indices out of packed words and finish the remaining computation straightforwardly. This means that we condensed $n^{\Delta_1}(\epsilon \log n/\log\log n)$ indices (each word contains on the average $\epsilon \log n/\log\log n$ winning indices) into $n^{\Delta_1}/\log n$ words.

Because each word w contains a set S_w of no more than $\epsilon \log n/\log\log n$ columns (these columns have 1's and the other columns have 0's) in F_i there are $\sum_{l=1}^{\epsilon \log n/\log\log n} \binom{L}{l} \leq n^{4\epsilon}$ choices (that many different words to separate apart to ensure second property of grouping same valued words together), where c is a constant. When $\epsilon < \Delta_1/4$ there are many words among the n^{Δ_1} words having the same S_w sets (same pattern, second and third properties). 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 (same pattern) 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).

The scheme for combining words is a little complicated as we follow the third property. The complication of our algorithm comes from the fact that indices are encoded in $O(n^3(\log\log n/\log n)^2)$ words in Section 3. To deal with this encoding we have to design an algorithm that utilizes the special characteristics of the encoding.

We use a different labeling (should say ranking) scheme for the matrix $B_0 = (b_{ij})$ and $A_0 = (a_{ij})$ (here we temporarily consider A_0 and B_0 instead of a row of A_0 and L columns of B_0). We will, for each $0 \leq i, j \leq n^{\Delta_1} - 1$ and $i \neq j$, sort all $b_{ik} - b_{jk}$ together with all $a_{kj} - a_{ki}$, $k = 1, 2, ..., n$. For A and B the total time for sorting is $O(n^{2+\Delta_1}\log n)$ because the time for each pair of A_i and B_i is $O(n^{1+2\Delta_1}\log n)$ (A_i (B_i) has dimension $n \times n^{\Delta_1}$ ($n^{\Delta_1} \times n$) and every pair of cloumns (rows) of A_i (B_i) needs to be sorted) and there are $n^{1-\Delta_1}$ pairs of A_i and B_i. This gives the rank of $b_{ik} - b_{jk}$ ($a_{kj} - a_{ki}$) which we denote by $r_{b_0}(i, j, k)$ ($r_{a_0}(k, i, j)$). These ranks take value from 0 to $2n-1$ and have $\log n+1$ bits. There are $O(n^{2\Delta_1})$ choices of i, j pairs and for each of these choices (each pair of i and j, here about rows of B_0 and columns of A_0) and for each set $U_t = \{tL+1, tL+2, ..., (t+1)L\}$, $t = 0, 1, ..., n/L-1$ (values of k are taken from U_t), choose any subset of U_t containing no more than $\epsilon \log n/\log\log n$ elements (corresponds to 1's, here about columns of B_1) and there are no more than

$\sum_{l=1}^{\epsilon \log n/\log\log n} \binom{L}{l} < n^{4\epsilon}$ choices for a fixed t. Thus there are a total of $n^{2\Delta_1+4\epsilon}$ choices for all pairs of i and j for fixed t (There are $O(n^{2\Delta_1})$ choices of pairs of i and j. The reason we do not use the big-O notation for $n^{2\Delta_1+4\epsilon}$ is that we can adjust ϵ.).

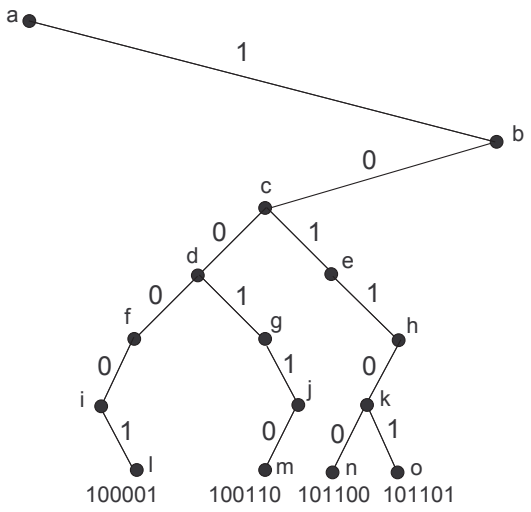
After fixing a choice in this previous paragraph we are looking at (a) a fixed choice of the pair i, j (there are $O(n^{2\Delta_1})$ choices), (b) a fixed choice (referred to as d) of the subset of U_t (there are $n^{4\epsilon}$ choices) and (c) no more than $\epsilon \log n/\log\log n$ ranks ($r_{b_0}(i,j,k)$'s, where i and j are fixed and k takes values over elements in the subset (representing 1's and there are no more than $\epsilon \log n/\log\log n$ of them) of U_t). We want to use a WORD of no more than $\log n$ bits (call it WORD to distinguish it from the previous word we used) to contain this information so that we can index into a precomputed table to decide which indices between i's and j's win (there are many pairs of i and j indices for various columns (at most $\epsilon \log n/\log\log n$ columns) of $B_0[0..n^{\Delta_1}/(\Delta_2 \log\log n) - 1, 0..L-1]$ and for each pair either i wins or j wins).

Note that straightforward packing will not work because it will take $O(\log^2 n/\log\log n)$ bits (thus cannot be stored in one WORD of $\log n$ bits) because a subset of U_t has up to $O(\log n/\log\log n)$ elements (corresponding to 1's) and each element corresponds to $O(\log n)$ bits of a $r_{b_0}(i,j,k)$. In the following we will show how to pack this information and store it into one WORD.

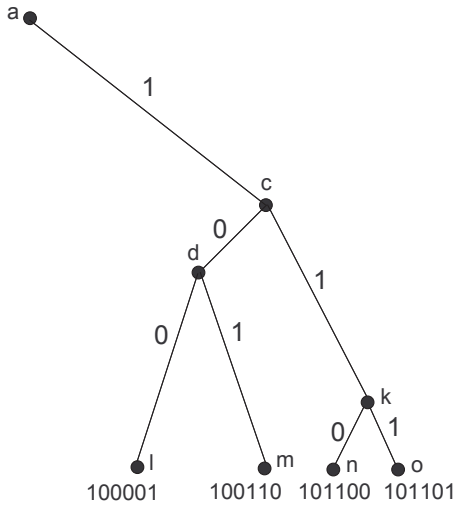
We first build a trie for the $\epsilon \log n/\log\log n$ ranks with each rank ranging from 0 to $2n-1$. An example of such a trie is shown in Fig. 1(a). This trie is a binary tree with a node having two children when there are ranks with the most significant bits differ at the node's level (see Fig. 1(a)). 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) with a weight indicating that $t-2$ edges (bits) are removed (using $\log\log n$ bits which is the logarithm of the height $(\log n + 1)$ of

the trie). We also label edge (v_1, v_t) using the label of (v_1, v_2) in the original trie to indicate that if at node v_1 and the next bit following from node v_1 is the label of (v_1, v_2) then we go to node v_t in the packed trie. Note here that we do not store the information about how these deleted edges branch to and later it will be seen that these branching information from these deleted edges need not be stored in the WORD. Also at leaves, we store only the relative address of k (having value between 0 and $L-1$) instead of the value of $r_{b_0}(i,j,k)$ (having value between 0 and $2n-1$). Note that when i and j are known then with the value of k we can find $r_{b_0}(i,j,k)$. Such a packed trie is shown in Fig. 1(c). The trie can be stored in a WORD W with $c \log n$ bits, where c is a constant less than 1. This is so because: the $\epsilon \log n / \log\log n$ relative addresses can be stored using $(\epsilon \log n / \log\log n) \log L$ bits and the packed trie has no more than $2\epsilon \log n / \log\log n$ nodes and edges and each edge has label with value from 1 to $\log\log n$ (logarithm of the height of the trie). Thus the number of bits needed to store this information is $(3\epsilon \log n / \log\log n) \log\log n))$ and it can be stored in a WORD.

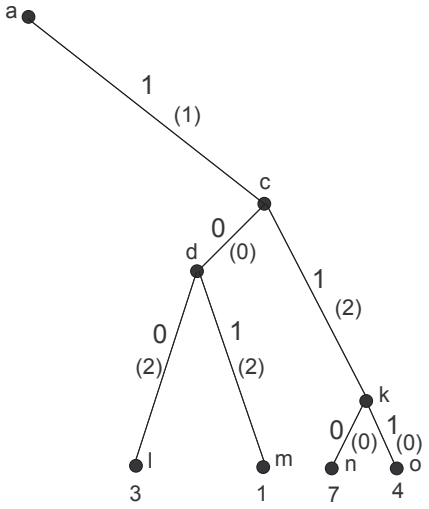
Note that $r_{a_0}(0,i,j)$ is to be compared with every leaf in the packed trie because different leaf represents different columns in B_0 and a row in A_0 is to be distance multiplied with every column of B_0. Now with $r_{a_0}(0,i,j)$, we first follow this packed trie in WORD W and reach a leaf l of the packed trie (by starting at the root, repeatedly deleting corresponding bits which has been removed from the trie to form the packed trie and following the path in the packed trie). In Fig. 1(d) we show such situations. Here we will use l to represent both the leaf and the relative address of k mentioned earlier. From l we get the value of k and we can then compare $r_{a_0}(0,i,j)$ and $r_{b_0}(i,j,k)$ to find the most significant bit where $r_{a_0}(0,i,j)$ and $r_{b_0}(i,j,k)$ 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, $r_{a_0}(0,i,j)$ and W (indexing into another precomputed table T_2 mentioned below) we can then figure out at which chain C of the original trie $r_{a_0}(0,i,j)$ "branches out". As in Fig. 1(d), rank 100101 will reach leaf m with rank 100110. The most significant bit 100101 and 100110 differ is the fifth most



(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.

significant bit. At that bit 100101 branches out to the left because that bit is 0. This says that leaf l has rank smaller 100101 and leaves m, n, o have rank larger than 100101 and this information can be found in the packed trie. Note that we do NOT need to know the value of C. We only need to know if a branch happens in C and whether it branches to the left or to the right (this information can be figured out with b, $r_{a_0}(0, i, j)$ and W. This branching condition determines which leaves in the trie have ranks smaller than $r_{a_0}(0, i, j)$ and which leaves in the trie have ranks not smaller than $r_{a_0}(0, i, j)$.

We can precompute a table T_1 and use the values of $r_{a_0}(0, i, j)$ and W to index into T_1 to get leaf l. (Note that $r_{a_0}(0, i, j)$ 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 because we are using these $\log n + 1$ bits to search down the packed trie.). We precompute a table T_2 to be indexed by values of b, $r_{a_0}(0, i, j)$ and W to figure out $r_{a_0}(0, i, j)$ "branches out" at which chain C of the original trie. $T_2[b, r_{a_0}(0, i, j), W]$ can be the columns where index i should be taken over index j.

We can store W as an array element in an array M as $M[i][j][t][d]$ (that is, let $M[i][j][t][d] = W$). Note that i, j, t, d were previously mentioned (i, j are the two comparing indices, t is for the choice of U_t and d is the packing of the L bits with no more than $\epsilon \log n / \log \log n$ 1's into a word). M has no more than $abc = n^{2\Delta_1+1+4\epsilon}$ elements, where $a < n^{2\Delta_1}$ are for the choices of i and j, $b < n$ is for the choice of t and $c = n^{4\epsilon}$ is for the choice of d. This is for B_0. For all B_i's, $0 \leq i < n^{1-\Delta_1}$, we need to create $n^{1-\Delta_1}$ tables (M's) and they have a total of $n^{2+\Delta_1+4\epsilon}$ elements. By choosing small values of Δ_1 and ϵ (say take $\Delta_1 = 1/16$ and $\epsilon < \Delta_1/4 = 1/64$) $n^{2+\Delta_1+4\epsilon} < n^{3-e}$ for a small constant e. Each $M[i][j][t][d]$ takes $\log^c n$ time to compute for a constant c because it involves assembling $r_{a_0}(i, j, k)$'s to form a packed trie, for the k's that correspond to 1's in d that is in U_t. We need not to pack $r_{a_0}(k, i, j)$'s because only one $r_{a_0}(k, i, j)$ is considered at a time.

Now for the above mentioned words w_1 and w_2 obtained in the Section 3 we can get t and d (both w_1 and w_2 are associated with the same t value because of the first property, w_1 and w_2 are associated with the same d value because of

the second property) (as we mentioned above that we can sort words to bring w_1 and w_2 together with the setting ϵ much smaller than Δ_1 (third property) to guarantee the second property). We can also get i from w_1 and j from w_2. Thus we can get $W = M[i][j][t][d]$. Now we use the values of $M[i][j][t][d]$ and $r_{a_0}(0, i, j)$ to index into precomputed tables T_1, T_2 to get rid of half of indices in both w_1 and w_2 (for every pair of indices from w_1 and w_2 gets rid one of them). We then update w_1 and w_2.

After one pass of combining two words into one word the average of the number of 1's in each word is halved and we then combine two words w_1, w_3 into one word, where the t value for w_1 is $2a$ for some a and the t value for w_2 is $2a + 1$ and the i value (index) is the same for w_1 and w_3. Thus we reduced the number of words by half and the average number of 1's in each word is still $\epsilon \log n / \log\log n$ and now each word covers $2L$ columns of B_0.

We now repeat the operation described so far in this section. We do $\log\log n$ iterations of these operations to reduce the number of words to $n^3/(\log^3 n/(\log\log n)^2)$. Thereafter we disassemble the indices from packed words 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^c with c being an arbitrarily small constant.

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

The main concept for achieving this reduction of complexity is to let E_0 (F_0) grow to $c_1 \log n$ columns (rows) for a small constant c_1 and thus we will have only a fractions of the n^3 bits ($1/(c_1 \log n)$ specifically) that will be 1's. Earlier we used $O(\log\log n)$ bits for each label and in order to assemble $c_1 \log n$ labels into a word we need to have each label contain only a constant number of bits. This seems impossible because there are $O(\log\log n)$ bits for each label. In this

section, we show how to accomplish this and ensure that, on average, a label has only a constant number of bits.

In our algorithm outlined in Sections 3 and 4 we partitioned the number of rows of E_0 and the number of columns of F_0 by a factor of $O(\log^9 n)$ at a time for each j, $0 \leq j < \Delta_2 \log n / \log\log n$, in $l_e(k_1, i, j)$ and $l_f(i, j, k_2)$ for fixed i and $0 \leq k_1, k_2 < n$. What we did essentially is that such partition is done for $j = 0$ resulting in columns (rows) of F_0 (E_0) be partitioned into $O(\log^9 n)$ parts. We then do $j = 1, 2, ..., l$, resulting in the columns (rows) of F_0 (E_0) be partitions into $O(\log^{9l} n)$ parts. And therefore l can at most go to $O(\log n / \log\log n)$ where we used $\Delta_2 \log n / \log\log n$. Because each $l_f(k, i, j)$ uses $O(\log\log n)$ bits we can pack $O(\log n / \log\log n)$ $l_f(k, i, j)$'s , $j = 0, 1, ..., \Delta_2 \log n / \log\log n$, into a word. This $O(\log n / \log\log n)$ levels of partitioning result in the loss of a factor of $\log\log n$ in time complexity. If we partition the number of rows of E_0 and the number of columns of F_0 by a constant factor at a time our algorithm would have $O(\log n)$ levels of partitioning and thus can remove another factor of $\log\log n$ from time complexity. When we use this approach the rows of E_0 and columns of F_0 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 are as follows.

Let $E_0 = (e_{ij})$ and $F_0 = (f_{ij})$. We will first, for each $0 \leq i, j < c_1 \log n$, sort $f_{ik} - f_{jk}$, $k = 1, 2, ..., n$. After sorting each $f_{ik} - f_{jk}$ has a rank in $[0, n-1]$. We then give $f_{ik} - f_{jk}$ a label $l_f(i, j, k) = l$ if $f_{ik} - f_{jk}$ has rank in $[ln/2, (l+1)n/2)$. $l_f(i, j, k)$ uses 1 bit. For each $e_{kj} - e_{ki}$ we will give it label $l_e(k, i, j) = l$ if $f_{ik_1} - f_{jk_1} \leq e_{kj} - e_{ki} < f_{ik_2} - f_{jk_2}$, where $f_{ik_1} - f_{jk_1}$ has rank (not label) $ln/2$ and $f_{ik_2} - f_{jk_2}$ has rank $(l+1)n/2$.

Now if $l_e(k_1, i, j) \neq l_f(i, j, k_2)$ 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 partition. In this way when we partitioned to $c_2 \log n$ levels for a constant c_2 then the area of undetermined divisions is n^2/n^{c_2}. See Fig. 2.

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. Once the undetermined divisions area reaches n^2/n^{c_2} we stop partitioning. 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.

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 $l_e(k_1, i, j + \Delta_2 \log n / \log\log n)$ and $l_f(i, j + \Delta_2 \log n / \log\log n, k_2)$ (i.e. combining about $\log n / \log\log n$ columns (rows) of E_0 (F_0)), 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_0 (F_0)) for a constant c_1. This is the key for us to remove a factor of $\log\log n$ in time complexity. The reason that we can go to level $c_1 \log n$ is that after going to level d the total area of the rectangles in Fig. 2 that can be further partitioned dominates and we will analyze this in next paragraph.

Note that we partition the $n \times n$ area in Fig. 2. once by drawing a horizontal line and a vertical line. The vertical line partitions the x-dimension into two equal parts but the horizontal line needs not necessarily to partition the y-dimension into two equal parts. After such partition we get 2 rectangles (the top right one and the bottom left one) with total area of $n^2/2$ that are resolves (i.e. the winning index is known) and the other 2 rectangles (the top left one and the bottom right one) with total area of $n^2/2$ that are undetermined. If we then draw a horizontal line and a vertical line in the undetermined rectangles we get 4 additional rectangles with total area of $n^2/4$ that are resolved and 4 rectangles with total area of $n^2/4$ remaining that are unresolved. If we do the partitioning step $c_2 \log n$ times for a $0 < c_2 < 1$ then we get 2^c rectangles with total area $n^2/2^c$ (the average area is $n^2/4^c$ per rectangle) that are resolved, for $c = 1, 2, ..., c_2 \log n$, and $2^{c_2 \log n}$ rectangles of total area $n^2/2^{c_2 \log n}$ (the average area is $n^2/4^{c_2 \log n}$ per rectangle) that are unresolved. This is the situation when

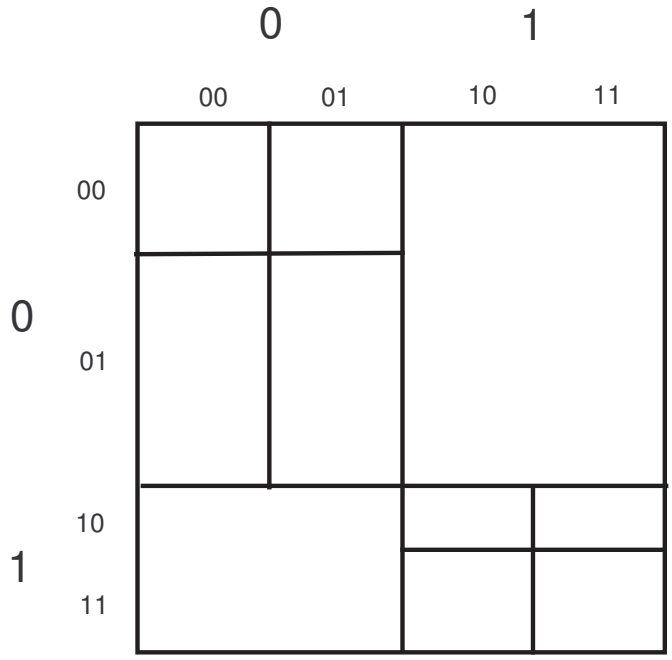


Fig. 2

we finish with $l_e(k_1, i, j)$ and $l_f(i, j, k_2)$ for fixed i and j and $0 \le k_1, k_2 < n$. We use the notation $N = (\sum_{i_1=1}^{c_2 \log n} 2^{i_1} R(n^2/4^{i_1})) + 2^{c_2 \log n} I(n^2/4^{c_2 \log n})$ to denote this fact, where $R(n^2/4^{i_1})$ indicates the average area per resolved rectangle and $I(n^2/4^{c_2 \log n})$ is the average area per undetermined rectangle.

We now do $l_e(k_1, i, j+1)$ and $l_f(i, j+1, k_2)$. This further divides each rectangle obtained in the above paragraph. The formula for N becomes

$$N = \sum_{i_1=1}^{c_2 \log n-1} \sum_{i_2=1}^{c_2 \log n-i_1} 2^{i_1+i_2} R(n^2/4^{i_1+i_2}) + \sum_{i_1=1}^{c_2 \log n-1} \sum_{i_2=c_2 \log n-i_1}^{c_2 \log n-i_1} 2^{i_1+i_2} I(n^2/4^{c_2 \log n})$$
$$= \sum_{i_1=1}^{c_2 \log n-1} \sum_{i_2=1}^{c_2 \log n-i_1} 2^{i_1+i_2} R(n^2/4^{i_1+i_2}) + \sum_{i_1=1}^{c_2 \log n-1} \sum_{i_2=c_2 \log n-i_1}^{c_2 \log n-i_1} 2^{c_2 \log n} I(n^2/4^{c_2 \log n})$$
$$= \sum_{i_1=1}^{c_2 \log n-1} \sum_{i_2=1}^{c_2 \log n-i_1} 2^{i_1+i_2} R(n^2/4^{i_1+i_2}) + (c_2 \log n - 1) 2^{c_2 \log n} I(n^2/4^{c_2 \log n}).$$

Note that index i_1 now goes to $c_2 \log n - 1$ instead of $c_2 \log n$ because index i_2 needs to divide at least once for every resolved rectangle remained after the dividing for i_1 for otherwise index i_2 is unresolved in the rectangle. Index i_2 needs to divide every resolved rectangle remained after the dividing of i_1 because we need to determine $l_e(k_1, i, j+1)$ and $l_f(i, j+1, k_2)$ for each resolved rectangle.

If the above dividing goes up to $l_e(k_1, i, j + c_1 \log n)$ and $l_f(i, j + c_1 \log n, k_2)$, where $c_1 < c_2$, we have the formula

$$N = \sum_{i_1=1}^{c_2 \log n-c_1 \log n} \sum_{i_2=1}^{c_2 \log n-c_1 \log n+1-i_1} \sum_{i_3=1}^{c_2 \log n-c_1 \log n+2-i_1-i_2} \cdots$$
$$\cdots \sum_{i_{c_1 \log n+1}=1}^{c_2 \log n-i_1-i_2-\cdots-i_{c_1 \log n}} 2^{i_1+i_2+i_3+\cdots+i_{c_1 \log n+1}} R(n^2/4^{i_1+i_2+i_3+\cdots+i_{c_1 \log n+1}})$$
$$+ \sum_{i_1=1}^{c_2 \log n-c_1 \log n} \sum_{i_2=1}^{c_2 \log n-c_1 \log n+1-i_1} \sum_{i_3=1}^{c_2 \log n-c_1 \log n+2-i_1-i_2} \cdots$$
$$\cdots \sum_{i_{c_1 \log n+1}=c_2 \log n-i_1-i_2-\cdots-i_{c_1 \log n}}^{c_2 \log n-i_1-i_2-\cdots-i_{c_1 \log n}} 2^{i_1+i_2+i_3+\cdots+i_{c_1 \log n+1}} I(n^2/4^{i_1+i_2+i_3+\cdots+i_{c_1 \log n+1}})$$
$$= \sum_{i_1=1}^{c_2 \log n-c_1 \log n} \sum_{i_2=1}^{c_2 \log n-c_1 \log n+1-i_1} \sum_{i_3=1}^{c_2 \log n-c_1 \log n+2-i_1-i_2} \cdots$$
$$\cdots \sum_{i_{c_1 \log n+1}=1}^{c_2 \log n-i_1-i_2-\cdots-i_{c_1 \log n}} 2^{i_1+i_2+i_3+\cdots+i_{c_1 \log n+1}} R(n^2/4^{i_1+i_2+i_3+\cdots+i_{c_1 \log n+1}})$$
$$+ \sum_{i_1=1}^{c_2 \log n-c_1 \log n} \sum_{i_2=1}^{c_2 \log n-c_1 \log n+1-i_1} \sum_{i_3=1}^{c_2 \log n-c_1 \log n+2-i_1-i_2} \cdots$$
$$\cdots \sum_{i_{c_1 \log n+1}=c_2 \log n-i_1-i_2-\cdots-i_{c_1 \log n}}^{c_2 \log n-i_1-i_2-\cdots-i_{c_1 \log n}} 2^{c_2 \log n} I(n^2/4^{c_2 \log n}) \qquad (1)$$

The quantity

$$N' = \sum_{i_1=1}^{c_2 \log n-c_1 \log n} \sum_{i_2=1}^{c_2 \log n-c_1 \log n+1-i_1} \sum_{i_3=1}^{c_2 \log n-c_1 \log n+2-i_1-i_2} \cdots$$
$$\cdots \sum_{i_{c_1 \log n+1}=c_2 \log n-i_1-i_2-\cdots-i_{c_1 \log n}}^{c_2 \log n-i_1-i_2-\cdots-i_{c_1 \log n}} 2^{c_2 \log n} I(n^2/4^{c_2 \log n})$$

denotes the area for the unresolved rectangles and it can be estimated using integration as

$$(\int_{i_1=1}^{c_2 \log n-c_1 \log n} \int_{i_2=1}^{c_2 \log n-c_1 \log n+1-i_1} \int_{i_3=1}^{c_2 \log n-c_1 \log n+2-i_1-i_2} \cdots$$
$$\cdots \int_{i_{c_1 \log n+1}=c_2 \log n-i_1-i_2-\cdots-i_{c_1 \log n}}^{c_2 \log n-i_1-i_2-\cdots-i_{c_1 \log n}} 1 di_{c_1 \log n+1} di_{c_1 \log n} \cdots di_1) \cdot 2^{c_2 \log n} I(n^2/4^{c_2 \log n}).$$

Replacing $I(n^2/4^{c_2 \log n})$ with $n^2/4^{c_2 \log n}$ this integration gives $N' < ((c_2 \log n)^{c_1 \log n}/(c_1 \log n)!)n^2/2^{c_2 \log n} \leq ((c_2 e/c_1)^{c_1 \log n} \log n)n^{2-c_2}$. Here e is the base of natural logarithm that comes from Stirling approximation and $\log n$ is the enlarged factor of $\sqrt{c_1 \log n}$ coming from Stirling's approximation. Let $c_2 = 1/4$ and let $c_1 = 1/32$ then we get that $N' < ((c_2 e/c_1)^{c_1 \log n} \log n)n^{2-c_2} < n^{15/8}$. N' is the unresolved area for $E_0[0..n-1, 0..c_1 \log n]F_0[0..c_1 \log n, 0..n]$. Thus for AB the unresolved area (indeterminacy) is bounded by $n^{23/8}$.

In each resolved partitioned division (rectangle) 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_0 and consecutive $(c_1 \log n)(\epsilon \log n/\log\log n)$ columns (consecutive in matrix F_0) we use a word w_1 to indicate the $\epsilon \log n/\log\log n$ winning indices. Now compare words w_i's (obtained for the same row in E_0 and the same columns in F_0). 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 $\epsilon \log n/\log\log n$ winning indices and they are combined from $O(\log^3 n/(\log\log n)^2)$ 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.

Acknowledgment

The first author would like to thank Professor Timothy M. Chan for sending him the paper [3] upon his request.

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. T.M. Chan. More algorithms for all-pairs shortest paths in weighted graphs. *Proc. 2007 ACM Symp. Theory of Computing*, 590-598(2007).
4. 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).
5. T. H. Corman, C. E. Leiserson, R. L. Rivest, C. Stein. Introduction to Algorithms, Third Edition, *MIT Press*, 2009.
6. W. Dobosiewicz. A more efficient algorithm for min-plus multiplication. *Inter. J. Comput. Math.* **32**, 49-60(1990).
7. M. L. Fredman. New bounds on the complexity of the shortest path problem. *SIAM J. Computing* **5**, 83-89(1976).
8. M. L. Fredman, R. Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *Journal of the ACM*, 34, 596-615, 1987.
9. Z. Galil, O. Margalit. All pairs shortest distances for graphs with small integer length edges. *Information and Computation*, 134, 103-139(1997).
10. 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).
11. Y. Han. Improved algorithms for all pairs shortest paths. *Information Processing Letters*, 91, 245-250(2004).
12. 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)..
13. 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).
14. S. Pettie, V. Ramachandran. A shortest path algorithm for real-weighted undirected graphs. *SIAM J. Comput.*, Vol. 34, No. 6, 1398-1431(2005).
15. 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).
16. R. Seidel. On the all-pairs-shortest-path problem in unweighted undirected graphs. *J. Comput. Syst. Sci.*, 51, 400-403(1995).
17. T. Takaoka. A new upper bound on the complexity of the all pairs shortest path problem. *Information Processing Letters* **43**, 195-199(1992).
18. 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).
19. M. Thorup. Undirected single source shortest paths with positive integer weights in linear time. *Journal of ACM*, 46(3), 362-394(1999).
20. 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.
21. 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.
22. 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).