Sub-cubic Cost Algorithms for the All Pairs Shortest Path Problem*

Tadao TAKAOKA

Department of Computer Science
Ibaraki University, Hitachi, Ibaraki 316, JAPAN
E-mail: takaoka@cis.ibaraki.ac.jp

Abstract. In this paper we give three sub-cubic cost algorithms for the all pairs shortest distance (APSD) and path (APSP) problems. The first is a parallel algorithm that solves the APSD problem for a directed graph with unit edge costs in $O(\log^2 n)$ time with $O(n^\mu/\sqrt{\log n})$ processors where $\mu = 2.688$ on an EREW-PRAM. The second parallel algorithm solves the APSP, and consequently APSD, problem for a directed graph with non-negative general costs (real numbers) in $O(\log^2 n)$ time with $o(n^3)$ subcubic cost. Previously this cost was greater than $O(n^3)$. Finally we improve with respect to M the complexity $O((Mn)^\mu)$ of a sequential algorithm for a graph with edge costs up to M into $O(M^{1/3}n^{(6+\omega)/3}(\log n)^{2/3}(\log\log n)^{1/3})$ in the APSD problem.

1 Introduction

The all pairs shortest path (APSP) problem is to compute shortest paths between all pairs of vertices of a directed graph with non-negative real numbers as edge costs. The all pairs shortest distance problem (APSD) is defined similarly, the word "paths" replaced by distances. Traditionally the latter was called the APSP problem. Alon, Galil and Margalit [1] gave a sub-cubic algorithm for the APSD problem for a graph with small integer edge costs. Recently Alon, Galil, Margalit and Naor [2] distinguished these two problems and faced a higher complexity for the APSP problem with the same class of graphs. The best time complexities for the APSD and APSP problems for undirected graphs with unit edge costs are given by Seidel [10]. On the other hand, the complexity for the all pairs shortest distance (APSD) problem with general edge costs was slightly improved by Takaoka [12], from that by Fredman [5], which is n^3 divided by polylog of n, very close to n^3. With this algorithm [12], we can solve the APSP problem at the same time, and hence no need to distinguish between APSD and APSP.

The technique for obtaining paths in [2] is based on the concept of witnesses for Boolean matrix multiplication. They compute the witnesses of Boolean matrix multiplication for (n, n) matrices in $O(n^\omega \log^c n)$ where $c < 5$ and $\omega = 2.376$. Using this result they solve other shortest path problems, attaching $\log^c n$ as factors to the complexities of corresponding distance problems.

* Partially supported by Grant-in-aid No. 07680332 by Monbusho Scientific Research Program and a research grant from Hitachi Engineering Co., Ltd.

In this paper we design a parallel algorithm for the APSD problem for a directed graph with unit edge costs with $O(\log^2 n)$ time (worst case) and $O(n^{(3+\omega)/2}/\sqrt{\log n})$ processors. This result is compared with a parallel algorithm for a directed graph with general edge costs in $O(\log \log n)$ expected time and $O(n^{2.5+\varepsilon})$ processors in [13] where ε is a small positive real number.

Next we improve the parallel algorithm for the APSP problem with general edge costs in [12] whose cost (= number of processors $\times$ time) is slightly above $O(n^3)$. The cost of $O(n^3 \log n)$ in Dekel, Nassimi and Sahni [4] was also improved by Han, Pan and Reif [8] into that still avove $O(n^3)$. The cost of our new algorithm is slightly below $O(n^3)$ and the time is polylog of n, that is, in NC.

Finally we present a sequential algorithm for a graph with edge cost up to M whose complexity is sub-cubic when $M = O(n^{0.624})$.

2 Basic definitions

A directed graph is given by $G = (V, E)$, where $V = \{0, \cdots, n-1\}$ and E is a subset of $V \times V$. The edge cost of $(i, j) \in E$ is denoted by d_{ij}. The (n, n) matrix D is one whose (i, j) element is d_{ij}. We assume that $d_{ij} \geq 0$ and $d_{ii} = 0$ for all i, j. If there is no edge from i to j, we let $d_{ij} = \infty$. The cost, or distance, of a path is the sum of costs of the edges in the path. The length of a path is the number of edges in the path. The shortest distance from vertex i to vertex j is the minimum cost over all paths from i to j, denoted by d^*_{ij}. Let $D^* = \{d^*_{ij}\}$. We call n the size of the matrices.

Let A and B are (n, n)-matrices. The three products are defined using the elements of A and B as follows:

(1) Ordinary multiplication over a ring $\quad C = AB$

$$c_{ij} = \sum_{k=0}^{n-1} a_{ik} b_{kj},$$

(2) Boolean matrix multiplication $\quad C = A \cdot B$

$$c_{ij} = \bigvee_{k=0}^{n-1} a_{ik} \wedge b_{kj},$$

(3) Distance matrix multiplication $\quad C = A \times B$

$$c_{ij} = \min_{0 \leq k \leq n-1} \{a_{ik} + b_{kj}\}.$$

The best algorithm [3] computes (1) in $O(n^\omega)$ time. To compute (2), we can regard Boolean values 0 and 1 in A and B as integers and use the algorithm for (1), and convert non-zero elements in the resulting matrix to 1. Therefore this

complexity is $O(n^\omega)$. If we have an algorithm for (3) with $T_D(n)$ ($T_P(n)$) time we can solve the APSD (APSP) problem in $O(T_D(n)\log n)$ ($O(T_P(n)\log n)$) time by the repeated squaring method, described as the repeated use of $A \Leftarrow A \times A$. In this description, $T_P(n)$ is the time for the algorithm to compute (3) with witnesses, that is, giving k that gives the minimum for c_{ij}. The witnesses of (2) is the matrix $W = \{w_{ij}\}$ where $w_{ij} = k$ for some k such that $a_{ik} \wedge b_{kj} = 1$. If there is no such k, $w_{ij} = 0$.

The definition of computing shortest paths is to give a path matrix of size n by which we can give a shortest path from i to j in $O(\ell)$ time where ℓ is the length of the path. More specifically, if $w_{ij} = k$ in the path (or witness) matrix $W = \{w_{ij}\}$, it means that the path from i to j goes through k. Therefore a recursive function $path(i,j)$ is defined by $(path(i,k),\ k,\ path(k,j))$ if $path(i,j) = k > 0$ and nil if $path(i,j) = 0$, where a path is defined by a list of vertices excluding endpoints. In the following sections, we record k in w_{ij} whenever we can find k such that a path from i to j is modified or newly set up by paths from i to k and from k to j.

3 All pairs shortest paths

We review the algorithm in [1] in this section. Let the costs of edges of the given graph be ones. Let $D^{(\ell)}$ be the ℓ-th approximate matrix for D^* defined by $d_{ij}^{(\ell)} = d_{ij}^*$ if $d_{ij}^{(\ell)} \leq \ell$, and $d_{ij}^{(\ell)} = \infty$ otherwise. Then we can compute $D^{(r)}$ by the following algorithm.

Algorithm 1 Shortest distances by Boolean matrix multiplication

```
1  A := {a_ij} where a_ij = 1 if d_ij ≤ 1, and 0 otherwise;
2  B := I;   {Boolean identity matrix}
3  for ℓ := 2 to r do begin
4     B := A · B; {Boolean matrix multiplication}
5     for i := 0 to n − 1 do for j := 0 to n − 1 do
6        if b_ij = 1 then d_ij^(ℓ) := ℓ else d_ij^(ℓ) = ∞;
7        if d_ij^(ℓ−1) ≤ ℓ then d_ij^(ℓ) := d_ij^(ℓ−1)
8  end.
```

In this algorithm, $D^{(\ell)}$ is computed in increasing order of ℓ. Note that $D^{(1)} = D$. Since we can compute line 4 in $O(n^\omega)$ time, the computing time of this algorithm is $O(rn^\omega)$.

The following algorithm in [1] for the APSD problem uses Algorithm 1 as an "accelerating phase" and repeated squaring as a "cruising phase."

Algorithm 2 Solving APSD

```
   {Accelerating phase}
 1 for ℓ := 2 to r do compute D^(ℓ) using Algorithm 1;
   {Cruising phase}
 2 ℓ := r;
```

3 **for** $s := 1$ **to** $\lceil \log_{3/2} n/r \rceil$ **do begin**
4 **for** $i := 0$ **to** $n-1$ **do**
5 Scan the i-th row of $D^{(\ell)}$ and find the smallest set of equal $d_{ij}^{(\ell)}$'s such that $\lceil \ell/2 \rceil \leq d_{ij}^{(\ell)} \leq \ell$ and let the set of corresponding indices j be S_i;
6 $\ell_1 := \lceil 3\ell/2 \rceil$;
7 **for** $i := 0$ **to** $n-1$ **do for** $j := 0$ **to** $n-1$ **do begin**
8 **if** $S_i \neq \emptyset$ **do** $m_{ij} := \min_{k \in S_i} \{d_{ik}^{(\ell)} + d_{kj}^{(\ell)}\}$ **else** $m_{ij} := \infty$
9 **if** $d_{ij}^{(\ell)} \leq \ell$ **then** $d_{ij}^{(\ell_1)} := d_{ij}^{(\ell)}$
10 **else if** $m_{ij} \leq \ell_1$ **then** $d_{ij}^{(\ell_1)} := m_{ij}$
11 **else** $\{m_{ij} > \ell_1\}$ $d_{ij}^{(\ell_1)} := \infty$
12 **end**;
13 $\ell := \ell_1$
14 **end**

Algorithm 2 computes $D^{(\ell)}$ from $\ell = 2$ to r in the accelerating phase spending $O(rn^3)$ time and computes $D^{(\ell)}$ for $\ell = r$, $\lceil \frac{3}{2} r \rceil$, $\left\lceil \frac{3}{2} \lceil \frac{3}{2} r \rceil \right\rceil$, $\cdots$, n' by repeated squaring in the cruising phase, where n' is the smallest integer in this series of ℓ such that $\ell \geq n$. The key observation in the cruising phase is that we only need to check S_i at line 8 whose size is not larger than $2n/\ell$. Hence the computing time of one iteration beginning at line 3 is $O(n^3/\ell)$. Thus the time of the cruising phase is given with $N = \lceil \log_{3/2} n/r \rceil$ by

$$O(\sum_{s=1}^{N} n^3/((3/2)^s r)) = O(n^3/r).$$

Balancing the two phases with $rn^\omega = n^3/r$ yields the time of Algorithm of $O(n^{(\omega+3)/2})$ with $r = O(n^{(3-\omega)/2})$.

When we have a directed graph G whose edge costs are between 0 and M where M is a positive integer, we can convert the graph G to $G' = (V', E')$ by adding auxiliary vertices $v_1, \cdots, v_{M-1}$ for $v \in V$. The edge set is also modified to E'. If $c(v, w) = \ell$, w is connected from $v_{\ell-1}$ in E' where $v_0 = v$. Obviously we can solve the problem for G by applying Algorithm 2 to G', which takes $O\left((Mn)^{(\omega+3)/2}\right)$ time.

Now witnesses can be kept at lines 4 – 7 of Algorithm 1 in $O(n^\omega \log^c n)$ time. At line 8 of Algorithm 2 witnesses are obtained by storing k in the witness matrix. This does not increase the order of complexity of the cruising phase. Thus we have the complexity of the APSP of $O\left(n^{(\omega+3)/2}\sqrt{(\log n)^c}\right)$. For the graph with edge costs of M or less, the complexity becomes $O\left((Mn)^{(\omega+3)/2}\sqrt{(\log(Mn))^c}\right)$.

4 Parallelization for graphs with unit costs

We design a parallel algorithm on an EREW-PRAM for a directed graph with unit edge costs. In this section and the next section, we mainly describe our

algorithm using a CREW-PRAM for simplicity. The overhead time to copy data in $O(\log n)$ time with a certain number of processors depending on each phase is absorbed in the dominant complexities. Let A be the adjacency matrix used in Algorithm 1. That is, $a_{ij} = 1$ if there is an edge from i to j and 0 otherwise. All diagonal elements are 1. There is a path from i to j of length $\leq \ell$ if and only if the (i, j) element A^ℓ is 1, where A^ℓ is the ℓ-th power of A by Boolean matrix multiplication. By repeated squaring, we can get A^ℓ ($\ell = 1, 2, 4, \cdots, n'$) with $\lceil \log_2 n \rceil$ Boolean matrix multiplications, where n' is the smallest integer in this series of ℓ such that $\ell \geq n$. These matrices give a kind of approximate estimation on the path lengths. That is, if the (i, j) element of A^{2^r} becomes 1 for the first time, we can say that the shortest path length from i to j is between $2^{r-1}+1$ and 2^r for $r \geq 1$. As r gets large, the gap between $2^{r-1}+1$ and 2^r gets large. As Gazit and Miller [6] observe we can fill the gap in increasing order of r in the following way. Let shortest paths up to 2^{r-1} be computed already. Then a shortest path from i to j, whose length is between $2^{r-1}+1$ and 2^r consists of a shortest path from i to k whose length is up to 2^{r-1} and a path from k to j of length 2^{r-1}. Formally we have the following algorithm, in which the approximation phase is shown for explanation purposes.

Algorithm 3 Shortest distances up to 2^R

{Approximation phase}
1 $A^{(1)} := A$; $\ell := 1$;
2 **for** $s := 1$ **to** R **do begin**
3 $A^{(2\ell)} := A^{(\ell)} \cdot A^{(\ell)}$; {Boolean matrix multiplication}
4 $\ell := 2\ell$
5 **end**;
{Gap filling phase}
6 $D^{(1)} := D$;
7 **for** $s := 1$ **to** R **do begin**
8 **for** $\ell := 2^{s-1} + 1$ **to** 2^s **do begin**
9 $A^{(\ell)} := A^{(\ell - 2^{s-1})} \cdot A^{(2^{s-1})}$; {Boolean matrix multiplication}
10 **for** $i := 0$ **to** $n-1$ **do for** $j := 0$ **to** $n-1$ **do**
11 **if** $a_{ij}^{(\ell)} = 1$ **then** $b_{ij}^{(\ell)} := \ell$ **else** $b_{ij}^{(\ell)} := \infty$
12 **end**;
13 **for** $i := 0$ **to** $n-1$ **do for** $j := 0$ **to** $n-1$ **do**
14 $d_{ij}^{(2^s)} := \min_{2^{s-1} < \ell \leq 2^s} \{b_{ij}^{(\ell)}\}$
15 **end**.

In this algorithm arrays $B^{(\ell)} = \{b_{ij}^{(\ell)}\}$ are used as working space to compute $d_{ij}^{(2^s)}$ at line 14. We fill the gaps for $A^{(\ell)}$, but do not do so for $D^{(\ell)}$, since we are only interested in $D^{(2^R)}$. The computing time of this algorithm is $O(2^R n^\omega)$. If we let $R = \lceil \log_2 n \rceil$, we can solve the APSD, but then the time becomes $O(n^{\omega+1})$, which is not efficient. If we substitute Algorithm 3 for Algorithm 1 in Algorithm 2 and let $2^R = n^{(3-\omega)/2}$, however, we can solve the APSD problem in $O(n^{(\omega+3)/2})$ time, which is on a par with Algorithm 2.

The merit of Algorithm 3 is that it is easy to parallelize. It is well-known that we can multiply two matrices over a ring in $O(\log n)$ time with $O(n^\omega)$ processors in parallel. We substitute Algorithm 3 for Algorithm 1 in Algorithm 2 and call the resulting algorithm Algorithm 2'. We can perform 2^{s-1} multiplications in parallel at line 9. Also we can compute 2^{s-1} matrices $B^{(\ell)}$ at lines 10 and 11 in parallel. At line 14, we can find the minimum in $O(s)$ time with $O(2^s/s)$ processors.

Turning our attention to the cruising phase of Algorithm 2' (or 2), we can find the minimum at line 8 in $O(\log n)$ time with $O(n/(\ell \log n))$ processors. The rest is absorbed in these complexities. Now we summarize the complexities for the parallel algorithm. T is for time and P is for the number of processors.

Accelerating phase	$T = O(R \log n), P = O(n^\omega \cdot 2^R)$
Cruising phase	$T = O(\log(n/2^R) \cdot \log n), P = O(n^3/(2^R \log n))$

If we let $2^R = n^{(3-\omega)/2}/\sqrt{\log n}$, we have the overall complexity as follows:

$$T = O(\log^2 n), P = n^{(3+\omega)/2}/\sqrt{\log n}.$$

If we have a graph with edges costs up to M we can replace n by Mn in the above complexities.

5 Parallelization for graphs with general costs

If edges costs are non-negative real numbers, we can not apply techniques in the previous sections. Even in the previous section, if M, the magnitude of edge costs, is $O(n)$, the efficiencies of both sequential and parallel algorithms get much worse than primitive methods.

Fredman [5] first gave an algorithm for the APSD problem in $o(n^3)$, that is, $O(n^3 (\log\log n/\log n)^{1/3})$ time, by showing that distance matrix multiplication can be solved in this complexity. Takaoka [12] improved this to $O(n^3(\log\log n / \log n)^{1/2})$ and pointed out that the APSP problem can be solved in the same complexity. This algorithm was also parallelized in [12]. The parallel version takes the repeated squaring approach. The parallel algorithm for distance matrix multiplication has complexities of $T = O(\log n)$ and $P = O(n^3(\log\log n)^{1/2} /(\log n)^{3/2})$ on an EREW-PRAM. Therefore the APSP problem can be solved with $T = O(\log^2 n)$ and $P = O(n^3(\log\log n)^{1/2}/(\log n)^{3/2})$. In this algorithm, we can keep track of witnesses easily and thus the APSP problem can be solved in the same complexities. The cost $PT = O(n^3(\log n \log\log n)^{1/2})$ is slightly above $O(n^3)$. Since then, it has been a major open problem whether there is an NC algorithm whose cost is $O(n^3)$.

In this section, we show a stronger result that there exists an NC algorithm for the APSP problem, whose cost is $o(n^3)$.

Definition 1. Let a parallel algorithm have time complexity T and use $P(t)$ processors at the t-th step. Then the cost complexity C is defined by

$$C = \sum_{t=1}^{T(n)} P(t).$$

A time interval I_i over which the number of processors is fixed is called a processor phase. That is, $P(t)$ are equal to P_i for all $t \in I_i$ and interval $[0..T]$ is divided as $[0..T] = I_0 \cup \cdots \cup I_{k-1}$ where k is the number of processor phases. Then we have

$$C = \sum_{i=0}^{k-1} P_i T_i,$$

where T_i is the size of interval I_i.

Brent's theorem[7] states that other processor phases can be simulated by a smaller number of processors at the expense of increasing computing time, without mentioning the overhead time for rescheduling processors. We suggest that the number of processor phases be finite so that the rescheduling does not cause too much overhead time. In the following parallel algorithm, the number of processor phases is two.

The engine, so to speak, in the acceleration phase in Algorithm 2 was a fast algorithm for Boolean matrix multiplication. We use the fast distance matrix multiplication algorithm in [12] as the engine and modify the cruising phase slightly to fit our parallel algorithm. In Algorithm 2, there is no difference between distances and lengths of paths since the edge costs are ones. In line 5 of Algorithm 2, we choose set S_i based on the distances $d_{ij}^{(\ell)}$ $(j = 0, \cdots, n-1)$ satisfying $\lceil \ell/2 \rceil \leq d_{ij}^{(\ell)} \leq \ell$ to guarantee the correct computation of distances between ℓ and $\lceil 3\ell/2 \rceil$. We observe that the computation of S_i is essentially based on path lengths, not distances. If we keep track of path lengths, therefore, we can adapt Algorithm 2 to our problem. The definition of $d_{ij}^{(\ell)}$ here is that it gives the cost of the shortest path whose length is not greater than ℓ. The algorithm follows with a new data structure, array $Q^{(\ell)}$, such that $q_{ij}^{(\ell)}$ is the length of a path that gives $d_{ij}^{(\ell)}$.

Algorithm 4

{Accelerating phase}

1 $\ell := 1$; $D^{(1)} := D$; $Q^{(1)} := \{q_{ij}^{(1)}\}$, where $q_{ij}^{(1)} = \begin{cases} 0, & \text{if } i = j \\ 1, & \text{if } i \neq j \text{ and } (i,j) \in E \\ \infty, & \text{otherwise;} \end{cases}$

2 **for** $s := 1$ **to** $\lceil \log_2 r \rceil$ **do begin**

3 $D^{(2\ell)} := D^{(\ell)} \times D^{(\ell)}$; {distance matrix multiplication in [12]}

4 $Q^{(2\ell)} := \{q_{ij}^{(2\ell)}\}$ where $q_{ij}^{(2\ell)} = \begin{cases} q_{ik}^{(\ell)} + q_{kj}^{(\ell)}, & \text{if } d_{ij}^{(2\ell)} \text{ is updated by} \\ & d_{ik}^{(\ell)} + d_{kj}^{(\ell)} \\ q_{ij}^{(\ell)}, & \text{otherwise;} \end{cases}$

5 $\ell := 2\ell$
6 **end**;
{Cruising phase}
7 **for** $s := 1$ **to** $\lceil \log_{3/2} n/r \rceil$ **do begin**
8 **for** $i := 0$ **to** $n-1$ **do**
9 Scan the i-th row of $Q^{(\ell)}$ and find the smallest set of equal $q_{ij}^{(\ell)}$'s such that $\lceil \ell/2 \rceil \leq q_{ij}^{(\ell)} \leq \ell$ and let the set of corresponding indices j be S_i;
10 $\ell_1 := \lceil 3\ell/2 \rceil$;
11 **for** $i := 0$ **to** $n-1$ **do for** $j := 0$ **to** $n-1$ **do begin**
12 **if** $S_i \neq \emptyset$ **then begin**
13 $m_{ij} := \min_{k \in S_i} \{d_{ik}^{(\ell)} + d_{kj}^{(\ell)}\}$;
14 $k :=$ one that gives the above minimum and satisfies that $q_{ik}^{(\ell)} + q_{kj}^{(\ell)}$ is minimum among such k;
15 $L := q_{ik}^{(\ell)} + q_{kj}^{(\ell)}$;
16 **end else** $\{S_i = \emptyset\}$ $L := \infty$;
17 **if** $m_{ij} < d_{ij}^{(\ell)}$ **then begin** $d_{ij}^{(\ell_1)} := m_{ij}$; $q_{ij}^{(\ell_1)} := L$ **end**
18 **else begin** $d_{ij}^{(\ell_1)} := d_{ij}^{(\ell)}$; $q_{ij}^{(\ell_1)} := q_{ij}^{(\ell)}$ **end**;
19 $\ell := \ell_1$
20 **end**
21 **end**.

As described in [12], we can parallelize the distance matrix multiplication at line 3 on an EREW-PRAM. We index time T and the number of processors P in the accelerating phase and cruising phase by 1 and 2. Then we have

$$T_1 = O(\log r \log n), P_1 = O(n^3 (\log \log n)^{1/2} / (\log n)^{3/2}).$$

The computation of all S_i can be done in $O(\log n)$ time with $O(n^2)$ processors. The dominant complexity in the cruising phase is at line 13. This part can be computed in $O(\log(n/r))$ time with $O((n/r)/\log(n/r))$ processors. Thus we have

$$T_2 = O(\log n \log(n/r)), P_2 = O(n^2 (n/r)/\log(n/r)).$$

Letting $r = (\log n / \log \log n)^{3/2}$ yields

$$\begin{aligned} T_1 &= O(\log n \log \log n), & P_1 &= O(n^3 (\log \log n)^{1/2} / (\log n)^{3/2}) \\ T_2 &= O(\log^2 n), & P_2 &= O(n^3 (\log \log n)^{3/2} / (\log n)^{5/2}). \end{aligned}$$

Thus the cost is given by

$$\begin{aligned} P_1 T_1 + P_2 T_2 &= O(n^3 (\log \log n)^{3/2} / (\log n)^{1/2}) + O(n^3 (\log \log n)^{3/2} / (\log n)^{1/2}) \\ &= O(n^3 (\log \log n)^{3/2} / (\log n)^{1/2}) \\ &= o(n^3). \end{aligned}$$

We note that we can solve the APSP problem with the same order of cost by this algorithm. We only need to keep track of witnesses at distance matrix multiplication and the minimum operation at line 13.

If we perform the accelerating phase with P_2 processors, the time for this phase will become $O(\log^2 n)$, and the cost will be the same as above for the whole computation. That is, we can keep the number of processors uniform and claim that the algorithm has the above complexities under the traditional definition of cost by $C = PT$.

6 An algorithm for graphs with small edge costs

When the edge costs are bounded by a positive integer M, we can do better than we saw in previous sections. We briefly review Romani's algorithm [9] for distance matrix multiplication.

Let A and B be distance matrices whose elements are bounded by M or infinite. Let the diagonal elements be 0. Then we convert A and B into A' and B' where

$$a'_{ij} = \begin{cases} n^{M-a_{ij}}, & \text{if } a_{ij} \neq \infty \\ 0, & \text{if } a_{ij} = \infty \end{cases}$$
$$b'_{ij} = \begin{cases} n^{M-b_{ij}}, & \text{if } b_{ij} \neq \infty \\ 0, & \text{if } b_{ij} = \infty. \end{cases}$$

Let $C' = A'B'$ be the product by ordinary matrix multiplication and $C = A \times B$ be that by distance matrix multiplication. Then we have

$$c'_{ij} = \sum_{k=0}^{n-1} n^{2M-(a_{ik}+b_{kj})}, \ c_{ij} = 2M - \lfloor \log_n c'_{ij} \rfloor.$$

Thus we can compute C with $O(n^\omega)$ arithmetic operations on integers up to n^M. Since these values can be expressed by $O(M \log n)$ bits and Schönhage and Strassen's algorithm [11] for multiplying k-bit numbers takes $O(k \log k \log\log k)$ bit operations, we can compute C in $O(n^\omega M \log n \log(M \log n) \log\log(M \log n))$ time.

We replace the accelerating phase of Algorithm 4 by the following and call the resulting algorithm Algorithm 5.

Algorithm 5

{Another accelerating phase}

1 $\ell := 1$; $D^{(1)} := D$; $Q^{(1)} := \{q_{ij}^{(1)}\}$; where $q_{ij}^{(1)} = \begin{cases} 0, & \text{if } i = j \\ 1, & \text{if } i \neq j \text{ and } (i,j) \in E \\ \infty, & \text{otherwise} \end{cases}$

2 **for** $s := 1$ **to** r **do begin**

3 $\quad D^{(\ell+1)} := D^{(\ell)} \times D$; {distance matrix multiplication by Romani in [7]}

4 $\quad Q^{(\ell+1)} := \{q_{ij}^{(\ell+1)}\}$; where $q_{ij}^{(\ell+1)} = \begin{cases} q_{ij}^{(\ell)} + 1, & \text{if } d_{ij}^{(\ell+1)} \text{ is updated} \\ q_{ij}^{(\ell)}, & \text{otherwise} \end{cases}$

5 $\quad \ell := \ell + 1$

6 **end**;

{Cruising phase} same as that in Algorithm 4.

Note that the bound M is replaced by ℓM in the distance matrix multiplication. The time for the accelerating phase is given by

$$O(n^{\omega} r^2 M \log n \log(rM \log n) \log\log(rM \log n)).$$

We assume that M is $O(n^k)$ for some constant k. Balancing this complexity with that of cruising phase, $O(n^3/r)$, yields the total computing time of

$$O(n^{(6+\omega)/3}(M \log n \log(nM \log n) \log\log(nM \log n))^{1/3})$$

with the choice of

$$r = O(n^{(3-\omega)/3}(M \log n \log(nM \log n) \log\log(nM \log n))^{-1/3}).$$

This complexity is simplified into

$$O(M^{1/3} n^{(6+\omega)/3} (\log n)^{2/3} (\log\log n)^{1/3}).$$

The value of M can be almost $O(n^{0.624})$ to keep the complexity within sub-cubic. This bound on M is a considerable improvement over $O(n^{0.116})$ given in [1].

In the above we solved only the APSD problem. In the Romani algorithm, we can not keep track of witnesses. If we could, we would be able to replace the accelerating phase by that based on repeated squaring and would have a better complexity for both the APSD and APSP problem with small edge costs.

7 Concluding remarks

The balancing parameters between the accelerating and cruising phases change depending on what engine we use in the accelerating phase. We may find more results if we use other algorithms for the engine in the accelerating phase.

Acknowledgment.

A comment on Definition 1 by Zhi-Zhong Chen is greatly appreciated.

References

1. N. Alon, Z. Galil and O. Margalit, On the exponent of the all pairs shortest path problem, Proc. 32th IEEE FOCS (1991), pp 569–575.
2. N. Alon, Z. Galil, and O. Margalit and M. Naor, Witnesses for Boolean matrix multiplication and for shortest paths, Proc. 33th IEEE FOCS (1992), pp. 417–426.
3. D. Coppersmith and S. Winograd, Matrix multiplication via arithmetic progressions, Journal of Symbolic Computation 9 (1990), pp. 251–280.

4. E. Dekel, D. Nassimi and S. Sahni, Parallel matix and graph algorithms, SIAM Jour. on Comp. 10 (1981), pp. 657–675.
5. M. L. Fredman, New bounds on the complexity of the shortest path problem, SIAM Jour. Comput. 5 (1976), pp. 49–60.
6. H. Gazit and G. Miller, An improved parallel algorithm that computes the bfs numbering of a directed graph, Info. Proc. Lett. 28 (1988) pp. 61–65.
7. A. Gibbons and W. Rytter, Efficient Parallel Algorithms, Cambridge Univ. Press (1988).
8. Y. Han, V. Pan and J. Reif, Efficient parallel algorithms for computing all pairs shortest paths in directed graphs, Proc. 4th ACM SPAA (1992), pp. 353–362.
9. F. Romani, Shortest-path problem is not harder than matrix multiplications, Info. Proc. Lett. 11 (1980) pp.134–136.
10. R. Seidel, On the all-pairs-shortest-path problem, Proc. 24th ACM STOC (1990), pp. 213–223.
11. A. Schönhage and V. Strassen, Schnelle Multiplikation Großer Zahlen, Computing 7 (1971) pp. 281–292.
12. T. Takaoka, A new upperbound on the complexity of the all pairs shortest path problem, Info. Proc. Lett. 43 (1992), pp. 195–199.
13. T. Takaoka, An efficient parallel algorithm for the all pairs shortest path problem, WG 88, Lecture Notes in Computer Science 344 (Springer, Berlin, 1988) pp. 276–287.

This article was processed using the LaTeX macro package with LLNCS style