An $O(n^3(\log\log n/\log n)^{5/4})$ Time Algorithm for All Pairs Shortest Path[1]

Yijie Han

School of Computing and Engineering
University of Missouri at Kansas City
5100 Rockhill Road
Kansas City, MO 64110
hanyij@umkc.edu

Abstract

We present an $O(n^3(\log\log n/\log n)^{5/4})$ time algorithm for all pairs shortest paths. This algorithm improves on the best previous result of $O(n^3/\log n)$ time.

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 [8], where m is the number of edges of the graph. Recently Pettie showed [12] an algorithm with time complexity of $O(mn + n^2\log\log n)$. See [13] for recent development. There are also results for all pairs shortest paths for graphs with integer weights[9, 14, 15, 18, 19, 20]. 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 bounds for all pairs shortest paths to $O(n^3(\log\log n/\log n)^{1/2})$ [16]. 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. In 2004 we obtained an algorithm with time complexity $O(n^3(\log\log n/\log n)^{5/7})$

[1]Research supported in part by NSF grant 0310245.

[10]. Very recently Takaoka obtained an algorithm with time $O(n^3 \log\log n/\log n)$ [17] and Zwick gave an algorithm with time $O(n^3\sqrt{\log\log n}/\log n)$ [21]. Chan gave an algorithm with time complexity of $O(n^3/\log n)$ [5]. Chan's algorithm does not use tabulation and bit-wise parallelism. His algorithm also runs on a pointer machine. We were unaware of Chan's result and published [11] in which we achieved $O(n^3/\log n)$ time using large word size. Since Chan published [5] before us the result of $O(n^3/\log n)$ time should be fully attributed to Chan.

Takaoka raise the question [17] whether $O(n^3/\log n)$ can be achieved. It is thought that $O(n^3/\log n)$ is a natural bound for this problem [5]. We show, however, that this bound can be improved to $O(n^3(\log\log n/\log n)^{5/4})$. Our technique is the traditional table lookup technique. However, we construct many tables and maximize the saving the table lookup could bring to us. We shall give reasons why the results presented in this paper would be difficult to improve on.

2 The Approach

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.

Let $p = c_1(\log n/\log\log n)^{1/2}$ and $q = c_2(\log n/\log\log n)^{1/4}$, where $0 < c_1, c_2 \leq 1$ are suitable constants to be chosen later.

We divide the first matrix A into $n/p \cdot n/q$ small matrices each has dimension $p \times q$. We divide the second matrix B into $n/q \cdot n/p$ small matrices each has dimension $q \times p$.

For a $p \times q$ matrix $E = (e_{ij})$, we obtain the ranks for $e_{ir} - e_{is}$ for all $1 \leq r < s \leq q$. Here rank is defined in [17] and will be given explicitly in the next section. These ranks are $O(\log\log n)$ bits numbers. We therefore obtain a matrix E' of ranks. E' has dimension $e \times f$ where $e = p$ and $f = O(q^2)$. E' has $c_3 \log n$ bits with $c_3 < 1/2$ being a constant. The corresponding $q \times p$ matrix F is processed similarly. Now EF can be computed in one step by

a table lookup. Since direct or naive computation of EF needs $O((\log n/\log\log n)^{5/4})$ time we saved a factor of $(\log n/\log\log n)^{5/4}$ in the time complexity. However, the result R has $c_1^2 \log n/\log\log n$ numbers (indices) each having $O(\log\log n)$ bits. We cannot disassemble R immediately for otherwise we lose a factor of $\log n/\log\log n$ in the time complexity.

What we do is to combine the results for $\log n/\log\log n$ small matrix product into one. This should take $O(\log n/\log\log n)$ time by repeated table lookup (details in the later section). After that we disassemble R and get the $c_1^2 \log n/\log\log n$ resulting indices.

This concludes the strategy of our approach and we give our goal as:

Goal: Compute the distance produce of a $p \times q$ matrix and a $q \times p$ matrix in constant time and compute the distance product of a $p \times q\log n/\log\log n$ matrix and a $q\log n/\log\log n \times p$ matrix in $O(\log n/\log\log n)$ time, after processing some lookup tables.

3 Obtaining the Ranks

Let $l = (\log n/\log\log n)^{5/4}$ and $m = \log^4 n$.

We first divide the first matrix A into $n/m \cdot n/l$ medium matrices each of dimension $m \times l$ and second matrix B into $n/l \cdot n/m$ medium matrices each of dimension $l \times m$. We will further divide each medium matrix into small matrices in the next section. The reason we need these medium matrices is that we need to compute the ranks to be defined below.

Let A_1 and B_1 be two medium matrices. We want to compute $C_1 = A_1B_1$, where $A_1 = (a_{ij})$, $B_1 = (b_{ij})$ and $C_1 = (c_{ij})$. We have that $c_{ij} = \min_k\{a_{ik} + b_{kj}\}$. Fredman noticed [7] that $a_{ir} + b_{rj} \leq a_{is} + b_{sj} \Longleftrightarrow a_{ir} - a_{is} \leq b_{sj} - b_{rj}$. Takaoka[16] showed that by sorting $\{a_{ir} - a_{is}\}_{i=1,2,...,m}$ into sorted list E_{rs} and sorting $\{b_{sj} - b_{rj}\}_{j=1,2,...m}$ into sorted list F_{rs} and then merging E_{rs} and F_{rs} we obtain a sorted list G_{rs}. Let H_{rs} be the list of ranks of $a_{ir} - a_{is}$ $(i = 1, 2, ..., m)$ in G_{rs} and let L_{rs} be the list of ranks of $b_{sj} - b_{rj}$ $(j = 1, 2, ...m)$ in G_{rs}. Let $H_{rs}[i]$ and $L_{rs}[j]$ be the i-th and the j-th component of H_{rs} and L_{rs}, respectively. Then:

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

The lists H_{rs} and L_{rs} for $1 \leq r < s \leq l$ can be made in $O(l^2 m)$ time, when the sorted lists are available. The following fact was used in [16, 17].

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

Note that each $H_{rs}[i]$ and $L_{rs}[j]$ has $4 \log\log n + 1$ bits.

Here we calculate the time for sorting and for merging. Sorting for each pair of rs for each medium matrix takes $O(m \log m)$ time. For all pairs of rs this is $O(ml^2 \log m)$ time. For the input matrix A this takes $O((n/m)(n/l)ml^2 \log m) = O(n^2 l \log m) = O(n^2(\log n/\log\log n)^{5/4} \log\log n)$ time. The same time holds for matrix B. The merging takes $O((n/m)^2(n/l)l^2 m) = O(n^3 l/m)$ $= O(n^3(\log n/\log\log n)^{5/4}/\log^4 n) = O(n^3(\log\log n/\log n)^{5/4})$ time.

The purpose of this section is to obtain all the ranks $H_{rs}[i]$'s and $L_{rs}[j]$'s.

4 Computing the Small Matrix Product

We further divide each medium matrix into small matrices. Divide the first medium matrix A_1 into small matrices each of dimension $p \times q$ and divide the second medium matrix B_1 into small matrices each of dimension $q \times p$. Let $E = (e_{ij})$ be a small matrix from A_1 and let $F = (f_{ij})$ be a small matrix from B_1. We are to compute $D = EF$.

Thus if we have $H_{rs}[i]$ and $L_{rs}[j]$ for $1 \leq r < s \leq q$ we can determine the index k such that $h_{ij} = e_{ik} + f_{kj}$. Since $q = c_2(\log n/\log\log n)^{1/4}$ we have $O((\log n/\log\log n)^{1/2})$ rs pairs. We form matrix E' and F':

$$E' = \begin{bmatrix} H_{11}[1] & H_{12}[1] & \ldots & H_{23}[1] & \ldots & H_{q-1,q}[1] \\ H_{11}[2] & H_{12}[2] & \ldots & H_{23}[2] & \ldots & H_{q-1,q}[2] \\ \ldots & \ldots & \ldots & \ldots & \ldots & \ldots \\ H_{11}[p] & H_{12}[p] & \ldots & H_{23}[p] & \ldots & H_{q-1,q}[p] \end{bmatrix}$$

$$F' = \begin{bmatrix} L_{11}[1] & L_{12}[1] & \ldots & L_{23}[1] & \ldots & L_{q-1,q}[1] \\ L_{11}[2] & L_{12}[2] & \ldots & L_{23}[2] & \ldots & L_{q-1,q}[2] \\ \ldots & \ldots & \ldots & \ldots & \ldots & \ldots \\ L_{11}[p] & L_{12}[p] & \ldots & L_{23}[p] & \ldots & L_{q-1,q}[p] \end{bmatrix}$$

Thus E' and F' have $c_3 \log n/\log\log n$ numbers each having $4 \log\log n + 1$ bits. That is

E' and F' have $c_4 \log n$ bits. Here c_3 and c_4 are constants which can be made small. Now E' and F' can be used to index into a lookup table T_1 to get the $p^2 = c_1^2 \log n / \log\log n$ indices for the matrix multiplication of EF. These p^2 indices are stored in one word. The lookup table T_1 can be built in $O(n)$ time. Thus small matrix multiplication takes constant time.

5 Combining Results

Each small matrix multiplication returns a word R of $c_5 \log n$ bits. R can be viewed as an $p \times p$ matrix $R[1..p, 1..p]$ with each element taking $(1/4)(\log\log n - \log\log\log n) + \log c_2$ bits to indicate the winner's index. Suppose we have two small matrix multiplication result R_1 and R_2. Let $R_1[i,j] = r_{ij}$ and $R_2[i,j] = s_{ij}$, if we can obtain R_3 with $R_3[i,j] = (H_{r_{ij}s_{ij}}[i], r_{ij}, s_{ij})$, then our computation can continue. Note that the small matrix multiplications return a three dimensional array R_{xyz}, $0 \le x, z < n/p$, $0 \le y < n/q$, of words mentioned above.

We now use R_{u1v} and R_{u2v}, for a fixed u and $0 \le v < m/p$, to represent the results of matrix multiplication of the two (consecutive) small matrices in A_1 with a row of (a total of m/p) small matrices in B_1. Here consecutive means we are pairing even y indexed R_{xyz} with next odd $y+1$ indexed $R_{x(y+1)z}$. Note that u is fixed while v takes various values and therefore we are considering many pairs of R_{xyz} simultaneously. We first group R_{u1v} and R_{u2v} into one matrix R_{uv} by a table lookup such that $R_{u1v}[i,j]$ and $R_{u2v}[i,j]$ are consecutively placed in $R_{uv}[i,j]$. Therefore $R_{uv}[i,j]$ has $2\log q = (1/2)(\log\log n - \log\log\log n) + 2\log c_2$ bits. We may assume that the address (that is (u,v,i,j)) is stored together with each original number $R_{uv}[i,j]$. Since each address is a $9\log\log n$ bit number this should not create any problem. We shall call (u,v,i,j) the address of $R_{uv}[i,j]$ and the original number in $R_{uv}[i,j]$ the winning index of $R_{uv}[i,j]$.

We build a table for each row of a medium matrix in A and each medium matrix of a row of medium matrices in B, i.e. fixing A_1 and B_1 and for each row in A_1 build a table. That is we need $(n^2/\log^{5/4} n)(n/\log^4 n) = n^3/\log^{21/4} n$ tables. Each of such a table has size

$O(\log^{2*5/4} n) = O(\log^{5/2} n)$ because $R_{uv}[i,j]$ has 2 values each can be as large as $\log^{5/4} n$. For each of $R_{uv}[i,j]$ value we can index into a table to find the H_{rs} value.

We first sort the $R_{uv}[i,j]$'s within each R_{uv} by their $(i, \text{winning index})$ key and move the addresses with the winning indices. This is done by a table lookup (not the table built above) and therefore the cost is constant time for each fixed R_{uv}. Since there are $p = c_1(\log n/\log\log n)^{1/2}$ winning indices with the same i address and each winning index has $2\log q = (1/2)(\log\log n - \log\log\log n) + 2\log c_2$ bits, by a suitable choice of c_1 and c_2 so that $2^{2\log q} \leq p$ and therefore there are more winning indices than the different values of winning indices. Note that we can put all these possible winning indices into one word. Within each R_{uv} for each value of winning indices we just need to keep one copy. We use a table lookup to obtain the H_{rs} value separately for each different winning index value in R_{uv}. Since there are q^2 different winning index values we need to take $O(q^2)$ time. This is for a fixed i in $R_{uv}[i,j]$. For all R_{uv}'s for a fixed u we need $O(pq^2)$ time. After that we assemble all these separately obtained H_{rs} values into one word w (as we have mentioned that all these values fit into one word). This is done for all R_{uv}'s and there are m/p of them (because for differnet v's the table can be reused). We make copies of w and concatenate one copy of w with each R_{uv}. That is the way each R_{uv} obtains its H_{rs} values. Note that we do not do sorting other that the sorting within each word by table lookup mentioned above.

The similar process is done on the second input matrix B. If $R_1[i,j] = r_{ij}$ and $R_2[i,j] = s_{ij}$, then we will obtain R_4 with $R_4[i,j] = (L_{r_{ij}s_{ij}}[j], r_{ij}, s_{ij})$.

Now use R_3 and R_4 to index into yet another table T_4. T_4 will give result R_5 with $R_5[i,j]$ being the winner between r_{ij} and s_{ij} for $1 \leq i,j \leq p$. R_3 and R_4 have $c_5 \log n$ bits and therefore table T_4 can be built in $O(n)$ time. This accomplishes the combining of R_1 and R_2 into R_5. This is the 0-th step of combining. Each step of the combining pairs-off every 2 small matrices.

In the t-th step, we consider every 2^{t+1} R_{xyz}'s (for fixed x and z and consecutive 2^{t+1} y's). By the previous steps now the winning indices of these R_{xyz}'s are in R_{uwv} (fixed u, w

and v). Now $0 \leq u, v < n/p$, $0 \leq w < n/(q2^{t+1})$. Here again R_{uwv} contains $R_{uw(1)v}$ and $R_{uw(2)v}$ and we are going to combine the winning indices into one. We replace each duplicate winning index in $R_{uwv}[i, *]$ with a dummy (can be set to max) but keep the first winning index among the same duplicated winning indices. Let the resulting word be R^1_{uwv}. We use each distinct winning index value of $R_{uwv}[i.j]$ and index into the lookup table to obtain the H_{rs} values. There are $2^{2(t+1)}q^2$ distinct winning indices for $R_{uwv}[i, j]$ because $R_{uwv}[i, j]$ consists of $R_{uw(1)v}[i, j]$ and $R_{uw(2)v}[i, j]$ and each of them can take $2^{t+1}q$ different values. Since table can be reused for $0 \leq v < m/p$, looking these indices values up in the tables for fixed u, w, i in $R_{uwv}[i, j]$ takes $O(2^{2(t+1)}q^2)$ time. For all i's in u this takes $O(2^{2(t+1)}pq^2)$ time. But this is done once for all R_{uwv}'s (for all i in u, fixed w and $0 \leq v < m/p$ since table can be reused for each fixed i, u, w and different ranges of v). We have that $2^{t+1} \leq O(\log^{5/4} n)$ (because we need not combine more than $O(\log^{5/4} n)$ words into one word). The number of R_{uwv}'s (for fixed u, w, i and $0 \leq v < m/p$) is $\Omega(m/p) = \Omega(2^{2(t+1)} \log n \log\log n)$. Thus we can spend $O(2^{2(t+1)} \log n \log\log n)$ time and still maintain linear time in each step. The actual time we need as mentioned above is $O(2^{2(t+1)}pq^2) = O(2^{2(t+1)}(\log n/ \log\log n))$. Thus we can iterate this part of this step at least $\log\log n$ times (remember there are $O(\log\log n)$ steps) and still keep linear time.

In the following we will use bitonic sorting. We put the fact about bitonic sorting in the following Lemma.

Lemma [3, 4]: n data items can be sorting on the n-node hypercube network in $O(\log^2 n)$ time. The serialized version of it can sort n data items in $O(n \log^2 n)$ time.

We put these H_{rs} values together with its winning indices into $2^{2(t+1)}$ words w_k, $0 \leq k < 2^{2(t+1)}$ to contain H_{rs} values for the $2^{2(t+1)}q^2$ distince winning indices(for fixed i, w and u). We then sort R^1_{uwv}, for fixed u, w and $k2^{2(t+1)} \leq v < (k+1)2^{2(t+1)}$, together with w_k, $0 \leq k < 2^{2(t+1)}$, by the packed winning indices in them, for $k = 0, 1, ..., m/(p2^{2(t+1)})$. We can do this by a serialized version of the bitonic sort[3, 4]. The bitonic sort will bring a factor of $O(t^2)$ to the complexity (note that we are sorting $O(2^{2(t+1)})$ words, the winning indices

within each word can be sorted by a table lookup.). Since in the t-th step we have reduced the data amount by a factor of 2^t (the range of y has shrunk) the factor of $O(t^2)$ in the sorting can be absorbed. Therefore we can achieve linear time for sorting.

After sorting we now copy H_{rs} value to attach them to winning indices. For each winning index there are at most $2^{2(t+1)}$ copies since duplicates have been removed. Therefore copying can be done by incurring a factor of $O(t)$ in the time complexity and since data amount have be reduced this extra factor can be absorbed.

Now use bitonic sort to sort on the addresses to bring the H_{rs} values to each R_{uv}.

The combining (pairing-off) of each small matrix thus takes constant time. Therefore after $O(\log\log n)$ steps we have combined $\log n/\log\log n$ small matrix multiplication result into one resulting matrix R_6. R_6 can be viewed as an $c(\log n/\log\log n)^{1/2} \times c(\log n/\log\log n)^{1/2}$ matrix with $R_6[i,j]$ giving the index k for $c_{ij} = a_{ik} + b_{kj}$ (mentioned in the previous section).

6 The Result

Since we expend $O(\log n/\log\log n)$ time multiplying a $p \times l$ matrix with an $l \times p$ matrix where $p = c(\log n/\log\log n)^{1/2}$ and $l = (\log n/\log\log n)^{5/4}$, and direct or naive matrix multiplication takes $O((\log n/\log\log n)^{9/4})$ time we save a factor of $(\log n/\log\log n)^{5/4}$. By choosing small enough constant c_i's all our preprocessing and table built up can be done in $O(n^3(\log\log n/\log n)^{5/4})$ time. Thus our matrix multiplication algorithm takes $O(n^3(\log\log n/\log n)^{5/4})$ time.

Theorem: All-pairs shortest paths can be computed in $O(n^3(\log\log n/\log n)^{5/4})$ time.

We give reasons why the results presented in this paper would be difficult to improve on. When using tabulation and bit-parallelism we can expect to save a factor of roughly $\log n$ because $\log n$ bits are encoded in one word. However, in all pairs shortest paths computation if we encode $\log n$ numbers into a word then we can compute $\log^2 n$ results in constant time by table lookup. That is we can speed up algorithm by a factor of $\log^2 n$. However, the $\log^2 n$

results have to be stored in at least $\log n$ words and therefore takes $\log n$ time to access. Thus the speedup factor is reduced to $\log n$. Since we need $\log\log n$ bits to encode a number, with $\log n$ bits we can use one word to store $\log n/\log\log n$ results. Thus the best strategy is to encode a $(\log n/\log\log n)^{1/2} \times a$ matrix and an $a \times (\log n/\log\log n)^{1/2}$ matrix into a word. Here the larger the a, the better. However, we can encode at most $\log n/\log\log n$ numbers into one word because each number takes $\log\log n$ bits, plus if we use Fredman-Takaoka approach[7, 16] then the encoding will blow a numbers to a^2 numbers and thus we have that $a^2(\log n/\log\log n)^{1/2} = \log n/\log\log n$. Therefore $a = (\log n/\log\log n)^{1/4}$. This is the dimension of the small matrices we used earlier in the paper.

Acknowledgment

The author wishes to thank a referees for pointing out a serious error in the manuscript.

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. 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).

[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. Improved algorithms for all pairs shortest paths. *Information Processing Letters*, 91, 245-250(2004).

[11] Y. Han. Achieving $O(n^3/\log n)$ time for all pairs shortest paths by using a smaller table. *Proc. 21st Int. Conf. on Computers and Their Applications (CATA-2006)*, Seattle, Washington, 36-37(2006).

[12] 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).

[13] S. Pettie, V. Ramachandran. A shortest path algorithm for real-weighted undirected graphs. *SIAM J. Comput.*, Vol. 34, No. 6, 1398-1431(2005).

[14] 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).

[15] R. Seidel. On the all-pairs-shortest-path problem in unweighted undirected graphs. *J. Comput. Syst. Sci.*, 51, 400-403(1995).

[16] T. Takaoka. A new upper bound on the complexity of the all pairs shortest path problem. *Information Processing Letters* **43**, 195-199(1992).

[17] 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).

[18] M. Thorup. Undirected single source shortest paths with positive integer weights in linear time. *Journal of ACM*, 46(3), 362-394(1999).

[19] 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.

[20] 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.

[21] 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).