All-pairs disjoint paths from a common ancestor in $\tilde{O}(n^\omega)$ time

Raphael Yuster *

Abstract

A *common ancestor* of two vertices u, v in a directed acyclic graph is a vertex w that can reach both. A $\{u, v\}$-*junction* is a common ancestor w so that there are two paths, one from w to u and the other from w to v, that are internally vertex-disjoint. A *lowest common ancestor* (LCA) of u and v is a common ancestor w so that no other common ancestor of u and v is reachable from w. Every $\{u, v\}$-LCA is a $\{u, v\}$-junction, but the converse is not true. Similarly, not every common ancestor is a junction.

The *all-pairs common ancestor* (APCA) problem computes (or determines the non-existence of) a common ancestor for all pairs of vertices. Similarly defined are the *all-pairs junction* (APJ) and the *all-pairs LCA* (APLCA) problems. The APCA problem also has an *existence* version.

Bender et al. obtained an algorithm for APCA existence by reduction to transitive closure. Their algorithm runs in $\tilde{O}(n^\omega)$ time where $\omega < 2.376$ is the exponent of fast Boolean matrix multiplication and n is the number of vertices. Kowaluk and Lingas obtained an algorithm for APLCA whose running time is $O(n^{2+1/(4-\omega)}) \leq o(n^{2.616})$. Our main result is an $\tilde{O}(n^\omega)$ time algorithm for APJ. Thus, junctions for all pairs can also be computed in essentially the time needed for transitive closure.

For a subset of vertices S, a common ancestor of S is a vertex that can reach each vertex of S. A *lowest* common ancestor of S is a common ancestor w of S so that no other common ancestor of S is reachable from w. For $k \geq 2$, the k-APCA and the k-APLCA problems are to find, respectively, a common ancestor and a lowest common ancestor for each k-set of vertices. We prove that for all fixed $k \geq 8$, the k-APCA problem can be solved in $\tilde{O}(n^k)$ time, thereby obtaining an essentially optimal algorithm. We also prove that for all $k \geq 4$, the k-APLCA problem can be solved in $\tilde{O}(n^{k+1/2})$ time.
Keywords: algorithm, directed acyclic graph, common ancestor, fast matrix multiplication

1 Introduction

The problem of finding common ancestors in a tree and, more generally, in a directed acyclic graph (dag) is a basic, extensively-studied algorithmic problem that has numerous applications, ranging from analysis of genealogical data to object inheritance in programming languages (see [3, 15, 10]). A *common ancestor* of a pair of distinct vertices u, v in a dag is a vertex w so that both u and v are descendants of w. A *lowest common ancestor* (LCA) of vertices u, v is a common ancestor of u, v that has no proper descendant that is a common ancestor of u, v. For example, vertex c in Figure 1 is a $\{u, v\}$-LCA while vertices a and b are common ancestors of u, v that are not $\{u, v\}$-LCA's. One important useful feature of an LCA is that there are two paths starting from the LCA, one to u and the other to v, that are internally vertex-disjoint. However, a common ancestor need not be an LCA to possess this property. A common ancestor w of u, v is called a $\{u, v\}$-*junction* if there are two paths, one from w to u and the other from w to v, that

*Department of Mathematics, University of Haifa, Haifa 31905, Israel. E–mail: raphy@math.haifa.ac.il

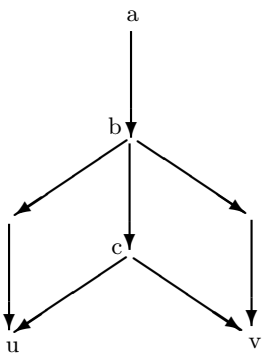

Figure 1: Types of common ancestors of a pair of vertices u, v

are internally vertex-disjoint. Clearly, a $\{u, v\}$-LCA is a $\{u, v\}$-junction. The vertex b in Figure 1 is a $\{u, v\}$-junction that is not a $\{u, v\}$-LCA. The vertex a in Figure 1 is a common ancestor of u, v that is not a $\{u, v\}$-junction. Thus, we have a 3-fold type hierarchy for common ancestors.

For a dag G, the *all-pairs common ancestor* (APCA) problem is to find (or determine the non-existence of) a common ancestor for all pairs of vertices of G. Similarly defined are the *all-pairs junction* (APJ) and the *all-pairs lowest common ancestor* (APLCA) problems. In all three problems, the output is a square symmetric matrix M whose rows and columns are indexed by the vertices, and each entry $M(u, v)$ is either 0 if u, v do not have a common ancestor, or is a common ancestor of the required type. For completeness we define $M(u, u) = u$. Clearly, the complexity of the APCA problem is at most the complexity of the APJ problem which, in turn, is at most the complexity of the APLCA problem.

Bender et al. [2] have shown that the APCA *existence* problem (namely, we only wish to know, for each pair of vertices, whether or not they have a common ancestor) can be easily reduced to the problem of computing the transitive closure of a directed graph. The latter is a classical application of Boolean square matrix multiplication, as shown by Furman [9] and Munro [14]. The fastest algorithm for Boolean matrix multiplication of two square matrices of order n runs in $O(n^\omega)$ where $\omega < 2.376$, as shown by Coppersmith and Winograd [5]. The exponent ω is sometimes called the *exponent of fast matrix multiplication*; many researchers believe that $\omega = 2 + o(1)$. It is not difficult to modify the algorithm from [2], at the price of a logarithmic factor, and obtain a witness common ancestor, if one exists. Thus, the APCA problem for dags with n vertices can be solved in $\tilde{O}(n^\omega)$ time.

On the other end of the scale, the fastest algorithm for the APLCA problem, due to Kowaluk and Lingas [12], improving an earlier algorithm of Bender et al., runs in $O(n^{2+1/(4-\omega)}) \leq o(n^{2.616})$ time. A minor improvement, using the fast *rectangular* matrix multiplication algorithm from [11] can improve the running time to $O(n^{2.575})$ [6]. In case $\omega = 2 + o(1)$ the algorithm of Kowaluk and Lingas runs in $\tilde{O}(n^{2.5})$ time. Recently, Kowaluk and Lingas constructed an $\tilde{O}(n^\omega)$ time algorithm that finds the LCA for all pair of vertices that have a *unique* LCA [13]. Eckhardt et al. showed how to compute, for all pairs of vertices, the set of all LCA's in $O(n^{3.34})$ time [7].

The main result of this paper is an algorithm for the APJ problem whose running time is $\tilde{O}(n^\omega)$. It is somewhat surprising that finding a representative junction for all pairs can be done in essentially the same time needed to find just a representative arbitrary common ancestor for all pairs. In fact, our algorithm also yields an implicit representation of the disjoint paths from the junction to its corresponding pair.

Theorem 1.1 (Main result) *Given a dag G with n vertices, an APJ matrix M of G can be computed in $\tilde{O}(n^\omega)$ time. Furthermore, given a pair of vertices u, v with $M(u,v) = w \neq 0$, then two internally disjoint paths from w to u and to v can be produced in time proportional to the sum of their lengths.*

Generalizing the notion of common ancestry, for a subset of vertices S, a common ancestor of S is a vertex that can reach each vertex of S. A *lowest* common ancestor of S is a common ancestor w of S so that no other common ancestor of S is a descendant of w. For $k \geq 2$, the k-APCA and the k-APLCA problems are to find, respectively, a common ancestor and a lowest common ancestor (if exist) for each k-set of vertices. The case $k = 2$ corresponds to the APCA and the APLCA problems. Notice that just listing the output requires $\Theta(n^k)$ time. It turns out that for all fixed $k \geq 8$, the k-APCA problem can be solved in $\tilde{O}(n^k)$ time, and hence we have an essentially tight algorithm. For the k-APLCA problem, we can prove that for all fixed $k \geq 4$, the problem can be solved in $\tilde{O}(n^{k+1/2})$ time. Here we still have a gap between the trivial lower bound and the upper bound.

Theorem 1.2 *For all fixed $k \geq 8$, the k-APCA problem can be solved in $\tilde{O}(n^k)$ time. For all fixed $k \geq 4$ the k-APLCA problem can be solved in $\tilde{O}(n^{k+1/2})$ time.*

The rest of this paper is organized as follows. Section 2 contains the algorithm proving Theorem 1.1. Algorithms for k-APLA and k-APLCA yielding Theorem 1.2 are given in Section 3. The final section contains some concluding remarks.

2 Finding junctions for all pairs of vertices

We describe an algorithm proving Theorem 1.1. Let $G = (V, E)$ be a dag with n vertices, and suppose that $V = \{1, \ldots, n\}$ is a topological ordering. Thus, $(u, v) \in E$ implies $u < v$. Let $A = A_G$ be the adjacency matrix of G with 1 in the diagonal.

The output of our algorithm consists of two matrices. The first, denoted M, is a solution to the APJ problem. Namely, $M(u, v)$ is either 0 if u, v do not have a common ancestor or else $M(u, v)$ is a $\{u, v\}$-junction. The second matrix, denoted P, serves as an implicit representation of internally disjoint paths from a junction to its corresponding pair. We define P as follows. Suppose $u < v$ and $M(u, v) = w \notin \{0, u\}$. Then, $P(u, v) = x$ where $(x, u) \in E$ and $M(x, v) = w$. Note that it is possible that $x = w$. In case $u \geq v$ or $M(u, v) \in \{0, u\}$ we leave the value of $P(u, v)$ unspecified.

Let $k = \lceil \log_2(n - 1) \rceil$. The first step of the algorithm consists of constructing several matrices. Let $A_0 = A$. For $r = 1, \ldots, k$ let $A_r = A_{r-1} \cdot A_{r-1}$ (a Boolean product) and let W_r be a corresponding matrix of witnesses. Notice also that A_k is the transitive closure matrix of G. Let $F(u, v)$ be the smallest index r so that $A_r(u, v) \neq 0$. If $A_k(u, v) = 0$ then define $F(u, v) = k + 1$. Finally, let C be an APCA matrix for G.

Since M is symmetric, it suffices to compute $M(u, v)$ and $P(u, v)$ for all pairs so that $u < v$. We will compute these values in order so that $M(u, v)$ and $P(u, v)$ are computed before $M(u', v')$ and $P(u', v')$ if and only if $u < u'$ or $u = u'$ and $v < v'$. In fact, in some cases the value of $M(u, v)$ depends on previously computed values. The algorithm computing M and P is given in Figure 2.

Lemma 2.1 *The matrices M and P are computed in $\tilde{O}(n^\omega)$ time.*

```
1. for u = 1 to n − 1 do
2.     for v = u + 1 to n do
3.         if A_k(u, v) = 1 then
4.             M(u, v) ← u
5.         else
6.             p* ← C(u, v)
7.             if p* = 0 then
8.                 M(u, v) ← 0
9.             else
10.                p ← p*
11.                while (p, u) ∉ E do
12.                    s ← F(p, u)
13.                    p ← W_s(p, u)
14.                M(u, v) ← M(p, v)
15.                P(u, v) ← p
```

Figure 2: Computing the matrices M and P.

Proof: Computing all the matrices A_r for $r = 1, \ldots, k$ requires $O(n^\omega \log n)$ time and computing all the matrices W_r requires $\tilde{O}(n^\omega \log n) = \tilde{O}(n^\omega)$ using, say, the algorithm of Alon and Naor [1]. The matrix F is a by-product of the computations of $A_1, \ldots, A_k$. As noted in the introduction, an $APCA$ matrix C can be computed in $\tilde{O}(n^\omega)$ time.

It remains to show that the *while* loop in line 11 of Figure 2 performs $O(\log n)$ iterations (thus, even if $\omega = 2 + o(1)$, the algorithm would still run in $\tilde{O}(n^\omega)$ time). Indeed, the value of s in Line 12 decreases with each iteration (see also the next lemma), while in the first iteration we already have $s \leq k = \lceil \log_2(n-1) \rceil$. ■

Lemma 2.2 *The algorithm correctly computes the matrices M and P.*

Proof: We prove the lemma by induction. Thus, we show that $M(u, v)$ and $P(u, v)$ are correctly computed assuming all previous values have been correctly computed. In the case $u = 1$, Line 3 guarantees that if 1 is an ancestor of v then $M(1, v) = 1$ and if 1 is not an ancestor of v then $p^* = 0$ in Line 6 and thus $M(1, v) = 0$ in line 8. In any case, a correct value for $M(1, v)$ is established, and $P(1, v)$ need not be assigned in this case.

Consider the general case where $1 < u < v \leq n$. If u is an ancestor of v then $A_k(u, v) = 1$ and it is correct to define $M(u, v) = u$ (and leave $P(u, v)$ unspecified) as is done in Line 4. If u and v have no common ancestor then $p^* = 0$ in Line 6 and it is correct to define $M(u, v) = 0$ (and leave $P(u, v)$ unspecified) as is done in Line 8. Otherwise, p^*, assigned in Line 6, is a common ancestor of u and v and $p^* \neq u$.

If $(p^*, u) \in E$ then the *while* loop in Line 11 is not performed. Notice that $M(p^*, v)$ has already been computed and $M(p^*, v) = p^*$ (since p^* is an ancestor of v). Any path from p^* to v is internally vertex disjoint from the one-arc path (p^*, u) from p^* to u and hence it is correct to define $M(u, v) = M(p^*, v) = p^*$ in this case and correct to define $P(u, v) = p^*$.

```
DISJOINT(x, y)
1.    w ← M(x, y)
2.    if w = x
3.        q_1 ← (w)
4.        q_2 ← GENPATH(x, y)
5.    else
6.        z ← P(x, y)
7.        [q_1, q_2] ← DISJOINT(z, y)
8.        q_1 → q_1 ∪ (z, x)
9.    return [q_1, q_2]
```

```
GENPATH(x, y)
1.    if (x, y) ∈ E
2.        q ← (x, y)
3.    else
4.        s ← F(x, y)
5.        p ← W_s(x, y)
6.        q ← GENPATH(x, p)∪
              GENPATH(p, y)
7.    return q
```

Figure 3: Computing disjoint paths from a common ancestor.

If $(p^*, u) \notin E$, then the goal of the while loop is to find vertices on a path from p^* to u getting closer and closer to u, until an incoming neighbor of u is found. Let $p_0 = p^*, p_1, p_2, \ldots$ be the sequence of values of p at the beginning of each iteration of the while loop, and let $s_1, s_2, \ldots$ be the sequence of values of s established in Line 12. We cannot have $s_1 = 0$ since $A_0(p_0, u) = 0$ (as $(p_0, u) \notin E$). Since $p_0 = p^*$ is an ancestor of u we have $A_k(p_0, u) = 1$. Thus, $1 \leq s_1 \leq k$. The minimality of s_1 guarantees that p_1 is an *internal* vertex on a path from p_0 to u. Since $A_{s_1} = A_{s_1-1}A_{s_1-1}$ we have that $A_{s_1-1}(p_1, u) = 1$. Thus, $F(p_1, u) \leq s_1 - 1$. Either $(p_1, u) \in E$ or else $s_2 < s_1$ and p_2 is an *internal* vertex on a path from p_1 to u. Since s always decreases and never reaches 0, we eventually end up with some p_z so that $(p_z, u) \in E$ and there is a path from $p^* = p_0$ to u of the form $(p^*, \ldots, p_z, u)$. Suppose $M(p_z, v) = w$. Clearly $w \neq 0$ since p_z and u have a common ancestor (e.g. p^* is one such ancestor). It may be the case that p_z is an ancestor of v. In this case we have $w = p_z$ and any path from p_z to v is internally disjoint from the one-edge path (p_z, u) (since u is not an ancestor of v). Thus, it is correct to define $M(u, v) = M(p_z, v) = p_z = w$ in this case and to define $P(u, v) = p_z = w$. If p_z is not an ancestor of v then consider two internally disjoint paths one from w to p_z denoted q_1 and the other from w to v denoted q_2. Since both p_z and u do not appear on q_2 we may extend q_1 to a path from w to u which is internally disjoint from q_2. Thus, it is correct to define $M(u, v) = M(p_z, v) = w$ and to define $P(u, v) = p_z$. ■

Lemma 2.3 *Given two vertices u, v that have a common ancestor $M(u, v) = w$, two internally disjoint paths, $q_1 = (w, \ldots, u)$ and $q_2 = (w, \ldots, v)$ so that $V(q_1) \cap V(q_2) = w$ can be produced in $O(|q_1| + |q_2|)$ time.*

Proof: We shall use two algorithms, the first $GENPATH(x, y)$ assumes that $x < y$ and that x is an ancestor of y. It returns a path q from x to y in $O(|q|)$ time. The code for $GENPATH(x, y)$, given on the r.h.s. of Figure 3 is easily verified to produce the required path in the claimed running time. The second algorithm, $DISJOINT(x, y)$ assumes that x and y have a common ancestor and that $x < y$. It returns a pair of paths q_1 and q_2 as in the statement of the lemma. The code for $DISJOINT(x, y)$ is given on the l.h.s. of Figure 3. The correctness of $DISJOINT(x, y)$ follows immediately from the definition of P. To produce the disjoint paths as in the statement of the lemma we simply call $DISJOINT(u, v)$ (assuming $u < v$). ■

3 Algorithms for k-APCA and k-APLCA

Let $\omega(r,s,t)$ be the minimal exponent so that the Boolean product of an $n^r \times n^s$ matrix with an $n^s \times n^t$ matrix can be computed in $O(n^{\omega(r,s,t)})$ time. Recall that $\omega = \omega(1,1,1) < 2.376$. For the proof of Theorem 1.2 we will need bounds for $\omega(1,r,1)$, the exponent of rectangular matrix multiplication of a particular form.

We define two more constants, α and β, related to rectangular matrix multiplication.

Definition 3.1 $\alpha = \sup\{\, 0 \le r \le 1 \mid \omega(1,r,1) = 2 + o(1) \,\}$, $\beta = \dfrac{\omega - 2}{1 - \alpha}$.

The following result has been proved by Coppersmith [4].

$$\alpha > 0.294 \,. \tag{1}$$

It is not difficult to see that (1) implies the following theorem. A proof can be found, for example, in Huang and Pan [11].

Theorem 3.2 $\omega(1,r,1) \le \begin{cases} 2 + o(1) & \text{if } 0 \le r \le \alpha, \\ 2 + \beta(r - \alpha) + o(1) & \text{otherwise.} \end{cases}$

Note that with $\omega = 2.376$ and $\alpha = 0.294$ we get $\beta \simeq 0.533$. If $\omega = 2$ then $\alpha = 1$. (In this case β is not defined, but also not needed.) Another point to notice is that the algorithm of Alon and Naor [1] finds witnesses to the Boolean product of an $n \times n^r$ matrix with an $n^r \times n$ matrix in $\tilde{O}(n^{\omega(1,r,1)})$ time.

Proof of Theorem 1.2: Suppose our input dag $G = (V,E)$ has its vertices $\{1,\ldots,n\}$ given in topological order. We begin by computing the transitive closure matrix T_G in $O(n^\omega)$ time.

We consider first the k-APCA problem for $k \ge 3$. Assume first that k is even. Let $\mathcal{S}_{k/2}$ be the set of all $\binom{n}{k/2}$ subsets of $k/2$ vertices. We create a Boolean matrix R with rows indexed by $\mathcal{S}_{k/2}$ and columns indexed by V. For each $S \in \mathcal{S}_{k/2}$ and for each $v \in V$, we let $R(S,v) = 1$ if v is a common ancestor of each element of S. Otherwise, $R(S,v) = 0$. Notice that given T_G, R can be constructed in $O(n^{k/2+1})$ time. Consider the Boolean product $M = RR^t$. Suppose S is a k-subset and suppose that $S_1 \cup S_2 = S$ is an arbitrary partition of S with $|S_1| = |S_2| = k/2$. Clearly, $M(S_1,S_2) = 1$ if and only if S has a common ancestor. Thus, if W is a matrix of witnesses for the Boolean product $M = RR^t$ then W contains the solution to the k-APCA problem. In fact, for each k-subset S there are $\binom{k}{k/2}$ locations in W that provide (possibly distinct) common ancestors of S.

Let $N = n^{k/2}$. The matrix R has $\Theta(N)$ rows and $N^{2/k}$ columns. Thus, by Theorem 3.2 using $r = 2/k$, and the comment following it, a k-APCA solution can be computed in $\tilde{O}(N^{\omega(1,2/k,1)})$ time. For all $k \ge 8$ we have $2/k \le 1/4 < \alpha$ and hence $\omega(1,2/k,1) = 2 + o(1)$. Thus, a k-APCA solution can be computed in $\tilde{O}(N^2) = \tilde{O}(n^k)$ time, as required. We note that for $k = 6$ we can use $\omega(1,1/3,1) < 2.021$ and obtain that a 6-APCA solution can be computed in $O(n^{6.063})$ time. Likewise, a 4-APCA solution is computed in $O(n^{4.22})$ time.

If k is odd we can reduce the k-APCA problem to n subproblems of $(k-1)$-APCA. Indeed, fix any vertex u, and let R_u be a matrix with rows indexed by all $(k-1)/2$-subsets of $V - \{u\}$ and columns indexed by V. We let $R_u(S,v) = 1$ if v is a common ancestor of $S \cup \{u\}$. Clearly, a witness matrix for $R_u \times R_u^t$ contains the required solution for all k-subsets that contain u. By performing this for each $u \in V$, the solution to the k-APCA problem is obtained.

We now consider the k-APLCA problem for $k \geq 3$. Assume first that k is even and let $m = \sqrt{n}$ (we may assume that m is an integer as this will not affect the asymptotic nature of our result). We partition V into m parts $V_1, \ldots, V_m$ where $V_j = \{(j-1)m+1, \ldots, jm\}$ for $j = 1, \ldots, m$. Let $\mathcal{S}_{k/2}$ be the set of all $\binom{n}{k/2}$ subsets of $k/2$ vertices. We create m Boolean matrices $R_1, \ldots, R_m$ with the rows of R_j indexed by $\mathcal{S}_{k/2}$ and columns indexed by V_j. For each $S \in \mathcal{S}_{k/2}$ and for each $v \in V_j$, we let $R_j(S, v) = 1$ if v is a common ancestor of each element of S. Otherwise, $R_j(S, v) = 0$. Notice that given T_G, R_j can be constructed in $O(n^{k/2+1/2})$ time. Consider the Boolean products $M_j = R_j R_j^t$ for $j = 1, \ldots, m$. Suppose S is a k-subset and suppose that $S = S_1 \cup S_2$ is an arbitrary partition of S with $|S_1| = |S_2| = k/2$. It takes $O(m)$ time to locate the largest index j so that $M_j(S_1, S_2) = 1$. If no such j exists we conclude that S does not have a common ancestor. Otherwise, we look, in $O(m)$ time, for the largest $v \in V_j$ for which $R_j(S_1, v) = 1$ and $R_j(S_2, v) = 1$. Clearly, the largest v found is an LCA of S. Thus, the running time to compute a solution to the k-APLCA problem is at most the time needed to compute the m products $M_j = R_j R_j^t$.

Let $N = n^{k/2} = m^k$. The matrix R_j has $\Theta(N)$ rows and $N^{1/k}$ columns. Thus, by Theorem 3.2 using $r = 1/k$, and the comment following it, a k-APLCA solution can be computed in $\tilde{O}(mN^{\omega(1,1/k,1)})$ time. Since $k \geq 4$ we have $1/k \leq 1/4 < \alpha$ and hence $\omega(1, 1/k, 1) = 2 + o(1)$. Thus, a k-APLCA solution can be computed in $\tilde{O}(mN^2) = \tilde{O}(n^{k+1/2})$ time, as required. The case where k is odd is reduced to n subproblems of $(k-1)$-APLCA, as for the k-APCA problem. ■

4 Concluding remarks and open problems

We have shown that computing junctions (which are special types of common ancestors) for all pairs of vertices can be done in essentially the same time needed to find just arbitrary common ancestors.

Not all junctions are of the same quality. The *ancestral distance* between two vertices u and v of a dag that have a common ancestor, denoted $d(u, v)$, is the minimum sum of lengths of paths from a common ancestor to each of them. Clearly, a common ancestor exhibiting $d(u, v)$ is also a $\{u, v\}$-junction. However, the fastest algorithm for computing $d(u, v)$ for all pairs of vertices runs in $O(n^{2.575})$ time, as shown in [2]. The problem is reduced to an instance of the *all-pairs shortest paths* (APSP) problem. In fact, it is not difficult to use the APSP algorithm of Zwick [16] so that a junction exhibiting $d(u, v)$ is *found* in $O(n^{2.575})$ time, for all plausible pairs. It would be interesting to find a $\tilde{O}(n^\omega)$ time algorithm for APJ which guarantees that the junctions found exhibit the shortest ancestral distance or, at least, are close to it. The *approximate* APSP algorithm of Zwick, which does run in $\tilde{O}(n^\omega)$ time does guarantee an ancestor (but not necessarily a junction) which approximates the ancestral distance.

Acknowledgment

I wish to thank Uri Zwick and the referees for some useful comments.

References

[1] N. Alon and M. Naor. Derandomization, witnesses for Boolean matrix multiplication and construction of perfect hash functions. *Algorithmica*, 16:434–449, 1996.

[2] M.A. Bender, M. Farach-Colton, G. Pemmasani, S. Skiena, and P. Sumazin. Lowest common ancestors in trees and directed acyclic graphs. *Journal of Algorithms*, 57(2):75–94, 2005.

[3] O. Berkman and U. Vishkin. Finding level-ancestors in trees. *Journal of Computer and System Sciences*, 48(2):214–230, 1994.

[4] D. Coppersmith. Rectangular matrix multiplication revisited. *Journal of Complexity*, 13:42–49, 1997.

[5] D. Coppersmith and S. Winograd. Matrix multiplication via arithmetic progressions. *Journal of Symbolic Computation*, 9:251–280, 1990.

[6] A. Czumaj, M. Kowaluk, and A. Lingas. Faster algorithms for finding lowest common ancestors in directed acyclic graphs. *Theoretical Computer Science*, 380(1-2):37–46, 2007.

[7] S. Eckhardt, A. Mühling, and J. Nowak. Fast lowest common ancestor computations in dags. *Proceedings of the 15th Annual European Symposium on Algorithms (ESA)*, Lecture Notes in Computer Science 4698, Springer (2007), 705–716.

[8] M.J. Fischer and A.R. Meyer. Boolean matrix multiplication and transitive closure. In *Proceedings of the 12th Symposium on Switching and Automata Theory*, East Lansing, Michigan, (1971), 129–131.

[9] M.E. Furman. Application of a method of fast multiplication of matrices in the problem of finding the transitive closure of a graph. Dokl. Akad. Nauk SSSR, 11 (1970), no. 5, p. 1252.

[10] D. Harel and R.E. Tarjan. Fast algorithms for finding nearest common ancestors. *SIAM Journal on Computing*, 13(2):338-355, 1984.

[11] X. Huang and V.Y. Pan. Fast rectangular matrix multiplications and applications. *Journal of Complexity*, 14:257–299, 1998.

[12] M. Kowaluk and A. Lingas. LCA queries in directed acyclic Graphs. In *Proceedings of the 32nd International Colloquium on Automata, Languages, and Programming (ICALP)*, Lecture Notes in Computer Science 3580, Springer (2005), 241–248.

[13] M. Kowaluk and A. Lingas. Unique lowest common ancestors in Dags are almost as easy as matrix multiplication. *Proceedings of the 15th Annual European Symposium on Algorithms (ESA)*, Lecture Notes in Computer Science 4698, Springer (2007), 265–274.

[14] I. Munro. Efficient determination of the strongly connected components and the transitive closure of a graph. Unpublished manuscript, Univ. of Toronto, Toronto, Canada, 1971.

[15] B. Schieber and U. Vishkin. On finding lowest common ancestors: Simplification and parallelization. *SIAM Journal on Computing*, 17:1253–1262, 1988.

[16] U. Zwick. All-pairs shortest paths using bridging sets and rectangular matrix multiplication. *Journal of the ACM*, 49:289–317, 2002.