IEICE TRANS. INF. & SYST., VOL.E91–D, NO.2 FEBRUARY 2008

PAPER Special Section on Foundations of Computer Science

Longest Path Problems on Ptolemaic Graphs***

Yoshihiro TAKAHARA†*, Sachio TERAMOTO†**, *Nonmembers, and* Ryuhei UEHARA†a), *Member*

SUMMARY Longest path problem is a problem for finding a longest path in a given graph. While the graph classes in which the Hamiltonian path problem can be solved efficiently are widely investigated, there are few known graph classes such that the longest path problem can be solved efficiently. Polynomial time algorithms for finding a longest cycle and a longest path in a Ptolemaic graph are proposed. Ptolemaic graphs are the graphs that satisfy the Ptolemy inequality, and they are the intersection of chordal graphs and distance-hereditary graphs. The algorithms use the dynamic programming technique on a laminar structure of cliques, which is a recent characterization of Ptolemaic graphs.

key words: *dynamic programming, Hamiltonian path/cycle problem, longest path/cycle problem, Ptolemaic graphs*

1. Introduction

The Hamiltonian path and cycle problems are the most well known $\mathcal{NP}$-hard problems having numerous applications [1]. There are two major approaches for such intractable problems; approximation algorithms and algorithms with parameterized complexity analyses. We have to change the decision problems to optimization ones in both approaches. Thus the longest path/cycle problems are basic problems from the viewpoint of combinatorial optimization. It is also natural to try to find a longest path/cycle in a given graph even if it does not have a Hamiltonian path/cycle from the practical point of view. However, the longest path/cycle problems seem to be more difficult than the Hamiltonian path/cycle problems: It is impossible to find a path of length $n-n^{\epsilon}$ in polynomial time for any $\epsilon < 1$ unless $\mathcal{P} = \mathcal{NP}$ even if a graph has a Hamiltonian path [2]. The longest path/cycle problems are hard to approximate. They are not in $\mathcal{APX}$ unless $\mathcal{P} = \mathcal{NP}$. The best known approximation algorithm by Gabow [3] finds a cycle of length $\exp(\Omega(\sqrt{\log \ell / \log\log \ell}))$ in polynomial time in a graph that contains a longest cycle of length ℓ (see also [4]–[6] for related results).

We turn to the polynomial time algorithms that find exact solutions for a given restricted graph. There are few polynomial time algorithms of the longest path/cycle problems for restricted graph classes. A linear time algorithm for finding a longest path in a tree was invented by Dijkstra around 1960, and the formal proof is given by Bulterman et al. [7]. Recently, Uehara and Uno propose two approaches [8]; one is an extension of the Dijkstra's algorithm to some tree-like graph classes, and the other is dynamic programming to some graph classes that have interval representations. As the extensions of the Dijkstra's algorithm, they show polynomial time algorithms for weighted trees, block graphs, Ptolemaic graphs, and cactus.

However, the algorithm for a Ptolemaic graph ([8, Theorem 3]) contains an error [9], and fails in general. Later, Uehara and Uno investigate the laminar structure of Ptolemaic graphs, and show a linear time algorithm for the Hamiltonian cycle problem on a Ptolemaic graph, which gives a partial answer to the longest path problem on a Ptolemaic graph [10]. We extend their algorithm to the longest path problem. In other words, we complete to correct the wrong algorithm in [8] by the following theorem:

Theorem 1: For a Ptolemaic graph $G = (V, E)$, (1) a longest cycle can be found in $O(|V|^3)$ time and $O(|V|^2)$ space, and (2) a longest path can be found in $O(|V|^5)$ time and $O(|V|^2)$ space.

A Ptolemaic graph G satisfies Ptolemy inequality for any four vertices, which is originally a relation in Euclidean geometry between the four sides and two diagonals of a quadrilateral. Intuitively, each edge in a Ptolemaic graph connects "near" vertices in Euclidean space. From the viewpoint of graph theory, Ptolemaic graphs are the intersection of chordal graphs and distance-hereditary graphs. The Hamiltonian path problem is $\mathcal{NP}$-complete on chordal graphs [11]. Hence we have no polynomial time algorithm for the longest path problem on chordal graphs unless $\mathcal{P} = \mathcal{NP}$. Recently, some polynomial time algorithms for the Hamiltonian problems on distance-hereditary graphs are shown [12]–[15]. They stand on the characterization by Bandelt and Mulder [16], which seems to be hard to extend to the longest path problem. The new characterization in [10] by a simple laminar structure allows us to use the dynamic programming technique.

2. Preliminaries

The *neighbor* set of a vertex v in a graph $G = (V, E)$ is the set $N_G(v) = \{u \in V \mid \{u, v\} \in E\}$, and the *degree* of a vertex v is $|N_G(v)|$ and is denoted by $\deg_G(v)$. For a subset U of V, we

Manuscript received March 27, 2007.
Manuscript revised June 18, 2007.

†The authors are with School of Information Science, JAIST, Nomi-shi, 923–1292 Japan.
*Presently, with Sekisui House.
**Presently, with Service Platforms Research Laboratories, NEC Corporation.

***An extended abstract was presented at KyotoCGGT 2007. The authors are partially supported by the Ministry, Grant-in-Aid for Scientific Research (C).

a) E-mail: uehara@jaist.ac.jp

DOI: 10.1093/ietisy/e91–d.2.170

Copyright © 2008 The Institute of Electronics, Information and Communication Engineers

denote by $N_G(U)$ the set $\{v \in V \mid v \in N(u) \text{ for some } u \in U\}$. If no confusion can arise we will omit the index G. Given a graph $G = (V, E)$ and a subset U of V, the induced subgraph by U, denoted by $G[U]$, is the graph (U, E'), where $E' = \{\{u, v\} \mid u, v \in U \text{ and } \{u, v\} \in E\}$. A vertex set U is a *clique* if every pair of vertices in U is joined by an edge.

Given a graph $G = (V, E)$, a sequence of the distinct vertices $v_1, v_2, \ldots, v_\ell$ is a *path*, denoted by $(v_1, v_2, \ldots, v_\ell)$, if $\{v_j, v_{j+1}\} \in E$ for each $1 \leq j < \ell$. A sequence of the distinct vertices $v_1, v_2, \ldots, v_\ell$ is a *cycle*, denoted by $(v_1, v_2, \ldots, v_\ell, v_1)$, if $(v_1, v_2, \ldots, v_\ell)$ is a path and $\{v_1, v_\ell\} \in E$. The *length* of a path P (denoted by $|P|$) and a cycle C (denoted by $|C|$) is the number of edges of the path and cycle, respectively. Hence we have $|P| = \ell - 1$ for a path $P = (v_1, v_2, \ldots, v_\ell)$, while $|C| = \ell$ for a cycle $C = (v_1, v_2, \ldots, v_\ell, v_1)$. *Longest path* (and *cycle*) *problem* is the problem for finding a longest path (and cycle, respectively) in a given graph. *Hamiltonian path* (and *cycle*) *problem* is the problem for finding a path (and cycle, respectively) that visits each vertex in a given graph exactly once. An edge which joins two vertices of a cycle but is not itself an edge of the cycle is a *chord* of that cycle. A graph is *chordal* if and only if each cycle of length at least 4 has a chord.

For two vertices u and v, the *distance* of the vertices, denoted by $d(u, v)$, is the minimum length of paths joining u and v. Given a graph $G = (V, E)$ and a subset U of V, an induced connected subgraph $G[U]$ is *isometric* if the distances in $G[U]$ are the same as in G. A graph G is *distance-hereditary* if and only if G is connected and every induced path in G is isometric.

A connected graph G is *Ptolemaic* if and only if we have the Ptolemy inequality $d(u, v)d(w, x) \leq d(u, w)d(v, x) + d(u, x)d(v, w)$ for any four vertices u, v, w, x of G. The Ptolemy inequality holds for any apexes of a quadrilateral in Euclidean geometry; that is, a Ptolemaic graph is a graph that has a geometric property. It is known that a graph G is Ptolemaic if and only if G is distance-hereditary and chordal [17].

Let V be a set of n vertices. Two sets X and Y *overlap* if and only if $X \cap Y \neq \emptyset$, $X \setminus Y \neq \emptyset$, and $Y \setminus X \neq \emptyset$. A family $\mathcal{F} \subseteq 2^V$ is *laminar* if and only if $\mathcal{F}$ contains no overlapping sets; that is, for any pair of two distinct sets X and Y in $\mathcal{F}$ satisfy either $X \cap Y = \emptyset$, $X \subset Y$, or $Y \subset X$.

For any given graph $G = (V, E)$, two sets of cliques are defined by $\mathcal{M}(G) := \{M \mid M \text{ is a maximal clique in } G\}$, and $\mathcal{L}(G) := \{L \mid L \text{ is a non-empty intersection of two or more cliques in } \mathcal{M}\}$. We denote by $C(G) := \mathcal{M}(G) \cup \mathcal{L}(G)$. For a maximal clique M in $\mathcal{M}(G)$, we denote by $C_M(G) := \{C \mid C \in C(G) \text{ and } C \subseteq M\}$. Then a graph G is Ptolemaic if and only if $C_M(G)$ is laminar for any maximal clique M [10].

For given Ptolemaic graph $G = (V, E)$, we can construct from $C(G)$, a directed tree $\vec{T}(C(G)) = (C(G), \vec{A})$ as follows [10]; $\vec{A}$ contains an arc (C_1, C_2) if and only if $C_2 \subset C_1$ and there are no other clique C such that $C_2 \subset C \subset C_1$ for any cliques C_1 and C_2 in $C(G)$. We denote the underlying graph of $\vec{T}(C(G))$ by $T(C(G)) = (C(G), A)$, where A is the set of edges obtained from $\vec{A}$ by ignoring directions. For given Ptolemaic graph G, the (directed) tree is uniquely determined up to isomorphism. Hence it is called *clique laminar tree* (CL-tree) of G. We borrow some notations from directed trees, and use slightly different meanings like roots (with indegree 0), leaves (with one associate edge), ancestors, and descendants. For example, each maximal clique is a *root* on $\vec{T}(C(G))$ since it has no indegree on the CL-tree. On the other hand, a leaf L is a maximal clique in G since any clique in $\mathcal{L}(G)$ has at least two parents in $\vec{T}(C(G))$. Thus every leaf L in $T(C(G))$ has one outdegree on $\vec{T}(C(G))$.

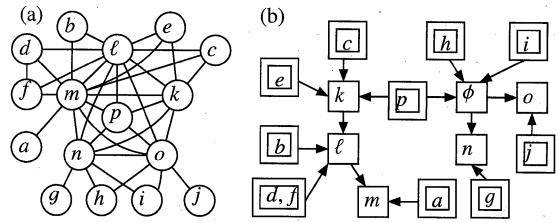


Fig. 1 Ptolemaic graph G and its CL-tree $\vec{T}(C(G), A)$.

We define a *label* of each node C in $\vec{T}(C(G))$, denoted by $\ell(C)$, as follows: If C is a leaf, $\ell(C) = C$. If C is not a leaf and has children $S_1, S_2, \ldots, S_h$, $\ell(C) = C \setminus (S_1 \cup S_2 \cup \cdots \cup S_h)$. That is, each vertex v in V appears in $\ell(C)$ where C is the minimal clique containing v. Since $C(G)$ is laminar, each vertex in V appears exactly once in $\ell(C)$ for some $C \subseteq V$, and its corresponding node is uniquely determined. We note that some internal nodes in $\vec{T}(C(G))$ have a label $\emptyset$ when it is partitioned completely into its subsets in $C(G)$.

For example, the CL-tree $\vec{T}(C(G))$ of the graph in Fig. 1 (a) is depicted in Fig. 1 (b); each double rectangle is a maximal clique in $\mathcal{M}(G)$ and each single rectangle is a non-maximal clique in $\mathcal{L}(G)$. The labels $\ell(C)$ of the cliques C are described in the rectangles; e.g., the center double rectangle with label p is the maximal clique $\{k, \ell, m, n, o, p\}$, which is the union of labels reachable from the rectangle. The CL-tree $\vec{T}(C(G))$ has at most $2|V|$ nodes, and it can be constructed in $O(|V| + |E|)$ time and space [10]. Hereafter, we assume that a Ptolemaic graph G is given in the form of the CL-tree $\vec{T}(C(G))$ as the labeled tree.

3. Proof of Theorem 1

The CL-tree $\vec{T}(C(G))$ plays an important role in our algorithms. The first point is that each node $L \in \mathcal{L}(G)$ is a separator of G. That is, if we have two arcs (L_1, L) and (L_2, L) in $\vec{A}$, L separates $L_1 \setminus L$ and $L_2 \setminus L$ [10], which allows us to use dynamic programming. The second point is that each node $C \in C(G)$ induces a clique. They lead us to the following lemma:

Lemma 1: Let P be any path in a Ptolemaic graph G. Let C_1, C_2 be two cliques in $C(G)$ with $C_1 \subset C_2$ such that P contains some edges joining two vertices in $\ell(C_1)$ and $\ell(C_2)$. Then, we can (re)connect P and obtain a new path P' such that (1) P' contains at most two edges joining two vertices

in $\ell(C_1)$ and $\ell(C_2)$, (2) P' has the same length of P, and (3) P' shares two endpoints with P.

Proof. Assume that P contains three independent edges $\{v_1, u_1\}$, $\{v_2, u_2\}$, and $\{v_3, u_3\}$ between $\ell(C_1)$ and $\ell(C_2)$ with $v_1, v_2, v_3 \in \ell(C_1)$ and $u_1, u_2, u_3 \in \ell(C_2)$. (The case that some edges share a common endpoint is similar, and omitted here.) Without loss of generality, we can assume that P consists of seven subpaths $\mathsf{P}_1 = (\ldots, v_1)$, $\mathsf{P}_2 = (v_1, u_1)$, $\mathsf{P}_3 = (u_1 \ldots, u_2)$, $\mathsf{P}_4 = (u_2, v_2)$, $\mathsf{P}_5 = (v_2, \ldots, v_3)$, $\mathsf{P}_6 = (v_3, u_3)$, and $\mathsf{P}_7 = (u_3, \ldots)$. Then, the sets $\{v_1, v_2, v_3\}$ and $\{u_1, u_2, u_3\}$ induce cliques since $\{v_1, v_2, v_3\} \subseteq \ell(C_1) \subseteq C_1$ and $\{u_1, u_2, u_3\} \subseteq \ell(C_2) \subseteq C_2$. Hence we can construct P' that consists of the subpaths $\mathsf{P}_1 = (\ldots, v_1)$, (v_1, v_2), $\mathsf{P}_5 = (v_2, \ldots, v_3)$, (v_3, u_1), $\mathsf{P}_3 = (u_1 \ldots, u_2)$, (u_2, u_3), and $\mathsf{P}_7 = (u_3, \ldots)$. Repeating this process, we can decrease the number of edges between $\ell(C_1)$ and $\ell(C_2)$ to at most 2 without changing the length and two endpoints of the path. □

We first show a polynomial time algorithm for finding a longest cycle in a Ptolemaic graph, and next extend it to find a longest path.

3.1 Longest Cycle

We first prove Theorem 1 (1). The algorithm is based on the standard dynamic programming. That is, the algorithm performs the computation along the edges of the underlying graph $T(C(G))$ of the CL-tree $\vec{T}(C(G))$. Except the last step, the algorithm deals with one node L whose all neighbors have been already processed by the algorithm except one neighbor X. We here note that X is a neighbor of L on the underlying graph $T(C(G))$. That is, X is either a parent of L or a child of L on the CL-tree $\vec{T}(C(G))$. Then, from $T(C(G))$, removing the edge between X and L, we obtain two subtrees of $T(C(G))$. The subtree T' containing L is the underlying graph of the CL-tree $\vec{T}'$ of a Ptolemaic graph G' induced by the set of vertices in V on the subtree, and all nodes of the tree T' have been processed by the algorithm. We call a longest cycle in G' *local longest cycle at* L, which is denoted by $LLC(L)$. We denote by $|LLC(L)|$ the length of the local longest cycle at L. (In this section, we will show the algorithms for computing the length of cycles and paths. However, it is easy to modify them to output cycle and path themselves.) The outline of the algorithm is described in Algorithm 1.

We describe the details of the algorithm. Let $p(L)$ and $c(L)$ be the numbers of parents and children of L except X, respectively. To simplify, we distinguish X from the other nodes. Let $P_1, P_2, \ldots, P_{p(L)}$ and $C_1, C_2, \ldots, C_{c(L)}$ be the parents and children of L on $\vec{T}(C(G))$, respectively. That is, we have $C_j \subset L \subset P_i$ with $1 \le i \le p(L)$ and $1 \le j \le c(L)$.

By Lemma 2 in [10], L separates G into $G_1, G_2, \ldots, G_{p(L)}$ such that G_i contains $P_i \setminus L$, and by Lemma 1, it is sufficient that L provides exactly two edges that join L and one ancestor to construct a longest cycle. We note that when L is joined to an ancestor A, the path has to go through a parent P_i with $L \subset P_i \subset A$.

Input : A CL-tree $\vec{T}(C(G))$
Output: The length of a longest cycle in G.
1 let Q be a queue of the leaves in $T(C(G))$;
2 **while** $Q \neq \emptyset$ **do**
3 pick up a node L in Q;
// L has only one unprocessed neighbor.
4 let X be the unique unprocessed neighbor of L;
// X is either parent or child of L.
5 update tables for L;
6 compute $|LLC(L)|$;
7 put X into Q if X has only one unprocessed neighbor;
8 **end**
// There remains only one node L' having all processed neighbors.
9 update tables for the last node L';
10 compute $|LLC(L')|$;
11 output $\max_L |LLC(L)|$;

Algorithm 1: Longest Cycle

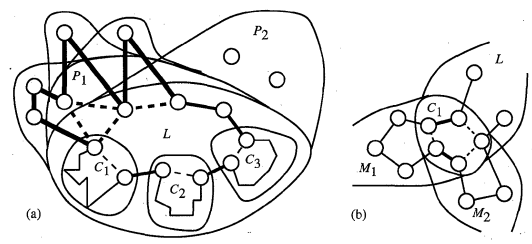


Fig. 2 Node L with its parents and children.

The basic strategy to construct a longest cycle is similar to the algorithm for finding a Hamiltonian cycle in [10]; each node L uses its own vertices in $\ell(L)$, and borrows some vertices from its children to connect paths provided by the parents of L. The algorithm expands an edge of a cycle in L or its children by replacing a longer path given by a parent. We note that L has $|\ell(L)| + c(L)$ edges in this level, which can be expanded to parents, joining the vertices in $\ell(L)$ and children. L may be able to provide more edges from its children, which cannot be computed in this level. For example, the node L in Fig. 2 (a) has $|\ell(L)|+c(L) = 3+3 = 6$ edges, and two edges of them are used to obtain five vertices from P_1 (and its ancestors). This expansion brings profits of length five to L.

3.1.1 Profit Tables

Each P_i and C_j have their *profit tables* $\Delta_{P_i}[k]$ and $\Gamma_{C_j}[k]$, respectively. From the tables, the algorithm computes the profit table of L which will be provided to X.

We first deal with the table $\Delta_{P_i}[k]$. Let G_i be the connected component that contains the vertices in $P_i \setminus L$ in graph induced by $V \setminus L$. Then, for a parent P_i, the table $\Delta_{P_i}[k]$ maintains the following information: $\Delta_{P_i}[k]$ gives the maximum number of vertices in G_i provided by the parent P_i to L such that the algorithm can replace k edges in L and its children by k paths that consists of $\Delta_{P_i}[k]$ vertices in G_i. Intuitively, P_i can give k independent paths to L, and the total number of the vertices on the paths is $\Delta_{P_i}[k]$, which is

maximized. For example, if P_i is a leaf, put in Q in step 1, $\Delta_{P_i}[j] = |\ell(P_i)|$ for all $j > 0$. Precisely, L can obtain a path of length $\Delta_{P_i}[k] + k$ by giving k edges to P_i. We note that k is the number of *edges* in $G[L]$, while $\Delta_{P_i}[k]$ is the number of *vertices* in G_i. This disunity makes the arguments simple. We define $\Delta_{P_i}[0] := 0$, $\delta_i[k] := \Delta_{P_i}[k] - \Delta_{P_i}[k-1]$ for $k > 1$, and hence $\delta_i[1] = \Delta_{P_i}[1]$. For the tables Δ and δ, we have the following lemma:

Lemma 2: (1) For each P_i, Δ_{P_i} and δ_i takes $O(n)$ space. (2) For each k with $\Delta_{P_i}[k] > 0$, $0 \le \delta_i[k+1] \le \delta_i[k]$.

Proof. By definition of G_i, the maximum value of $\Delta_{P_i}[k]$ is at most the number of vertices in G_i. Hence there exists an integer $k \le n$ such that $\Delta_{P_i}[k] = \Delta_{P_i}[k']$ and $\delta_i[k'] = 0$ for all $k' > k$. Thus we only need to store $\Delta_{P_i}[k']$ and $\delta_i[k']$ for $\delta_i[k'] > 0$, which can be represented in $O(n)$ space. Therefore we have (1).

Clearly, L can obtain longer path from G_i when L gives $k+1$ edges to G_i than the case that L gives only k edges to G_i. Thus we always have $\delta_i[k] \ge 0$ if $\Delta_{P_i}[k] > 0$.

Let $l := \Delta_{P_i}[k]$ for some k, and $l' := \Delta_{P_i}[k+1]$. Then, l is the total number of the vertices on paths $\mathsf{P}_1, \mathsf{P}_2, \ldots, \mathsf{P}_k$ in G_i that replace at most k edges in L. On the other hand, l' is the total number of the vertices on paths $\mathsf{P}'_1, \mathsf{P}'_2, \ldots, \mathsf{P}'_k, \mathsf{P}'_{k+1}$ in G_i that replace at most $k+1$ edges in L. Without loss of generality, we assume that $|\mathsf{P}_1| \ge |\mathsf{P}_2| \ge \cdots \ge |\mathsf{P}_k|$ and $|\mathsf{P}'_1| \ge |\mathsf{P}'_2| \ge \cdots \ge |\mathsf{P}'_k| \ge |\mathsf{P}'_{k+1}|$. By a simple induction for k with the same technique in the proof of Lemma 1, the paths can be arranged to satisfy $|\mathsf{P}_i| = |\mathsf{P}'_i|$ for each $i = 1, 2, \ldots, k$. In the case, $\delta_i[k+1] = |\mathsf{P}'_{k+1}|$ and $\delta_i[k] = |\mathsf{P}_k| = |\mathsf{P}'_k|$. Thus we have $\delta_i[k+1] \le \delta_i[k]$. □

Next we turn to the table $\Gamma_{C_j}[k]$ for the maintenance of paths and profits from children C_j. The table $\Gamma_{C_j}[k]$ is more involved than Δ. Intuitively, for each k, L can obtain k edges from C_j to be extended in L, and a path of length $\Gamma_{C_j}[k] - 1$ from C_j. Precisely, $\Gamma_{C_j}[k]$ gives the maximum number of vertices provided by the child C_j to L such that L can obtain k edges and a path of $\Gamma_{C_j}[k]$ vertices from C_j, which is maximized. When $k \ge |C_j|$, we define $\Gamma_{C_j}[k] = -\infty$ and inhibit the case. We note that all leaves L on the underlying graph $T(C(G))$ of the CL-tree $\vec{T}(C(G))$ is a maximal clique on G. Hence every node L put in Q in step 1 has profit table $\Delta_L[k]$; that is, we do not need to define any initial value of $\Gamma_L[k]$ for the leaves.

For example, we consider the node L in Fig. 2 (b) with its child C_1. The child C_1 is shared by three cliques L, M_1, M_2. By assumption of dynamic programming, M_1 and M_2 have been already processed by the algorithm. In the case, we have $\Gamma_{C_1}[0] = \Gamma_{C_1}[1] = \Gamma_{C_1}[2] = 10$, $\Gamma_{C_1}[3] = 8$, and $\Gamma_{C_1}[4] = 5$. Then L cannot use the fifth edge in C_1 since we already use it to join C_1 to L. Hence we have $\Gamma_{C_1}[5] = -\infty$.

Clearly, if L takes more edges from C_j, L obtains shorter path from C_j. We now define $\gamma_j[k] := \Gamma_{C_j}[k] - \Gamma_{C_j}[k+1]$ if $\Gamma_{C_j}[k] > 0$, and $\gamma_j[k] := \infty$ if $\Gamma_{C_j}[k] = -\infty$. For the tables Γ and γ, we have the following lemma.

Lemma 3: (1) For each C_j, Γ_{C_j} and γ_j takes $O(n)$ space. (2) For each k with $\Gamma_{C_j}[k] > 0$, $0 \le \gamma_j[k] \le \gamma_j[k+1]$.

The proof is essentially the same as Lemma 2, and hence omitted.

To perform dynamic programming in polynomial time, we have to merge the tables in polynomial time. The following two lemmas play key roles for the polynomial time computability.

Lemma 4: We assume that all tables Δ of parents P_i of the current node L (except X) are computed. Fix an integer k with $k > 0$. Then the length of a longest path among the paths P such that (1) P uses exactly $k+1$ vertices in L and other vertices in $G_1 \cup G_2 \cup \cdots \cup G_{p(L)}$ (2) P has two endpoints in L can be computed in $O(n \log n)$ time and $O(n)$ space.

Proof. When $k = 1$, we find $l := \max_{1 \le i \le p(L)} \Delta_{P_i}[1]$, and obtain a path of length $l+1$ that consists of l vertices in the parent and two endpoints in L. We show that we can extend it incrementally, or in a greedy way. The description of the algorithm is given in Algorithm 2.

We note that Q is a priority queue in decreasing order of $\delta_i[q_i]$. Hence, step 4 can be done in $O(1)$ time, and step 7 requires $O(\log n)$ time. Thus the greedy algorithm runs in $O(n \log n)$ time and $O(n)$ space. Hence we show the correctness. We let $q'_1, q'_2, \ldots, q'_{p(L)}$ be values at the last line of the algorithm. We define $l := \sum_{i=1}^{p(L)} \Delta_{P_i}[q'_i]$. Then, by Lemma 2, we can construct a path of length $l + k$ that use $k+1$ vertices in L and l vertices in $G_1 \cup G_2 \cup \cdots G_{p(L)}$ (the l vertices induces k paths).

Hence, to derive a contradiction, we assume that there is a longer path P' of length $l' > l$ by other values $q''_1, q''_2, \ldots, q''_{p(L)}$ where $l' := \sum_{i=1}^{p(L)} \Delta_{P_i}[q''_i]$ and $\sum_{i=1}^{p(L)} q''_i = k$. We assume that P' is the longest one among such paths.

Now, by definition, we have $l = \sum_{i=1}^{p(L)} \Delta_{P_i}[q'_i] = \sum_{i=1}^{p(L)} \sum_{q=1}^{q'_i} \delta_i[q]$. Then, since k is fixed, there are two integers i and i' such that $q'_i > q''_i$ and $q'_{i'} < q''_{i'}$. In some iteration, the algorithm chooses the variable q_i and increments from q''_i to $q''_i + 1$. Then, since the algorithm is greedy, $\delta_i[q''_i] > \delta_{i'}[q_{i'}]$. (we omit the case $\delta_i[q''_i] = \delta_{i'}[q_{i'}]$ in this paper, since it is tedious and not essentially). Then, by Lemma 2, $\delta_i[q''_i] > \delta_{i'}[q_{i'}] \ge \delta_{i'}[q''_{i'}]$. Hence, replacing $q''_{i'}$ by $q''_{i'} - 1$ and q''_i by $q''_i + 1$, we can improve the length of the longest path P' of length l', which contradicts the assump-

Input : Table $\Delta_{P_1}, \Delta_{P_2}, \ldots, \Delta_{P_{p(L)}}$ and integer k
Output: The length of a longest path using $k+1$ vertices in L and its parents.

```
1 foreach i = 1, 2, ..., p(L) do initialize q_i := 1;
2 let Q be a priority queue of δ_i s in decreasing order of δ_i[1];
3 while k > 0 do
4     pick up the largest element δ_i[q_i] in Q;
5     q_i := q_i + 1;
6     k := k − 1;
7     push δ_i into Q again according to the value of δ_i[q_i];
8 end
9 return (Σ_{i=1}^{p(L)} Δ_{P_i}[q_i − 1]);
```

Algorithm 2: Merge Δs

Input : Tables $\Gamma_{C_1}, \Gamma_{C_2}, \ldots, \Gamma_{C_{c(L)}}$ and integer k
Output: The length of a longest path using the vertices in G'_js and L that contains k edges in L.

```
1  if k ≥ |∪_{j=1}^{c(L)} C_j| then return −∞;
2  foreach j = 1, 2, ..., c(L) do initialize q_j := 1;
3  let Q be a priority queue of γ_j s in increasing order of γ_j[1];
4  while k > 0 do
5  |   pick up the smallest element γ_j[q_j] in Q;
6  |   q_j := q_j + 1;
7  |   k := k − 1;
8  |   push γ_j into Q again according to the value of γ_j[q_j];
9  end
10 return (Σ_{j=1}^{c(L)} max{Γ_{C_j}[q_j − 1], 0});
```

Algorithm 3: Merge Γs

tion. Hence the greedy algorithm computes the length of a longest path with the condition in $O(n \log n)$ time and $O(n)$ space. □

Next we turn to the table Γ. Suppose that we remove the node L from $\vec{T}(C(G))$. Then $\vec{T}(C(G))$ is partitioned into $c(L) + p(L) + 1$ subtrees. Among them, for each j with $1 \le j \le c(L)$, let $\vec{T}_j$ be a subtree of $\vec{T}(C(G))$ that contains the node C_j. Since C_js are children of L, exactly $c(L)$ subtrees $\vec{T}_j$ are defined. Let G'_j be induced subgraph by the set of vertices appearing as labels on $\vec{T}_j$. We note that (1) the vertex set of G'_j is not contained in L since C_j is shared by some other cliques, (2) for each j and j' with $j \neq j'$, G'_j and $G'_{j'}$ have no common vertex, (3) all nodes in $\vec{T}_j$ have been processed by the algorithm. For the subgraphs, we have the symmetric lemma for Γ:

Lemma 5: We assume that all tables Γ of children C_j of the current node L (except X) are computed. Fix an integer k with $k > 0$. Then the length of a longest path among the paths P such that (1) P consists of vertices in $G'_1 \cup G'_2 \cup \cdots \cup G'_{c(L)} \cup G[L]$ (2) P contains exactly k edges in $G[L]$ can be computed in $O(n \log n)$ time and $O(n)$ space.

Proof. Using Lemma 3 instead of Lemma 2, we can use a symmetric argument of the proof of Lemma 4. Hence we have a greedy Algorithm 3 for children of L. When k is too large, precisely, $k \ge \left|\cup_{j=1}^{c(L)} C_j\right|$, the children cannot give k edges to L. In the case, the algorithm returns $-\infty$. The other case is symmetric. □

At node L, the algorithm computes two tables $\hat{\Delta}_L[k]$ and $\hat{\Gamma}_L[k]$ obtained by calling Algorithms 2 and 3 for each $k = 1, 2, \ldots$, respectively. By Lemmas 4 and 5, the tables can be computed in $O(n \log n)$ time and $O(n)$ space. Using the tables, the algorithm computes a local longest cycle at L, and prepares a table of L for X.

3.1.2 Local Longest Cycle at L

We first show the computation of a local longest cycle at the node L. We have two cases.
(1) L is a leaf of the underlying graph $T(C(G))$ of the CL-tree $\vec{T}(C(G))$. Then L is a maximal clique of G, and hence $X \subset L$, and we have an arc (L, X) on $\vec{T}(C(G))$. This is a simple case; we have $|LLC(L)| = |\ell(L)|$.
(2) L is not a leaf of $T(C(G))$. In the case, the algorithm joins the paths from its parents and children by edges in L and children. The node L itself has $|\ell(L)| + c(L)$ edges (as shown in Fig. 2 (a)), and it borrows k edges from its children for some integer k with $k \ge 0$. Then, the algorithm obtain a path of length $|\ell(L)| + c(L) + \hat{\Gamma}_L[k]$ in the vertices in L and children. Moreover, the algorithm can extend each of $|\ell(L)| + c(L) + k$ edges in $G[L]$ by replacing a path provided by the parents. The total length of the extension is given by $\hat{\Delta}_L[|\ell(L)| + c(L) + k]$. We define $|LLC_k(L)|$ as follows:

$$|LLC_k(L)| := |\ell(L)| + c(L) + \hat{\Gamma}_L[k] + \hat{\Delta}_L[|\ell(L)| + c(L) + k].$$

Roughly, parents can provide longer path if L gives more edges. On the other hand, L and children have a limit to provide extra edges without decreasing their own paths. Thus, for each $k = 0, 1, 2, \ldots$, the values of $|LLC_k(L)|$ are unimodal; that is, $|LLC_0(L)| \le |LLC_1(L)| \le \cdots \le |LLC_k(L)| > |LLC_{k+1}(L)| \ge |LLC_{k+2}(L)| \cdots$ for some $k \le n$. Then, $|LLC(L)|$ is given by $\max_k\{|LLC_k(L)|\}$. Hence the algorithm can compute $|LLC(L)|$ for the k in $O(n)$ time and $O(n)$ space for given $\hat{\Gamma}_L$ and $\hat{\Delta}_L$.

3.1.3 Profit Table of L to Node X

Next, the algorithm prepares the table of L for the node X. We have three cases.
(1) L is a leaf of $T(C(G))$. Then L is a maximal clique of G, and hence $X \subset L$. Thus $\Delta_L[k] = |\ell(L)|$ for any $k \ge 1$.
(2) L is not a leaf of $T(C(G))$, and X is a child of L on the CL-tree $\vec{T}(C(G))$. Then the algorithm computes Δ_L for X. If X gives an edge to L, L can provide $|LLC(L)|$ vertices to X. If X gives more edges to L, L may be able to provide more edges to X since L can save the edges which was taken from its children. Precisely, $\Delta_L[i]$ is given by

$$\max_k\{|\ell(L)| + c(L) + \hat{\Gamma}_L[k] + \hat{\Delta}_L[|\ell(L)| + c(L) + k\underline{+i}]\}$$

for $i > 0$, and $\Delta_L[0] := 0$. Using the similar monotonicity in Lemmas 2 and 3, the algorithm can compute Δ_L incrementally in $O(n^2)$ time and $O(n)$ space.
(3) L is not a leaf of $T(C(G))$, and X is a parent of L on the CL-tree $\vec{T}(C(G))$. Then the algorithm computes Γ_L for X, or L has to provide a part of cycle to X. At first, we define

$$\Gamma_L[i] := -\infty \text{ for all } i \ge 0 \text{ if } |L| < 2$$

since L has no contribution to X. When $|L| \ge 2$, we can see that $\Gamma_L[0] := LLC(L)$ since L has a path of length $|LLC(L)|$ with two endpoints in L. In general, L can provide i edges to X from L if $0 < i \le |\ell(L)| + c(L)$. On the other hand, if $i > |\ell(L)| + c(L)$, L has to provide more edges from its children. Hence, by letting $\hat{\Delta}_L[k] := -\infty$ for $k \le 0$, we can define $\Gamma_L[i]$ for $i > 0$ by

$$\max_k\{|\ell(L)| + c(L) + \hat{\Gamma}_L[k] + \hat{\Delta}_L[|\ell(L)| + c(L) + k\underline{-i}]\}$$

if $|L| \ge 2$.

Similarly, $\Gamma_L[i]$ is unimodal for $k = O(n)$. Thus the algorithm can compute Γ_L incrementally in $O(n^2)$ time and $O(n)$ space.

Hence we can compute the length of a longest cycle by Algorithm 1. Each node has its table, which takes $O(n)$ space. At each node L, merging the tables from parents and children takes $O(n \log n)$ time and $O(n)$ space, and the computation of $|LLC(L)|$ with Δ_L or Γ_L takes $O(n^2)$ time and $O(n)$ space. Hence Algorithm 1 runs in $O(n^3)$ time and $O(n^2)$ space in total, which completes the proof of Theorem 1 (1).

3.2 Longest Path

Now we turn to Theorem 1 (2). The basic idea for finding a longest path is simple; for each two cliques S and T in $C(G)$, the algorithm guesses $\ell(S)$ and $\ell(T)$ contain two endpoints of a longest path. When $S = T$, the path induces a cycle, which can be found by Algorithm 1. Hence, hereafter, we assume that $S \neq T$. When S and T are fixed, there is a unique path joining them on $T(C(G))$, which is called (S,T)-*path*. Then, for each node L not on the (S,T)-path, the algorithm performs the dynamic programming process for computing Δ_L and Γ_L stated in Sect. 3.1, which finds a set of long paths induced by the processed nodes. Next, the algorithm joins the paths along the (S,T)-path and obtains a longest path among the paths of two endpoints belonging to S and T, respectively. The following lemma reduces the number of choices of S and T.

Lemma 6: Let P be a longest path in G, and S and T be two nodes in $C(G)$ such that $\ell(S)$ and $\ell(T)$ contain two endpoints of P. Then S and T are maximal cliques.

Proof. To derive a contradiction, we assume that S is not a maximal clique. Then $C(G)$ contains at least two maximal cliques S_1 and S_2 with $S \subset S_1$ and $S \subset S_2$. Without loss of generality, we can assume that S_1 is on (S,T)-path, and S_2 is not. Then, joining a vertex in $S_2 \setminus S$ to P, we can obtain a longer path, which contradicts that P is a longest path. Hence S and T are maximal cliques. □

Hence the outline of the algorithm is given in Algorithm 4.

We now show the details of steps 10, 12, and 14. Intuitively, the tables Δ_L and Γ_L carry the length of a local longest cycle by an edge, which can be extended by the next node. On the other hand, Δ'_L and Γ'_L carry the length of a local longest path by a vertex, which is a current endpoint, and the vertex will be extended by the next node. We remind that Δ'_L and Γ'_L maintains the number of vertices, not the number of edges.

At node S in Step 10: First, $\hat{\Delta}$ and $\hat{\Gamma}$ are computed by Δ_{P_i} and Γ_{C_j} for parents and children of S in the same way as Algorithm 1. By Lemma 6, S is a maximal clique. Hence L_2 is a child of S. Then $\Delta'_S[i]$ is defined by

$$\max_k\{|\ell(S)| + c(S) + \hat{\Gamma}_L[k] + \hat{\Delta}_S[|\ell(S)| + c(S) + k + i\underline{-1}]\}$$

for $i \geq 0$. We have two differences comparing to Δ. First,

Input : A CL-tree $\vec{T}(C(G))$
Output: The length of a longest path in G.
1 **foreach** S *and* T *in* $\mathcal{M}(G)$ *with* $S \neq T$, $\ell(S) \neq \emptyset$, *and* $\ell(T) \neq \emptyset$ **do**
2 find the (S,T)-path $\mathsf{P} = (S = L_1, L_2, \ldots, L_{h-1}, L_h = T)$;
3 let Q be a queue of the leaves in $C(G)$ and not on P;
4 **while** $Q \neq \emptyset$ **do**
5 pick up a node L in Q;
6 let X be the unique unprocessed neighbor of L;
7 compute Δ_L or Γ_L by Algorithm 1;
8 put X into Q if X has only one unprocessed neighbor and X is not on P;
9 **end**
10 compute Δ'_S or Γ'_S according to L_1;
11 **foreach** $h' = 2, 3, \ldots, h-1$ **do**
12 compute $\Delta'_{L_{h'}}$ or $\Gamma'_{L_{h'}}$ according to $L_{h'+1}$;
13 **end**
14 compute the length of a longest path at T;
15 **end**
16 **return** (The length of a longest path among all longest paths with two fixed endpoints);

Algorithm 4: Longest Path

we cannot use one edge in $\ell(S)$ since it makes a local cycle in S. Second, $\Delta'_S[0] \neq 0$ in general, since S will give an endpoint of a path to L_2 even if L_2 gives no edge to S.

At node $L_{h'}$ in Step 12: We have four subcases.

$L_{h'-1}$ is a parent and $L_{h'+1}$ is a child. In the case, we first modify the computation of $\hat{\Delta}$ in Algorithm 2; the algorithm has to handle the table $\Delta'_{L_{h'-1}}$. However, the only exception is $\hat{\Delta}_{L'_h}[0]$ which should be $\Delta'_{L_{h'-1}}[0]$; the algorithm can connect one vertex in $L_{h'}$ to the current endpoint in $L_{h'-1}$ without using an edge in $L_{h'}$. The other case can be computed by Algorithm 2. Hence, letting $\hat{\Delta}_{L_{h'}}[0] := \Delta'_{L_{h'-1}}[0]$ with Algorithm 2, we have $\hat{\Delta}'_{L_{h'}}$. The table $\hat{\Gamma}_{L_{h'}}$ can be computed by Algorithm 3. The algorithm can use $|\ell(L_{h'})| + c(L_{h'}) - 1$ edges in $L_{h'}$ to inhibit a cycle. Thus, $\Delta'_{L_{h'}}[i]$ is defined by

$$\max_k\{|\ell(L_{h'})| + c(L_{h'}) + \hat{\Gamma}_{L_{h'}}[k] + \hat{\Delta}'_{L_{h'}}[|\ell(L_{h'})| + c(L_{h'}) + k + i\underline{-1}]\}.$$

$L_{h'-1}$ is a child and $L_{h'+1}$ is a parent. This case is symmetric. Thus $\hat{\Delta}_{L_{h'}}$ by Algorithm 2 can be used, and $\hat{\Gamma}'_{L_{h'}}$ is computed by Algorithm 3 with letting $\hat{\Gamma}'_{L_{h'}}[0] := \Gamma'_{L_{h'-1}}[0]$. Using them, $\Gamma'_{L_{h'}}[i]$ is defined by

$$\max_k\{|\ell(L_{h'})| + c(L_{h'}) + \hat{\Gamma}'_{L_{h'}}[k] + \hat{\Delta}_{L_{h'}}[|\ell(L_{h'})| + c(L_{h'}) + k - i\underline{-1}]\}.$$

$L_{h'-1}$ and $L_{h'+1}$ are children. In the case, $L_{h'}$ has to make two endpoints in $L_{h'-1}$ and $L_{h'+1}$. Since no parents are on the (S,T)-path, $\hat{\Delta}_{L_{h'}}$ by Algorithm 2 can be used. The algorithm loses one edge between $L_{h'-1}$ and $L_{h'+1}$. (For example, in Fig. 2 (a), assume that C_1 and C_2 contain two endpoints. Then the algorithm cannot use the edge between them.) The table $\hat{\Gamma}'_{L_{h'}}$ is computed by Algorithm 3 with letting $\hat{\Gamma}'_{L_{h'}}[0] := \Gamma'_{L_{h'-1}}[0]$. Thus $\Delta'_{L_{h'}}[i]$ is defined by

$$\max_k\{|\ell(L_{h'})| + c(L_{h'}) + \hat{\Gamma}'_{L_{h'}}[k] + \hat{\Delta}_{L_{h'}}[|\ell(L_{h'})| + c(L_{h'}) + k + i\underline{-1}]\}.$$

$L_{h'-1}$ and $L_{h'+1}$ are parents. This case is symmetric and hence $\Gamma'_{L_{h'}}[i]$ is defined by

$$\max_k\{|\ell(L_{h'})| + c(L_{h'}) + \hat{\Gamma}_{L_{h'}}[k] + \underline{\hat{\Delta}'_{L_{h'}}}[|\ell(L_{h'})| + c(L_{h'}) + k - i\underline{-1}]\}.$$

At node T in Step 14: By Lemma 6, L_{h-1} is a child of T. The algorithm first computes $\hat{\Delta}$ by Δ_{P_i} for parents of T in the same way of Algorithm 1. Then the algorithm computes $\hat{\Gamma}'_T$ by Algorithm 3 with letting $\hat{\Gamma}'_T[0] := \Gamma'_{L_{h-1}}[0]$. Then the length of a longest path with two endpoints in S and T can be computed by

$$\max_k\{|\ell(T)| + c(T) + \underline{\hat{\Gamma}'_T}[k] + \hat{\Delta}_T[|\ell(T)| + c(T) + k\underline{-1}]\underline{-1}\}.$$

We note that the first "-1" inhibits to make a cycle, and the second "-1" adjusts the number of vertices to the length of a path.

The correctness follows from careful case analysis about the relationship of each node with two endpoints, which is straightforward and hence omitted. The number of choices of S and T is $\binom{n}{2} = O(n^2)$ and it takes $O(n^3)$ time and $O(n^2)$ space for each pair of S and T. Thus the complexity of the algorithm is $O(n^5)$ time and $O(n^2)$ space.

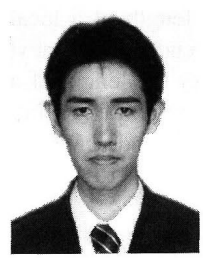
Algorithms 1 and 4 output the length of a longest cycle and a longest path, respectively. However, it is easy to modify them to output cycle and path themselves. Hence we have Theorem 1.

4. Concluding Remarks

We have two future works. First one is improving the complexity of the algorithms. Second one is the longest path problem on a distance-hereditary graph. It is open if this problem can be solved in polynomial time or not. The clique laminar tree of a Ptolemaic graph can be extended in a natural way to a distance-hereditary graph by replacing "clique" by "clique or independent set" [18]. However, the tree for a distance hereditary graph is not uniquely determined. Moreover, in the tree, if a node corresponds to an independent set, we cannot make a cycle that joins the vertices in the node like Fig. 2 (a). Hence it is hard to extend the algorithms in this paper to the tree structure for distance-hereditary graphs.

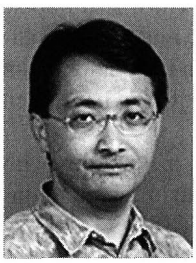
References

[1] M. Garey and D. Johnson, Computers and Intractability — A Guide to the Theory of NP-Completeness, Freeman, 1979.
[2] D. Karger, R. Motwani, and G. Ramkumar, "On approximating the longest path in a graph," Algorithmica, vol.18, pp.82–98, 1997.
[3] H.N. Gabow, "Finding paths and cycles of superpolylogarithmic length," Proc. 36th Symp. on Foundations of Computer Science, pp.407–416, IEEE, 2004.
[4] S. Vishwanathan, "An approximation algorithm for finding a long path in Hamiltonian graphs," Proc. 11th Ann. ACM-SIAM Symp. on Discrete Algorithms, pp.680–685, ACM, 2000.
[5] A. Björklund and T. Husfeldt, "Finding a path of superlogarithmic length," SIAM J. Comput., vol.32, no.6, pp.1395–1402, 2003.
[6] T. Feder and R. Motwani, "Finding large cycles in Hamiltonian graphs," Proc. 16th Ann. ACM-SIAM Symp. on Discrete Algorithms, pp.166–175, ACM, 2005.
[7] R. Bulterman, F. van der Sommen, G. Zwaan, T. Verhoeff, A. van Gasteren, and W. Feijen, "On computing a longest path in a tree," Inf. Process. Lett., vol.81, pp.93–96, 2002.
[8] R. Uehara and Y. Uno, "Efficient algorithms for the longest path problem," 15th Annual International Symposium on Algorithms and Computation (ISAAC 2004), Lecture Notes in Computer Science, vol.3341, pp.871–883, Springer-Verlag, 2004.
[9] H. Ito, Personal communication, 2004.
[10] R. Uehara and Y. Uno, "Laminar structure of ptolemaic graphs and its applications," 16th Annual International Symposium on Algorithms and Computation (ISAAC 2005), Lecture Notes in Computer Science, vol.3827, pp.186–195, Springer-Verlag, 2005.
[11] H. Müller, "Hamiltonian circuit in chordal bipartite graphs," Disc. Math., vol.156, pp.291–298, 1996.
[12] M. Chang, S. Wu, G. Chang, and H. Yeh, "Hamiltonian problems on ptolemaic graphs," Proc. Workshop on Algorithms and Theory of Computation, International Computer Symposium (ICS'00), pp.27–34, Taiwan, 2000.
[13] R.W. Hung, S.C. Wu, and M.S. Chang, "Hamiltonian cycle problem on distance-hereditary graphs," Journal of Information Science and Engineering, vol.19, pp.827–838, 2003.
[14] R.W. Hung and M.S. Chang, "Linear-time algorithms for the Hamiltonian problems on distance-hereditary graphs," Theor. Comput. Sci., vol.341, pp.411–440, 2005.
[15] S.Y. Hsieh, C.W. Ho, T.S. Hsu, and M.T. Ko, "The Hamiltonian problem on distance-hereditary graphs," Discrete Appl. Math., vol.154, pp.508–524, 2006.
[16] H.J. Bandelt and H. Mulder, "Distance-hereditary graphs," Journal of Combinatorial Theory, Series B, vol.41, pp.182–208, 1986.
[17] E. Howorka, "A characterization of ptolemaic graphs," Journal of Graph Theory, vol.5, pp.323–331, 1981.
[18] R. Uehara and Y. Uno, "On the laminar structure of ptolemaic and distance hereditary graphs," Complexity Seminar (Informal seminar in Japanese); available at http://www.jaist.ac.jp/~uehara/pdf/ptolemaic.pdf, March 2005.



Yoshihiro Takahara received the B.E. degree from Okayama University in 2005, and the M.S. degree from Japan Advanced Institute of Science and Technology in 2007. He works at Sekisui House since April, 2007. His main research interests include graph algorithm and their applications.

Sachio Teramoto received M.S. and Ph.D. degrees from School of Information Science, Japan Advanced Institute of Science and Technology (JAIST) in 2003, and 2007, respectively. He works at Service Platforms Research Laboratories, NEC Corporation since April, 2007. His research interests include design and analysis of combinatorial and geometric algorithms.



Ryuhei Uehara received B.E., M.E., and Ph.D. degrees from the University of Electro-Communications, Japan, in 1989, 1991, and 1998, respectively. He was a researcher in CANON Inc. during 1991–1993. In 1993, he joined Tokyo Woman's Christian University as an assistant professor. He was a lecturer during 1998–2001, and an associate professor during 2001–2004 at Komazawa University. He moved to Japan Advanced Institute of Science and Technology (JAIST) in 2004, and he is now an associate professor in School of Information Science. His research interests include computational complexity, algorithms, and data structures, especially, randomized algorithms, approximation algorithms, graph algorithms, and algorithmic graph theory. He is a member of EATCS, ACM, and IEEE.