Identifying Duplications and Lateral Gene Transfers Simultaneously and Rapidly

Zhi-Zhong Chen
Department of Information System Design
Tokyo Denki University
Email: zzchen@mail.dendai.ac.jp

Fei Deng
Department of Computer Science
City University of Hong Kong
Email: fdeng4@student.cityu.edu.hk

Lusheng Wang
Department of Computer Science
City University of Hong Kong
Email: lwang@cs.cityu.edu.hk

***Abstract*—This paper deals with the problem of enumerating all minimum-cost LCA-reconciliations involving gene duplications and lateral gene transfers (LGTs) for a given species tree S and a given gene tree G. Previously, Tofigh *et al.* [20] gave a fixed-parameter algorithm for this problem that runs in $O(m + 3^k n)$ time, where m is the number of vertices in S, n is the number of vertices in G, and k is the minimum cost of an LCA-reconciliation between S and G. In this paper, by refining their algorithm, we obtain a new one for the same problem that finds and outputs the solutions in a compact form within $O(mn^2 + 3^k)$ time.**

I. Introduction

Phylogenetic trees are a commonly used model for representing the evolutionary history of a set of species. In general, a gene tree may not be the same as its underlying species tree in the presence of evolutionary events such as gene duplications, gene losses, and lateral gene transfers (LGTs). The problem of inferring these evolutionary events from a pair of species tree and gene tree has been studied extensively in recent years [15], [3], [11], [5], [20], [21]. Tree reconciliation has been studied as a useful approach to this problem. A reconciliation between a gene tree G and a species tree S is a mapping from the vertices of G to the vertices of S, thus identifying these evolutionary events [20], [9], [7]. In this paper, we are interested in the problem of finding all minimum-cost reconciliations between a gene tree and the underlying species tree taking into account gene duplications and LGTs.

The concept of reconciliations between a gene tree G and a species tree S was first introduced by Goodman *et al.* [7] in which they define a least common ancestor mapping from the vertices of G to the vertices of S. This model was subsequently studied in the literature [18], [19], [14], [11], [9] involving gene duplications and gene losses. Later, as the importance of LGTs becomes widely aware, many algorithms have been proposed to deal with LGTs. Hallett *et al.* [10] extend the concept of reconciliation and use it to infer LGTs. Nakhleh *et al.* [17] give a fast and accurate heuristic algorithm to reconstruct LGTs for a species tree and a set of gene trees. Other efforts for inferring LGTs only include [17], [16], [13], [1], [2].

As for simultaneous identification of these evolutionary events, Górecki [8] gives a model involving duplications, losses and LGTs. However, it requires an extended species tree in which some edges are assumed to be LGTs as its input. An efficient algorithm for this problem is also proposed in [6] in which an additional time stamp function associated with the species tree is needed. Recently, Tofigh *et al.* [20] introduce a formal model to this problem involving gene duplications and LGTs. They give a fixed-parameter algorithm for the problem that outputs the solutions within $O(m+3^k n)$ total time, where m and n are respectively the numbers of vertices in the species tree and the gene tree, and k is the minimum cost over all reconciliations between the input gene tree and the species tree. They also prove that the acyclic version of the problem is NP-hard.

In this paper, we improve the running time of the fixed-parameter algorithm of [20] roughly by a factor of n. In our algorithm, we first perform a preprocessing on the input gene tree and species tree. Then we introduce some techniques which form the foundation of our algorithm on how to find duplications and LGTs. Our algorithm outputs the solutions in a compact form within $O(mn^2 + 3^k)$ total time.

The rest of this paper is organized as follows. Section II gives some basic notations and a formal definition of the problem we study in this paper. Section III gives a brief review of the fixed-parameter algorithm of [20]. Section IV shows several lemmas for our algorithm. Section V presents our improved algorithm.

II. Preliminaries

A. Basic Definitions

Let F be a rooted forest. We use $V(F)$ and $E(F)$ to denote the sets of vertices and edges in F, respectively. F is a *rooted tree* if it has only one root. F is a *rooted binary tree* if it is a rooted tree and the out-degree of every non-leaf vertex in F is 2. Two edges (u_1, v_1) and (u_2, v_2) of F are *siblings* if $u_1 = u_2$.

Let u and v be two vertices of F. For convenience, we view each vertex of F as both an ancestor and a descendant of itself in F. If u is an ancestor (respectively, descendant) of v in F, then we write $u \geq_F v$ (respectively, $v \leq_F u$). If $u \geq_F v$ and $u \neq v$, then u is a *proper* ancestor of v in F (denoted by $u >_F v$). If $u \leq_F v$ and $u \neq v$, then u is a *proper* descendant of v in F (denoted by $u <_F v$). If $u \geq_F v$ or $u \leq_F v$, then u and v are *comparable* in F; otherwise, u and v are *incomparable* in F. The lowest common ancestor of a set U of vertices in F is denoted by $\mathrm{LCA}_F U$. If $w = \mathrm{LCA}_F\{u, v\}$ exists and v is closer to w in F than u, then v is *higher than* u in F (or equivalently, u is *lower than* v in F).

If v has only one child in F, then v is *unifurcate*. If v is a root of F and is unifurcate, then *contracting* v *in* F is

978-1-4673-5875-0/13/$31.00

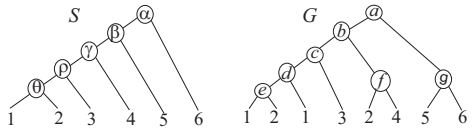

Fig. 1. A species tree S and a gene tree G on the same set {1, 2, 3, 4, 5, 6} of species.

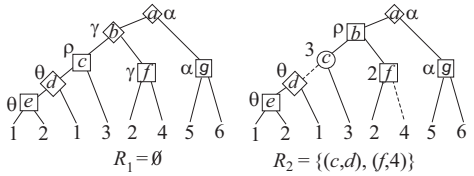

Fig. 2. The LCA mappings of the gene tree G in Figure 1 into the species tree S in Figure 1 associated with the transfer sets $R_1 = \emptyset$ and $R_2 = \{(c, d), (f, 4)\}$, where (1) for each vertex v, the image of v under a mapping is placed near v and (2) speciation vertices and duplication vertices are enclosed by squares and diamonds, respectively.

the operation that modifies F by deleting v. If v is a non-root vertex of F and is unifurcate, then *contracting v in F* is the operation that modifies F by first adding an edge from the parent of v to the child of v and then deleting v.

For a rooted binary tree T and a subset R of $E(T)$, $T \setminus R$ denotes the rooted forest obtained from T by removing the edges in R, while $T \wr R$ denotes the rooted forest obtained from $T \setminus R$ by repeatedly contracting a unifurcate vertex until none exists. Note that each non-leaf vertex of $T \wr R$ has exactly two children in $T \wr R$. For a vertex u of T, T_u denotes the subtree of T rooted at u. Moreover, for two vertices u and v in T with $u >_T v$, $T_u \backslash T_v$ denotes the tree obtained from T_u by removing the edges and vertices in T_v.

B. Transfer Sets and LCA-Reconciliations

Let X be a set of existing species. A *species tree on X* is a rooted binary tree S whose leaves one-to-one correspond to the species in X. A *gene tree on X* is a rooted binary tree G whose leaves (not necessarily one-to-one) correspond to the species in X. Since two leaves of G may correspond to the same species in X, G may have more than $|X|$ leaves. Figure 1 gives an example of S and G.

For a subset R of $E(G)$ containing no sibling edges, the *LCA mapping* of G into S associated with R (denoted by M_R) is defined as follows.

- If u is a leaf of G, then $M_R(u)$ is the unique leaf of S that corresponds to the same species in X as u.
- Otherwise, $M_R(u)$ is $\mathrm{LCA}_S\{M_R(v_1), \ldots, M_R(v_k)\}$, where $v_1, \ldots, v_k$ are the leaf descendants of u in $G \setminus R$.

A *transfer set* w.r.t. (S, G) is a subset R of $E(G)$ such that no two edges of R are siblings and $M_R(u)$ is incomparable to $M_R(v)$ in S for every edge $(u, v) \in R$. As an example, for S and G in Figure 1, two transfer sets R_1 and R_2 w.r.t. (S, G) are shown in Figure 2.

Let R be a transfer set w.r.t. (S, G). We call M_R an *LCA-reconciliation* between G and S. With respect to (w.r.t. for short) R, we classify the non-leaf vertices of G into three types as follows. For each non-leaf vertex u with children v and w in G,

- u is a *transfer vertex* if $(u, v) \in R$ or $(u, w) \in R$;
- u is a *speciation vertex* if it is not a transfer vertex and $M_R(v)$ and $M_R(w)$ are incomparable in S;
- u is a *duplication vertex* if it is not a transfer vertex and $M_R(v)$ and $M_R(w)$ are comparable in S.

We use $\mathrm{DV}(R)$ to denote the set of all duplication vertices w.r.t. R. For example, in Figure 2, $\mathrm{DV}(R_1) = \{a, b, d\}$ and $\mathrm{DV}(R_2) = \{a, d\}$.

The *cost* of R (denoted by $\mathrm{Cost}(R)$) is $|\mathrm{DV}(R)| + |R|$. For example, the costs of the transfer sets R_1 and R_2 in Figure 2 are 3 and 4, respectively.

In this paper, we want to solve the following problem (called the *transfer set enumeration (TSE) problem*):

- **Input:** A pair (S, G), where S is a species tree on a set X of species and G is a gene tree on X.
- **Output:** All minimum-cost transfer sets R w.r.t. (S, G).

For a nonnegative integer k, a *k-transfer set* w.r.t. (S, G) is a transfer set R w.r.t. (S, G) such that $\mathrm{Cost}(R) = k$. To enumerate all minimum-cost transfer sets w.r.t. (S, G), we can proceed as follows:

1) Initialize $k = 0$.
2) Enumerate all k-transfer sets w.r.t. (S, G).
3) If at least one k-transfer set is found in Step 2, then stop; otherwise, increase k by 1 and goto Step 2.

So, our problem has become how to perform Step 2, i.e., how to solve the following problem (called the *bounded transfer set enumeration (BTSE) problem*):

- **Input:** A triple (S, G, k), where S is a species tree on a set X of species, G is a gene tree on X, and k is an integer such that there is no k'-transfer set w.r.t. (S, G) for all nonnegative integers $k' < k$.
- **Output:** All k-transfer sets w.r.t. (S, G).

III. Algorithm for the BTSE Problem

Throughout this section, fix an input (S, G, k) to the BTSE problem.

A. Candidates

For a subset U of $V(G)$, $E[U]$ denotes the set of all edges (u, v) in G with $u \in U$. A *candidate* w.r.t. (S, G) is a triple (R, D, Σ) satisfying the following conditions:

1) R is a transfer set w.r.t. (S, G).
2) D and Σ are disjoint sets of vertices in G.
3) For each $u \in D \cup \Sigma$, $R \cap E[\{u\}] = \emptyset$.
4) For each $u \in \Sigma$, u is a speciation vertex w.r.t. R, u is not a root of $G \wr R$, and the parent of u in $G \wr R$ belongs to D.
5) For each $v \in D$, $G \wr R$ contains a directed path Q from v to a proper descendant u such that (1) $u \in \Sigma$

or u is a leaf, (2) all vertices of Q except u belong to D, and (3) $M_R(w) = M_R(u)$ for every vertex w of Q.

Throughout this subsection, let $C = (R, D, \Sigma)$ be a candidate w.r.t. (S, G). If a vertex u of G is a leaf vertex of G, belongs to $D \cup \Sigma$, or is incident to an edge in R, then u is *marked* w.r.t. C; otherwise, it is unmarked w.r.t. C. A vertex v of G is *settled* w.r.t. C if each descendant of v in G is marked w.r.t. C or a speciation vertex w.r.t. R. C is *final* if all vertices of G are settled w.r.t. C. The next lemmas have been implicitly proved in [20]) (for precise proofs, see [4]):

Lemma 1: Let R^* be a transfer set w.r.t. (S, G) such that $R \subseteq R^*$ and $E[D \cup \Sigma] \cap R^* = \emptyset$. Then, for each $u \in \Sigma$, $M_R(u) = M_{R^*}(u)$ and u is a speciation vertex w.r.t. M_{R^*}. Moreover, for each $v \in D$, $M_R(v) = M_{R^*}(v)$ and v is a duplication vertex w.r.t. M_{R^*}.

Lemma 2: If C is final, then there is no transfer set R^* w.r.t. (S, G) such that $R \subseteq R^*$ and $\mathrm{Cost}(R^*) < \mathrm{Cost}(R)$.

B. Outline of the Algorithm

To enumerate all k-transfer sets w.r.t. (S, G), we start with the candidate $C = (R, D, \Sigma) = (\emptyset, \emptyset, \emptyset)$, and then try all possible ways to extend it by *gradually* adding edges to R and adding vertices to Σ and D. By Lemma 1, $|R|+|D|$ is a lower bound on $\min_{R^*} \mathrm{Cost}(R^*)$, where R^* ranges over all transfer sets w.r.t. (S, G) such that $R \subseteq R^*$ and $E[\Sigma \cup D] \cap R^* = \emptyset$. Initially, the lower bound is 0 because R and D are empty. However, gradually extending C will gradually increase the lower bound. Once the lower bound becomes $> k$, we can stop extending C. By Lemma 2, we can also stop extending C once it becomes final.

C. Extending a Candidate

Throughout this subsection, fix a non-final candidate $C = (R, D, \Sigma)$ with $|R| + |D| < k$. To extend C, the idea is to make several types of *moves* as defined as follows. A *d-move* w.r.t. C is an unmarked vertex u w.r.t. C such that

- u has a child $v \in D \cup \Sigma$ in $G \diamond R$ with $M_R(u) = M_R(v)$, or
- all the leaf descendants of u in $G \diamond R$ correspond to the same species.

Eliminating a d-move u w.r.t. C means modifying C by adding u to D.

An *s-move* w.r.t. C is an unmarked speciation vertex u w.r.t. C such that u is not a root of $G \diamond R$, the parent p of u in $G \diamond R$ does not belong to $D \cup \Sigma$, and $M_R(p) = M_R(u)$. *Eliminating an s-move* u w.r.t. C means modifying C in one of the following possible ways: (1) adding edge (u, v) to R; (2) adding edge (u, w) to R; (3) adding p to D and u to Σ, where v and w are the children of u in G and p is the parent of u in $G \diamond R$.

A *move* w.r.t. C is a d- or s-move w.r.t. C. The next three lemmas have been implicitly proved in [20] (for precise proofs, see [4]):

Lemma 3: Suppose that u is a d-move w.r.t. C. Let R^* be a transfer set w.r.t. (S, G) such that $R \subseteq R^*$ and $R^* \cap E[D \cup \Sigma] = \emptyset$. Then, R^* contains no edge in $E[\{u\}]$ and $(R, D \cup \{u\}, \Sigma)$ is a candidate w.r.t. (S, G).

Input: A candidate $C = (R, D, \Sigma)$ w.r.t. (S, G).
Output: All final candidates (R', D', Σ') w.r.t. (S, G) such that $|R'| + |D'| \leq k$, $R \subseteq R'$, $D \subseteq D'$, and $\Sigma \subseteq \Sigma'$.

1. If either $|R| + |D| > k$, or $|R| + |D| = k$ and at least one vertex of G is not settled w.r.t. C, then return.
2. If no move w.r.t. C exists in G, then output R and return.
3. Find a move μ w.r.t. C and proceed as follows.

3.1. If μ is a d-move w.r.t. C, recursively call *k-AllTransSet* on input $(R, D \cup \{\mu\}, \Sigma)$ and then return.

3.2. If μ is an s-move, recursively call *k-AllTransSet* on input $(R \cup \{(\mu, \mu')\}, D, \Sigma)$, $(R \cup \{(\mu, \mu'')\}, D, \Sigma)$, and $(R, D \cup \{p\}, \Sigma \cup \{\mu\})$ in any order and then return, where μ' and μ'' are the children of μ in G and p is the parent of μ in $G \diamond R$.

Fig. 3. The outline of *k-AllTransSet*

Lemma 4: Suppose that u is an s-move w.r.t. C. Let v and w be the children of u in G, p be the parent of u in $G \diamond R$, and R^* be a transfer set w.r.t. (S, G) such that $R \subseteq R^*$ and $R^* \cap E[D \cup \Sigma] = \emptyset$. Then, the following statements hold:

1) Either $E[\{p, u\}] \cap R^* = \emptyset$ or $E[\{u\}] \cap R^*$ is equal to $\{(u, v)\}$ or $\{(u, w)\}$.
2) $(R \cup \{(u, v)\}, D, \Sigma)$, $(R \cup \{(u, w)\}, D, \Sigma)$, and $(R, D \cup \{p\}, \Sigma \cup \{u\})$ are candidates w.r.t. (S, G).

Lemma 5: Suppose that at least one vertex of G is not settled w.r.t. (R, U). Then, there is a move w.r.t. C.

Based on Lemmas 3, 4, and 5, we are now ready to describe a recursive subroutine (called *k-AllTransSet*) for extending a given candidate (R, D, Σ). It is depicted in Figure 3. To enumerate all k-transfer sets w.r.t (S, G), it suffices to call *k-AllTransSet* on input $(\emptyset, \emptyset, \emptyset)$.

Note that to find a move in Step 3, *k-AllTransSet* needs to scan the vertices of G until it finds a move w.r.t. C or finds that no moves w.r.t. C exist in G. Thus, it can take $O(n)$ time to find a single move w.r.t. C, where $n = |V(G)|$. So, *k-AllTransSet* takes $O(m + 3^k n)$ on input $(\emptyset, \emptyset, \emptyset)$, where $m = |V(S)|$.

IV. Useful Properties of Moves

Throughout this section, fix an input (S, G, k) to the BTSE problem and also fix a non-final candidate $C = (R, D, \Sigma)$ with $|R| + |D| < k$. We state three lemmas based on which our new algorithm will be designed. For lack of space, their proofs are given in Appendix A.

Lemma 6: Suppose that u_1 is a d-move w.r.t. C. For the candidate $C' = (R, D \cup \{u_1\}, \Sigma)$, the following statements hold:

1) Each d-move v w.r.t. C with $v \neq u_1$ is also a d-move w.r.t. C'.
2) Each d-move w.r.t. C' is either a d-move w.r.t. C or the parent of u_1 in $G \diamond R$.
3) Neither child of u_1 in $G \diamond R$ is an s-move w.r.t. C'.

4) Each s-move w.r.t. C that is not a child of u_1 in $G \setminus R$ is also an s-move w.r.t. C'.
5) Each s-move w.r.t. C' is also an s-move w.r.t. C.

Lemma 7: Suppose that u_1 is an s-move w.r.t. C. Let p_1 be the parent of u_1 in $G \setminus R$ and u'_1 be the sibling of u_1 in $G \setminus R$. For the candidate $C' = (R, D \cup \{p_1\}, \Sigma \cup \{u_1\})$, the following statements hold:

1) Each d-move v w.r.t. C with $v \neq p_1$ is also a d-move w.r.t. C'.
2) Each d-move w.r.t. C' is either a d-move w.r.t. C or the parent of p_1 in $G \setminus R$.
3) u'_1 is not an s-move w.r.t. C'.
4) Each s-move u_2 w.r.t. C with $u_2 \notin \{u_1, u'_1\}$ is also an s-move w.r.t. C'.
5) Each s-move w.r.t. C' is also an s-move w.r.t. C.

Lemma 8: Assume that u_1 is an s-move w.r.t. C, its children in G are v_1 and w_1, and its children in $G \setminus R$ are $\tilde{v}_1$ and $\tilde{w}_1$, where $v_1 \geq_G \tilde{v}_1$ and $w_1 \geq_G \tilde{w}_1$. For the candidate $C' = (R \cup \{(u_1, v_1)\}, D, \Sigma)$ w.r.t. (S, G), the following hold:

1) Every d-move w.r.t. C is a d-move w.r.t. C'.
2) If u_2 is a d-move w.r.t. C' but not a d-move w.r.t. C, then u_2 is a proper ancestor of u_1 in $G \setminus R$.
3) Every s-move w.r.t. C other than u_1 is still an s-move w.r.t. C'.
4) If u_2 is an s-move w.r.t. C' but not an s-move w.r.t. C, then u_2 is $\tilde{w}_1$ or a child of a proper ancestor of u_1 in $G \setminus R$.

V. The New Algorithm

From Section III, we know that the bottleneck of the algorithm in Section III is in finding a move w.r.t. the current candidate (R, D, Σ). To speed up the algorithm, our idea is to perform a preprocessing on the input trees G and S so that we can find a move w.r.t. the current candidate in constant amortized time.

A. The Preprocessing

It is known [12] that we can process a given tree T in linear time so that given two vertices u and v of T, we can find $\mathrm{LCA}_T\{u, v\}$ in $O(1)$ time. So, as in the algorithm in [20], we first perform a linear-time preprocessing on S (respectively, G) so that given two vertices u and v of S (respectively, G), we can find $\mathrm{LCA}_S\{u, v\}$ (respectively, $\mathrm{LCA}_G\{u, v\}$) in constant time. Then, in $O(n)$ total time, we can compute $M_\emptyset(u)$ for all vertices u of G. Once knowing $M_\emptyset$, we can find all d-moves (respectively, s-moves) w.r.t. the empty candidate $(\emptyset, \emptyset, \emptyset)$ in $O(n)$ time. We refer to them as the *initial d-moves* (respectively, *initial s-moves*) in G.

An *initial move* in G is an initial d- or s-move in G. By switching the left and the right subtrees of a vertex in G when necessary, we can assume that G satisfies the following condition:

C1. For every non-leaf vertex u of G, if the right child of u in G has a descendant that is an initial move, then so does the left child of u in G.

A vertex w of G is a *junction* if there are two distinct initial moves u and v in G such that $\mathrm{LCA}_G\{u, v\} = w$ and u is incomparable with v in G.

We also perform a postorder traversal of G in $O(n)$ time. A vertex u is *smaller than* another vertex v of G (denoted by $u < v$) if the postorder number of u is smaller than that of v. To each junction w of G, we associate a pair $(\mu_\ell(w), \mu_r(w))$ of initial moves such that $\mu_\ell(w)$ is the smallest initial move in G_u and $\mu_r(w)$ is the smallest initial move in G_v, where u and v are the left and the right children of w in G, respectively. Obviously, the pairs $(\mu_\ell(w), \mu_r(w))$ for all junctions w of G can be computed in $O(n)$ total time.

A *critical ancestral junction* of a vertex u in G is a junction w in G with $w >_G u$ such that if u is a descendant of the left (respectively, right) child of w in G, then $\mu_r(w)$ (respectively, $\mu_\ell(w)$) is not a child of w in G. For each non-leaf vertex u of G such that u has a critical ancestral junction in G, we associate u with the smallest critical ancestral junction $\vartheta(u)$. For convenience, if u is a non-leaf vertex of G with no critical ancestral junction in G, we let $\vartheta(u)$ be undefined. Obviously, $\vartheta(u)$ for all non-leaf vertices u of G can be computed in $O(n)$ total time.

Example 1: Let S and G be as in Figure 1. Obviously, there is no initial d-move in G, the initial s-moves in G are e, f, and g, and the junctions in G are a and b. Moreover, the pair (e, f) of initial s-moves is associated with b, while the pair (e, g) of initial s-moves is associated with a. Furthermore, $\vartheta(f) = b$, $\vartheta(g) = a$, and $\vartheta(a)$ through $\vartheta(e)$ are undefined.

We next define a function $\varphi_1 : V(G) \times V(S) \times \{0, 1\} \to V(G)$ as follows. Consider an arbitrary triple (u_0, α_0, b) such that $u_0 \in V(G)$, $\alpha_0 \in V(S)$, and $b \in \{0, 1\}$. Let $u_1, u_2, \ldots, u_h$ be the ancestors of u_0 in G, where $u_i >_G u_{i-1}$ for all $1 \leq i \leq h$. For each $1 \leq i \leq h$, let v_i be the child of u_i that is not an ancestor of u_0 in G. Imagine the situation where we have a candidate $C = (R, D, \Sigma)$ such that $M_R(u_0) = \alpha_0$, all of the edges in R and the vertices in $D \cup \Sigma$ appear in G_{u_0}, and $b = 0$ if and only if $u_0 \notin D$. If one or more vertices among $u_1, \ldots, u_h, v_1, \ldots, v_h$ are moves w.r.t. C, then $\varphi_1(u_0, \alpha_0, b)$ is the smallest one among such moves; otherwise, $\varphi_1(u_0, \alpha_0, b)$ is undefined. Even without knowing C exactly, we can compute $\varphi_1(u_0, \alpha_0, b)$ from u_0 and α_0 in $O(n)$ time (for lack of space, we omit the details). So, the function φ_1 can be computed in $O(n^2 m)$ total time.

We further define a function $\varphi_2 : V(G) \times V(G) \times V(S) \to V(S)$ as follows. Consider an arbitrary triple (u, v, α) such that $\{u, v\} \subseteq V(G)$, $u >_G v$, and $\alpha \in V(S)$. Imagine the situation where we have a transfer set R such that $M_R(v) = \alpha$ and each edge $(x, y) \in R$ with $u \geq_G x$ also satisfies that $v \geq_G x$. Then, $\varphi_2(u, v, \alpha) = M_R(u)$. Note that even without knowing R, we can compute $\varphi_2(u, v, \alpha)$ from u, v, and α as follows. Let $v_1, \ldots, v_h$ be the siblings of those ancestors w of v in G with $w <_G u$. Then, $\varphi_2(u, v, \alpha) = \mathrm{LCA}_S\{\alpha, M_\emptyset(v_1), \ldots, M_\emptyset(v_h)\}$. Obviously, we can compute $\varphi_2(u, v, \alpha)$ in $O(n)$ time. So, the function φ_2 can be computed in $O(n^3 m)$ total time. Indeed, we can improve the complexity to $O(n^2 m)$ time. To see this, it suffices to observe that for each pair (v, α) with $v \in V(G)$ and $\alpha \in V(S)$, we can compute $\varphi_2(u_1, v, \alpha)$, $\ldots$, $\varphi_2(u_b, v, \alpha)$ in $O(n)$ total time, where $u_1, \ldots, u_b$ are the proper ancestors of v in G.

Example 2: Let S and G be as in Figure 1. Then, $\varphi_1(b, \rho, 0) = g$ and $\varphi_2(a, b, \rho) = \alpha$. Moreover, $\varphi_1(d, 1, 1) = f$, $\varphi_1(e, 2, 0) = f$, $\varphi_2(d, e, 2) = \theta$, $\varphi_2(c, e, 2) = \rho$,

$\varphi_2(b, e, 2) = \gamma$ and $\varphi_2(a, e, 2) = \alpha$.

B. Finding and Eliminating Moves

Let $C = (R, D, \Sigma)$ be a candidate w.r.t. (S, G). A move v w.r.t. C is *extreme* if no proper descendant of v in G is a move w.r.t. C.

After the preprocessing on G and S, our algorithm starts with the empty candidate $C_0 = (R_0, D_0, \Sigma_0) = (\emptyset, \emptyset, \emptyset)$, and then proceeds to eliminate an extreme move μ_0 w.r.t. C_0 if there is any. If μ_0 is a d-move w.r.t. C_0, then there is only one way to eliminate it; otherwise, there are three ways to eliminate μ_0 and so we need to try all of them (by making three recursive calls of the algorithm in any order). More specifically, if μ_0 is a d-move w.r.t. C_0, then eliminating μ_0 requires modifying C_0 by adding μ_0 to D_0; otherwise, eliminating μ_0 requires modifying C_0 by either adding an edge to R_0 or adding a vertex to D_0 and another to Σ_0. In any case, we use C_1 to denote the modified C_0. In general, once we obtain a candidate $C_i = (R_i, D_i, \Sigma_i)$ $(i \geq 0)$, we then try to find a move w.r.t. C_i. If no move w.r.t. C_i exists, then we are done. Otherwise, we eliminate a move w.r.t. C_i to obtain a new candidate $C_{i+1} = (R_{i+1}, D_{i+1}, \Sigma_{i+1})$. The main difficulty is how to find a move w.r.t. C_{i+1} once we obtain C_{i+1}. We detail how to do this in constant amortized time below.

In the remainder of this section, for each integer $i \geq 0$, the phrase "at time i" means the time point immediately before eliminating a move μ_i w.r.t. C_i. During the execution of the algorithm, we always maintain the following invariant:

I1. For every integer $i \geq 0$, μ_i is an extreme move w.r.t. C_i and we know $M_{R_i}(\mu_i)$ at time i.

Since we find all initial moves in the preprocessing on G and S, we can assume that the smallest initial move μ_0 is available for free. To give the reader a glimpse how our algorithm works, we assume that μ_0 is an s-move w.r.t. C_0, and detail what will happen after eliminating μ_0. Let u_0, u_1, $\ldots$, u_h be the ancestors of μ_0 in G with $u_i >_G u_{i-1}$ for all $i \in \{1, \ldots, h\}$. Note that $u_0 = \mu_0$ and u_h is the root of G. Moreover, by Condition C1, u_{i-1} is the left child of u_i in G for each $1 \leq i \leq h$. For each $i \in \{1, \ldots, h\}$, let v_i be the right child of u_i in G. Recall that we have three ways to eliminate μ_0. The first way is to add u_1 to D and μ_0 to Σ, the second way is to add edge (μ_0, v_0) to R, and the third way is to add edge (μ_0, w_0) to R, where v_0 and w_0 are the children of μ_0 in G. So, $C_1 = (\emptyset, \{u_1\}, \{\mu_0\})$, $C_1 = (\{(\mu_0, v_0)\}, \emptyset, \emptyset)$, or $C_1 = (\{(\mu_0, w_0)\}, \emptyset, \emptyset)$.

Case 1: $C_1 = (\emptyset, \{u_1\}, \{\mu_0\})$. In this case, we proceed based on Lemma 7. Depending on whether u_1 is a junction in G, we distinguish two subcases as follows.

Case 1.1: u_1 is not a junction in G. In this case, u_1 is settled w.r.t. C_1. We then check if $\vartheta(u_1)$ and $\varphi_1(u_1, \alpha, 1)$ are defined or not, where $\alpha = M_\emptyset(u_1)$. If both are undefined, then there is no move w.r.t. C_1. So, suppose that $\vartheta(u_1)$ or $\varphi_1(u_1, \alpha, 1)$ is defined. If $\varphi_1(u_1, \alpha, 1)$ is defined and $\vartheta(u_1)$ is either undefined or defined but $\mathrm{LCA}_G(u_1, \varphi_1(u_1, \alpha, 1)) <_G \vartheta(u_1)$, then $\varphi_1(u_1, \alpha, 1)$ is an extreme move w.r.t. C_1; otherwise, $\mu_r(\vartheta(u_1))$ is an extreme move w.r.t. C'.

Case 1.2: u_1 is a junction in G. In this case, if either (1) $\mu_r(u_1) = v_1$ and v_1 is a d-move w.r.t. C_1 or (2) $\mu_r(u_1) <_G v_1$, then $\mu_r(u_1)$ is an extreme move w.r.t. C_1. Otherwise, u_1 is settled w.r.t. C_1 and we proceed as in Case 1.1.

Case 2: $C_1 = (\{(\mu_0, v_0)\}, \emptyset, \emptyset)$. In this case, we proceed based on Lemma 8. Obviously, if w_0 is an s-move w.r.t. C_1, then it is also an extreme move w.r.t. C_1. Otherwise, we check if $\vartheta(\mu_0)$ and $\varphi_1(\mu_0, \alpha_0, 0)$ are defined or not, where $\alpha_0 = M_{R_1}(\mu_0) = M_\emptyset(w_0)$.

Case 2.1: Both $\varphi_1(\mu_0, \alpha_0, 0)$ and $\vartheta(\mu_0)$ are undefined. In this case, there is no move w.r.t. C_1.

Case 2.2: $\varphi_1(\mu_0, \alpha_0, 0)$ is defined but $\vartheta(\mu_0)$ is not. In this case, $\varphi_1(\mu_0, \alpha_0, 0)$ is an extreme move w.r.t. C_1.

Case 2.3: $\vartheta(\mu_0)$ is defined but $\varphi_1(\mu_0, \alpha_0, 0)$ is not. In this case, $\mu_r(\vartheta(\mu_0))$ is an extreme move w.r.t. C_1.

Case 2.4: Both $\varphi_1(\mu_0, \alpha_0, 0)$ and $\vartheta(\mu_0)$ are defined. In this case, if $\vartheta(u_0) >_G \mathrm{LCA}_G(u_0, \varphi_1(u_0, \alpha_0, 0))$, then $\varphi_1(\mu_0, \alpha_0, 0)$ is an extreme move w.r.t. C_1. Otherwise, $\mu_r(\vartheta(\mu_0))$ is an extreme move w.r.t. C_1.

Case 3: $C_2 = (\{(\mu_0, w_0)\}, \emptyset, \emptyset)$. This case is similar to Case 2.

In the above, we have seen how to find an extreme move w.r.t. C_1 in constant time after eliminating an extreme s-move w.r..t C_0. In general, for $i \geq 1$, we need to find an extreme move μ_i w.r.t. C_i after eliminating an extreme move μ_{i-1} w.r.t. C_{i-1}. When $i = 1$, this is easy to do because even if $R_1 \neq R_0$, it is easy to compute $\alpha_0 = M_{R_1}(\mu_0)$ and $\varphi_1(\mu_0, \alpha_0, 0)$ can be used (to find μ_1) as it is (in the sense that all of the edges in R_1 and the vertices in $D_1 \cup \Sigma_1$ appear in G_{μ_0}). However, when $i \geq 2$, it may happen that μ_{i-1} has a proper ancestor w in G such that some edges in R_{i-1} or some vertices in $D_{i-1} \cup \Sigma_{i-1}$ appear in G_x, where x is the child of w in G that is not an ancestor of μ_{i-1} in G. If this really happens, $\varphi_1(\mu_{i-1}, \alpha_{i-1}, 0)$ and $\varphi_1(\mu_{i-1}, \alpha_{i-1}, 1)$ cannot be necessarily used (to find μ_i) as it is, where $\alpha_{i-1} = M_{R_i}(\mu_{i-1})$. To overcome this difficulty, our idea is to use a stack to keep track of such vertices w. The details of finding an extreme move μ_i w.r.t. C_i after eliminating an extreme move μ_{i-1} w.r.t. C_{i-1} are very lengthy, we omit the details for the lack of space.

C. A Sped-up Version of k-AllTransSet

Even if we can find the next move (to be eliminated) in amortized constant time, it is still unclear that *k-AllTransSet* can run in $O(3^k)$ time, because (1) we need to always memorize the current candidate $C = (R, D, \Sigma)$, $|R| + |D|$, $G \setminus R$, the move μ w.r.t. C to be eliminated next, the content of the stack, and so on, and (2) updating them can be expensive. Nonetheless, we can design a sped-up version (called *k-AllTransSet2*) of *k-AllTransSet* so that the following lemma holds (for details, see Appendix B):

Lemma 9: Subroutine *k-AllTransSet2* finds and outputs all optimal transfer sets w.r.t. (S, G) in $O(3^k + \#_{sol} \cdot k)$ time or in $O(3^k)$ time but in a compact form, where $\#_{sol}$ is the number of k-transfer set w.r.t. (S, G).

Input: A species tree S and a gene tree G on the same set of species.
Output: All optimal transfer sets w.r.t. (S, G).
1. *Preprocessing step:* Perform the preprocessing as described in Section V-A.
2. Initialize $k = 0$.
3. While no transfer set w.r.t. (S, G) has been outputted, perform the following:
3.1. Initialize the global variables defined in Section V-C for S, G and k.
3.2. Call the subroutine *k-AllTransSet2*.
3.3. Increase k by 1.

Fig. 4. The algorithm for enumerating all optimal transfer sets

D. The Algorithm for Enumerating Optimal Transfer Sets

We are now ready to present the new algorithm for enumerating all optimal transfer sets w.r.t. a given pair (S, G) of a species tree and a gene tree. It is detailed in Figure 4.

Theorem 10: Given a pair (S, G) of a species tree and a gene tree on the same set of species, we can find and output all optimal transfer sets w.r.t. (S, G) in $O(mn^2+3^k+\#_{sol}\cdot k)$ time or in $O(mn^2 + 3^k)$ time but in a compact form, where k is the cost of an optimal transfer set w.r.t. (S, G), $\#_{sol}$ is the number of k-transfer set w.r.t. (S, G), and m and n are the number of vertices in S and G, respectively.

Proof: It suffices to show that the algorithm in Figure 4 runs in $O(mn^2 + 3^k + \#_{sol} \cdot k)$ time. To this end, first note that by the discussion in Section V-A, the preprocessing step takes $O(n^2m)$ time. Moreover, the time needed for outputting the optimal transfer sets w.r.t. (S, G) is $O(\#_{sol} \cdot k)$. So, we hereafter ignore the time needed for the preprocessing and for outputting the optimal transfer sets. By Lemma 17, the algorithm takes $O(\sum_{i=0}^{k} 3^i) = O(3^k)$ time. Q.E.D. ■

REFERENCES

[1] L. Addario-Berry, M. Hallett, and J. Lagergren, "Towards identifying lateral gene transfer events," in *Proc. 8th Pacific Symp. on Biocomputing (PSB03)*. Citeseer, 2003, pp. 279–290.

[2] R. Beiko and N. Hamilton, "Phylogenetic identification of lateral genetic transfer events," *BMC evolutionary biology*, vol. 6, no. 1, p. 15, 2006.

[3] M. Charleston, "Jungles: a new solution to the host/parasite phylogeny reconciliation problem," *Mathematical Biosciences*, vol. 149, no. 2, pp. 191–223, 1998.

[4] Z.-Z. Chen, F. Deng, and L. Wang, "Simultaneous identification of duplications, losses, and lateral gene transfers," *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, pp. 1515–1528, 2012.

[5] M. Csűrös and I. Miklós, "A probabilistic model for gene content evolution with duplication, loss, and horizontal transfer," in *Research in computational molecular biology*. Springer, 2006, pp. 206–220.

[6] J. Doyon, C. Scornavacca, K. Gorbunov, G. Szöllősi, V. Ranwez, and V. Berry, "An efficient algorithm for gene/species trees parsimonious reconciliation with losses, duplications and transfers," *Comparative Genomics*, pp. 93–108, 2011.

[7] M. Goodman, J. Czelusniak, G. Moore, A. Romero-Herrera, and G. Matsuda, "Fitting the gene lineage into its species lineage, a parsimony strategy illustrated by cladograms constructed from globin sequences," *Systematic Zoology*, pp. 132–163, 1979.

[8] P. Górecki, "Reconciliation problems for duplication, loss and horizontal gene transfer," in *Proceedings of the eighth annual international conference on Resaerch in computational molecular biology*. ACM, 2004, pp. 316–325.

[9] R. Guigo, I. Muchnik, and T. Smith, "Reconstruction of ancient molecular phylogeny," *Molecular Phylogenetics and Evolution*, vol. 6, no. 2, pp. 189–213, 1996.

[10] M. Hallett and J. Lagergren, "Efficient algorithms for lateral gene transfer problems," in *Proceedings of the fifth annual international conference on Computational biology*. ACM, 2001, pp. 149–156.

[11] M. Hallett and J. Lagergren, "New algorithms for the duplication-loss model," in *Proceedings of the fourth annual international conference on Computational molecular biology*. ACM, 2000, pp. 138–146.

[12] D. Harel and R. Tarjan, "Fast algorithms for finding nearest common ancestors," *SIAM Journal on Computing*, vol. 13, no. 2, pp. 338–355, 1984.

[13] G. Jin, L. Nakhleh, S. Snir, and T. Tuller, "Maximum likelihood of phylogenetic networks," *Bioinformatics*, vol. 22, no. 21, p. 2604, 2006.

[14] B. Ma, M. Li, and L. Zhang, "From gene trees to species trees," *SIAM Journal on Computing*, vol. 30, no. 3, pp. 729–752, 2000.

[15] W. Maddison, "Gene trees in species trees," *Systematic biology*, vol. 46, no. 3, p. 523, 1997.

[16] G. Manolo and D. Vincent, "Detecting lateral gene transfers by statistical reconciliation of phylogenetic forests," *BMC Bioinformatics*, vol. 11.

[17] L. Nakhleh, D. Ruths, and L. Wang, "Riata-hgt: a fast and accurate heuristic for reconstructing horizontal gene transfer," *Computing and Combinatorics*, pp. 84–93, 2005.

[18] R. Page, "Maps between trees and cladistic analysis of historical associations among genes, organisms, and areas," *Systematic Biology*, vol. 43, no. 1, p. 58, 1994.

[19] R. Page and M. Charleston, "From gene to organismal phylogeny: Reconciled trees and the gene tree/species tree problem," *Molecular Phylogenetics and Evolution*, vol. 7, no. 2, pp. 231–240, 1997.

[20] A. Tofigh, M. Hallett, and J. Lagergren, "Simultaneous identification of duplications and lateral gene transfers," *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, pp. 517–535, 2011.

[21] L. Zhang, Y. Ng, T. Wu, and Y. Zheng, "Network model and efficient method for detecting relative duplications or horizontal gene transfers," *Proc. IEEE First Int'l Conf. in Computational Advances in Bio and Medical Sciences (ISSABS)*, pp. 214-219, 2011.

Appendix A: Omitted Proofs

The next lemma has been implicitly proved in [20] (for a precise proof, see [4]):

Lemma 11: Suppose that u is a speciation vertex w.r.t. R. Let R^* be a transfer set w.r.t. (S, G) such that $R \subseteq R^*$ and $E[\{u\}] \cap R^* = \emptyset$. Then, $M_R(u) = M_{R^*}(u)$. Moreover, u is also a speciation vertex w.r.t. R^*.

Lemma 12: Suppose that u is a d-move w.r.t. a candidate $C = (R, D, \Sigma)$ and $C' = (R', D', \Sigma')$ is another candidate such that $R \subseteq R'$, $D \subseteq D'$, and $\Sigma \subseteq \Sigma'$. Then, u remains to be a d-move w.r.t. C' if and only if $u \notin D'$.

Proof: The "only-if" part is obvious. To prove the "if" part, assume that $u \notin D'$. Since u is a d-move w.r.t. C, u has a child v in $G \setminus R$ such that $M_R(u) = M_R(v)$ and either $v \in D \cup \Sigma$ or all the descendants of v in $G \setminus R$ correspond to the same species. In either case, $M_{R'}(v) = M_R(v)$ and $M_R(v) \geq_S M_R(w)$, where w is the other child of u in $G \setminus R$. Moreover, since $R \subseteq R'$, $M_R(w) \geq_S M_{R'}(w)$. So, $M_{R'}(v) \geq_S M_{R'}(w)$ and in turn $M_{R'}(u) = M_{R'}(v)$. Consequently, neither edge incident to u can belong to R' because R' is a transfer set. Moreover, $u \notin \Sigma'$ because u is not a speciation vertex w.r.t. R'. Now, since $u \notin D'$, u is a d-move w.r.t. C'. Q.E.D. ■

Lemma 13: Suppose that u is an s-move w.r.t. a candidate $C = (R, D, \Sigma)$ and $C' = (R', D', \Sigma')$ is another candidate such that $R \subseteq R'$, $D \subseteq D'$, and $\Sigma \subseteq \Sigma'$. Then, u remains to be an s-move w.r.t. C' if and only if neither does the parent

of u in $G \setminus R'$ belong to $D' \cup \Sigma'$ nor is u incident to an edge in R'.

Proof: The "only-if" part is obvious. To prove the "if" part, assume that neither does the parent p' of u in $G \setminus R'$ belong to $D' \cup \Sigma'$ nor is u incident to an edge in R'. Let p and v be the parent and the sibling of u in $G \setminus R$, respectively. Since u is an s-move w.r.t. C, $M_R(u) \geq_S M_R(v)$. Moreover, since u is a speciation vertex w.r.t. R, Lemma 11 implies that $M_{R'}(u) = M_R(u)$. Furthermore, since $R \subseteq R'$, $M_R(v) \geq_S M_{R'}(v)$. So, $M_{R'}(u) \geq_S M_{R'}(v)$ and in turn $M_{R'}(u) = M_{R'}(p)$. Thus, neither edge incident to p in G belongs to R' because R' is a transfer set. Hence, $p' = p$. In addition, $u \notin \Sigma'$ because C' is a candidate and $p' \notin D'$. Moreover, $u \notin D'$ because u is a speciation vertex w.r.t. R' by Lemma 11. Therefore, u is still an s-move w.r.t. C'. Q.E.D. ■

Lemma 14: Suppose that u_1 is a d-move w.r.t. C. For the candidate $C' = (R, D \cup \{u_1\}, \Sigma)$, the following hold:

1) Each d-move v w.r.t. C with $v \neq u_1$ is also a d-move w.r.t. C'.
2) Each d-move w.r.t. C' is either a d-move w.r.t. C or the parent of u_1 in $G \setminus R$.
3) Neither child of u_1 in $G \setminus R$ is an s-move w.r.t. C'.
4) Each s-move w.r.t. C that is not a child of u_1 in $G \setminus R$ is also an s-move w.r.t. C'.
5) Each s-move w.r.t. C' is also an s-move w.r.t. C.

Proof: Statements 1 and 4 follow from Lemmas 12 and 13 immediately, respectively. Statements 3 and 5 are obvious. To prove Statement 2, suppose that u_2 is a d-move w.r.t. C' but not a d-move w.r.t. C. Since u_2 is a d-move w.r.t. C', $u_2 \neq u_1$. Moreover, since C and C' have the same transfer set (namely, R), the reason that u_2 becomes a d-move w.r.t. C' can only be that u_1 is a child of u_2 in $G \setminus R$. Q.E.D. ■

Lemma 15: Suppose that u_1 is an s-move w.r.t. C. Let p_1 be the parent of u_1 in $G \setminus R$ and u_1' be the sibling of u_1 in $G \setminus R$. For the candidate $C' = (R, D \cup \{p_1\}, \Sigma \cup \{u_1\})$, the following hold:

1) Each d-move v w.r.t. C with $v \neq p_1$ is also a d-move w.r.t. C'.
2) Each d-move w.r.t. C' is either a d-move w.r.t. C or the parent of p_1 in $G \setminus R$.
3) u_1' is not an s-move w.r.t. C'.
4) Each s-move u_2 w.r.t. C with $u_2 \notin \{u_1, u_1'\}$ is also an s-move w.r.t. C'.
5) Each s-move w.r.t. C' is also an s-move w.r.t. C.

Proof: Statements 1 and 4 follow from Lemmas 12 and 13 immediately, respectively. Statements 3 and 5 are obvious. To prove Statement 2, suppose that u_2 is a d-move w.r.t. C' but not a d-move w.r.t. C. Since u_2 is a d-move w.r.t. C', $u_2 \neq p_1$. Moreover, since C and C' have the same transfer set (namely, R), the reason that u_2 becomes a d-move w.r.t. C' can only be that a child v_2 of u_2 in $G \setminus R$ belongs to $\{p_1, u_1\}$. However, v_2 cannot be u_1 because otherwise u_2 would be p_1. So, $v_2 = p_1$ and in turn u_2 is the parent of p_1 in $G \setminus R$. Q.E.D. ■

Lemma 16: Suppose that u_1 is an s-move w.r.t. C, its children in G are v_1 and w_1, and its children in $G \setminus R$ are $\tilde{v}_1$ and $\tilde{w}_1$, where $v_1 \geq_G \tilde{v}_1$ and $w_1 \geq_G \tilde{w}_1$. For the candidate $C' = (R \cup \{(u_1, v_1)\}, D, \Sigma)$ w.r.t. (S, G), the following statements hold:

1) Every d-move w.r.t. C is a d-move w.r.t. C'.
2) If u_2 is a d-move w.r.t. C' but not a d-move w.r.t. C, then u_2 is a proper ancestor of u_1 in $G \setminus R$.
3) Every s-move w.r.t. C other than u_1 is still an s-move w.r.t. C'.
4) If u_2 is an s-move w.r.t. C' but not an s-move w.r.t. C, then u_2 is $\tilde{w}_1$ or a child of a proper ancestor of u_1 in $G \setminus R$.

Proof: Statements 1 and 3 follow from Lemmas 12 and 13 immediately, respectively. For convenience, let $R' = R \cup \{(u_1, v_1)\}$. We next prove Statements 2 and 4 separately.

Statement 2: Suppose that u_2 is a d-move w.r.t. C' but not a d-move w.r.t. C. Towards a contradiction, assume that u_2 is not a proper ancestor of u_1 in $G \setminus R$. Then, the subtree of $G \setminus R$ rooted at u_2 is the same as the subtree of $G \setminus R'$ rooted at u_2. But now, since C and C' have the same sets of duplication vertices and speciation vertices (namely, D and Σ), u_2 cannot become a d-move w.r.t. C'. So, we have a contradiction.

Statement 4: Two simple but crucial observations are in order. First, for each vertex x of G that is not an ancestor of u_1 in $G \setminus R$, $M_R(x) = M_{R'}(x)$. Moreover, for each vertex $x \notin \{u_1, v_1\}$, x is unmarked w.r.t. C if and only if x is unmarked w.r.t. C'. Suppose that u_2 is an s-move w.r.t. C' but not an s-move w.r.t. C. Let $\tilde{r}$ be the root ancestor of u_1 in $G \setminus R$. For a contradiction, assume that u_2 is neither $\tilde{w}_1$ nor a child of a proper ancestor of u_1 in $G \setminus R$. Then, since neither $\tilde{v}_1$ nor $\tilde{r}$ is an s-move w.r.t. C', u_2 is neither an ancestor nor a child of an ancestor of u_1 in $G \setminus R$. So, by the two observations in the above, the fact that u_2 is an s-move w.r.t. C' implies that u_2 is also an s-move w.r.t. C. However, this is a contradiction. Q.E.D. ■

Appendix B: Speeding up k-*AllTransSet*

We here present *k-AllTransSet2*, which is a sped-up version of *k-AllTransSet*. It is depicted in Figure 5. Unlike *k-AllTransSet*, *k-AllTransSet2* has no input. So, we use global variables to memorize the following:

- The current candidate $C = (R, D, \Sigma)$. (*Comment:* Initially, $C = (\emptyset, \emptyset, \emptyset)$.)
- $|R| + |D|$. (*Comment:* Initially, $|R| + |D| = 0$.)
- The move μ w.r.t. C to be eliminated next. (*Comment:* Initially, μ is the smallest initial move.)
- $G \setminus R$. (*Comment:* Initially, $G \setminus R$ is a copy of G.)

Lemma 17: Subroutine *k-AllTransSet2* takes $O(3^k + \#_{sol} \cdot k)$ time, where $\#_{sol}$ is the number of k-transfer set w.r.t. (S, G).

Proof: Obviously, the total time needed for outputting the optimal transfer sets is $O(\#_{sol} \cdot k)$. So, in the remainder of this proof, we ignore the time needed for outputting the optimal transfer sets.

We use another global stack (called the *history stack*) to keep track of the history of how the global variables have

Input: None.
Output: All transfer sets R' w.r.t. (S, G) whose cost is at most k.

1. If $|R| + |D| \geq k$, then return.
2. Let μ' and μ'' be the children of μ in G.
3. If μ is a d-move w.r.t. C, then perform the following steps:
3.1. Modify C by adding μ to D.
3.2. Try to find an extreme move ν w.r.t. C.
3.3. If ν is found in Step 3.2, then set $\mu = \nu$ and recursively call *k-AllTransSet2*; otherwise, output R.
3.4. Restore the global variables so that they have the same values as they did before Step 3, and then return.
4. If μ is an s-move w.r.t. C, then perform the following steps:
4.1. Modify C by adding p to D and μ to Σ, where p is the parent of μ in $G \setminus R$.
4.2. Try to find an extreme move ν w.r.t. C.
4.3. If ν is found in Step 4.2, then set $\mu = \nu$ and recursively call *k-AllTransSet2*; otherwise, output R.
4.4. Restore the global variables so that they have the same values as they did before Step 4.
4.5. Modify C by adding (μ, μ') to R.
4.6. Try to find an extreme move ν w.r.t. C.
4.7. If ν is found in Step 4.6, then set $\mu = \nu$ and recursively call *k-AllTransSet2*; otherwise, output R.
4.8. Restore the global variables so that they have the same values as they did before Step 4.
4.9. Modify C by adding (μ, μ'') to R.
4.10. Try to find an extreme move ν w.r.t. C.
4.11. If ν is found in Step 4.10, then set $\mu = \nu$ and recursively call *k-AllTransSet2*; otherwise, output R.
4.12. Restore the global variables so that they have the same values as they did before Step 4, and then return.

Fig. 5. The subroutine *k-AllTransSet2*

been modified. Initially, the history stack is empty. Every time we modify a global variable, we memorize how it is done in the history stack. With the help of this stack, we can perform Steps 3.4, 4.4, 4.8, and 4.12 of *k-AllTransSet2* in $O(1)$ time. Moreover, by the discussion in Section V-B, we can perform Steps 3.2, 4.2, 4.6, and 4.10 in $O(1)$ amortized time. So, excluding the recursive calls, each step of *k-AllTransSet2* can be done in $O(1)$ amortized time. Now, since there are a total number of at most $O(3^k)$ recursive calls, the total time complexity is $O(3^k)$. Q.E.D. ■

It is worth mentioning that we cannot have an algorithm for enumerating all optimal transfer sets for a given pair (S, G) whose time complexity is better than $O(\#_{sol} \cdot k)$. This is because there are $\#_{sol}$ optimal transfer sets and it takes $O(k)$ time to output each of them. Of course, $\#_{sol}$ is always smaller than or equal to 3^k. Indeed, $\#_{sol}$ is usually much smaller than 3^k.

We can also output all optimal transfer sets w.r.t. (S, G) in a compact form as follows. Remember that during the execution of *k-AllTransSet2*, R is always the current transfer set. We view R as a stack so that the earlier an edge is added to R, the closer the edge is to the bottom of R. When R becomes the first optimal transfer set w.r.t. (S, G), we output it completely. Later, every time R becomes an optimal transfer set w.r.t. (S, G) again, we only output those edges of R that have been pushed into R after the previous optimal transfer set w.r.t. (S, G) has been (partially or completely) outputted. In this way, the total time needed for outputting the optimal transfer sets w.r.t. (S, G) does not exceed the total time needed for seeking them. So, we have:

Lemma 18: Subroutine *k-AllTransSet2* finds and outputs all optimal transfer sets w.r.t. (S, G) in a compact form within $O(3^k)$ time.