A Linear Time Algorithm for Tree Mapping

Oliver Eulenstein *

August 8, 1996

Inconsistencies between a phylogenetic tree calculated from a set of related genes and known species phylogeny may be due to duplications events in the gene family under study. Tree Mapping attempts to pinpoint gene duplications in order to explain such discrepancies. The effort in this explanation is measured using a tree mapping measure. Very little algorithmic work has so far been invested into the computation of these measures and an algorithm in practical use today (Page, Syst. Biol. 43:58–77,1994) runs in time quadratic in the number genes. In this paper we present a linear time algorithm for a tree mapping measure. This improvement makes the processing of large gene and species trees possible as they may arise in the study of large gene families.

*University of Bonn, Dept. of Computer Science, Research Group of Prof.Lengauer, Römerstr. 164, D-53117 Bonn, Germany, e-mail: oliver@center.informatik.uni-bonn.de

1 Introduction

Around 1960 the first phylogenetic trees were reconstructed from genes sequences (see Fitch and Margoliash [3]). These were called gene trees and the goal was from them to reconstruct a phylogenetic tree for the species, called species tree. However, gene trees and their corresponding species tree may be inconsistent. One possible reason for this are unrecognized gene duplications in the gene tree. For example, a biologist may reconstruct the correct gene tree of a sample of globin–genes containing $\alpha - human$, $\beta - chimp$ and $\alpha - horse$ in Figure 1. Obviously this gene tree contradicts the commonly accepted species tree, since it clusters man with horse and not with the chimpanzee. The discrepancy may

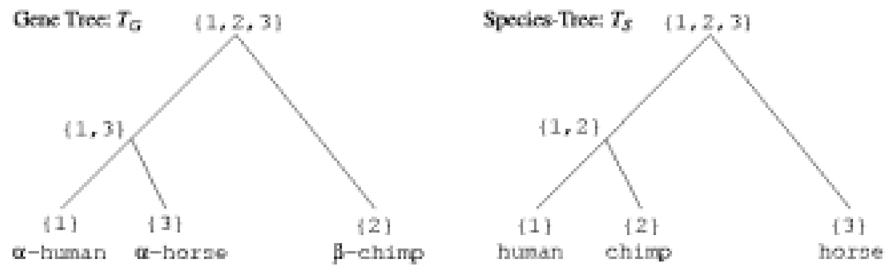


Figure 1: INCONSISTENT GENE AND SPECIES TREE.

arise from an unrecognized duplication of genes in the past. Today it is commonly accepted that in the history the globin-specifying gene duplicated, and the two copies diverged to become the α and β families. The biologist in our example was not aware of the duplication event in the globine family. So (s)he reconstructs the gene tree not recognizing the gene duplication. Figure 2 represents the gene tree of the sample genes taken into account the ancient duplication event. This tree, called the reconciled tree [12], explains the inconsistency of the gene and species tree by postulating a gene duplication. Embedding the biologists gene tree into the reconstructed tree shows that the inconsistency between gene and species tree results from the sample containing genes of the α and β family. Goodman *et al.* [4] outlined a model that explains the inconsistency between gene and

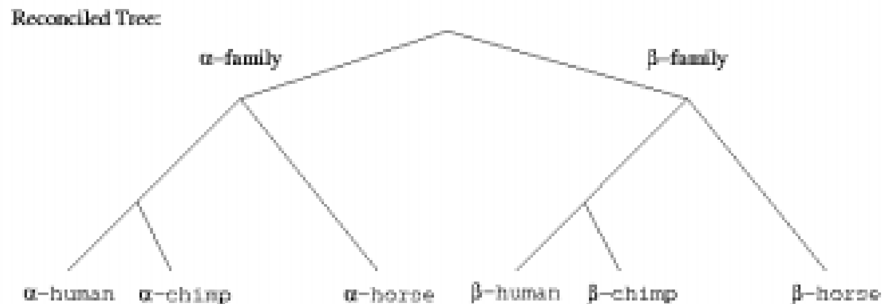


Figure 2: RECONCILED TREE

species tree using the minimal number of unrecognized gene duplications to reconstruct the reconciled tree. The mere number of postulated duplications, however, does not yet adequately describe whether the embedding of a gene tree into a species tree is in fact

parsimonious. Page [11], Mirkin *et al.* [10], and Guigó *et al.* [6] introduce biologically more meaningful measures for the inconsistency of a gene and a species tree. While these measures are also based on the assumption of Goodman *et al.* (that the inconsistency between the gene and species tree only depends on the minimal set of unrecognized gene duplications) they also consider the effect that these duplications have on the explanatory power of a gene tree. Since it has recently been found that these measures although formally very different are in fact equivalent (Eulenstein and Vingron [2], and manuscript in preparation) we will focus on the measure from [6]. Its precise definition will be given below.

A fast algorithm for the computation of such a biologically meaningful measure makes it feasible to deal with large gene and species trees as may arise due to the rapid increase in the amount of new biological sequences. Furthermore, in testing different hypotheses about gene or species trees large numbers of comparisons between trees need to be performed. Thus a linear time algorithm for calculating this measures of inconsistency between the gene trees and the species tree aids in the analysis of the reconstructed gene and species trees.

We first introduce in Section 2 the definitions for gene and species tree. For postulating gene duplications we give in Section 3 a function for mapping nodes of a gene tree onto nodes of a species tree. Based on this tree mapping we define in Section 4 the nodes in the gene tree that are postulated as gene duplications. Then we are ready to explain the linear time and space algorithm for the calculation of the tree mapping measure in Section 5.

2 Basic definitions

Let a set of n species and one gene from each of them be given. This seems to be a hard restriction for practical purposes, but they are useful to explain the linear time algorithm and can be easily relaxed. Due to this one–to–one relationship between genes and species we use the integers $1, .., n$ to denote both of them. Gene tree and species tree are rooted binary trees with leaves labeled $\{1\}, .., \{n\}$. We use sets to label the leaves in order to maintain consistency with the additional convention of labeling an inner node of a tree with the union over the sets at the leaves following this node (see Gordon [5]). For an example see Figure 1. We think of the edges of the tree as directed from the root to the leaves. We speak from the children of a node in a binary tree as a left and right child. The set of all such trees with n leaves is defined as $\mathbb{T}_n$.

Furthermore the following conventions are used. For a node v in a tree one arbitrarily chosen child will be denoted v_c while the other one is denoted $v_{\bar{c}}$. For a tree T the subtree underneath v is abbreviated $T(v)$. Generally, we will use $T_G := (V_G, E_G) \in \mathbb{T}_n$ for the gene tree and $T_S := (V_S, E_S) \in \mathbb{T}_n$ for the species tree. $T_G(v)$ and $T_S(v)$ denotes the subtree in T_G and T_S rooted at v, respectively. Any node in V_G corresponds to a gene but when the existence of a further gene (i.e., $\notin V_G$) is deduced this gene will be called an unobserved gene. In this context elements of V_G will also be called observed genes. We speak of "a gene being contained in a species" when we can deduce from the given data that a certain ancestral gene must have been present in an ancestral species. This is derived based on the containment of a leaf of the gene tree in a leaf-species that was assumed initially. An ancestral gene then is the common ancestor to a set of leaves. Therefore a species which

is ancestral to all corresponding species must have contained this ancestral gene because otherwise the leaf-genes could not be derived from the ancestral one.

3 Tree mapping

The postulation of duplication events is based on a function $M : V_G \rightarrow V_S$ called tree mapping. M maps a gene $g \in V_G$ onto the most recent (lowest) species containing this gene. Formally this is the least common ancestor in the species tree of the leaf-species which contained the given (leaf-)genes. Consider for example the gene $\{1,3\}$ in Figure 1. The leaves of $T_G(\{1,3\})$ are $\{1\}$ and $\{3\}$. Hence their least common ancestor in T_S is $\{1,2,3\}$ and we have $M(\{1,3\}) = \{1,2,3\}$. Formally we define the mapping as follows.

Definition 3.0.1
$M : V_G \rightarrow V_S$,

$$M(a) = x :\Longleftrightarrow a \subseteq x \wedge \nexists x' \in V_S : a \subseteq x' \subset x$$

4 Gene duplications and measuring inconsistency between gene and species tree

The function M is used to postulate gene duplications. A duplication of a gene a is postulated exactly if M maps one of a's children, say a_c, on the same species as a itself, i.e. $M(a) = M(a_c)$. We will give now a short insight for postulating a duplicated gene a under the condition $M(a) = M(a_c)$.
The rationale for this definition is conveniently explained based on an example. Consider gene $a := \{1,2,3\}$ from Figure 1. M tells us that a and its child $a_c := \{1,3\}$ are contained in species $x := \{1,2,3\}$. Thus all species in $T_S(x)$ contain only genes descending from a_c and no genes descending from a. But the mapping M tells us that the child $a_{\bar{c}} := \{2\}$ of a is contained in species $\{2\}$ of $T_S(x)$. A way out of this contradiction is to postulate a duplication of gene a into the copies a^+ and a^-. We denote the real gene trees rooted at a^+ and a^- by $+$tree and $-$tree, respectively. Now a species in $T_S(a)$ can contain a gene of the $+$tree and the $-$tree. Say gene a_c is in the $-$tree. Then all species in $T_S(a)$ can only contain a gene in the $-$tree that descended from a_c. We conclude that gene $a_{\bar{c}}$ must be in the $+$tree. Its succeeding genes contained in the species $\{1,2\}$ and $\{1,2,3\}$ are still unobserved. The reconciled tree including the postulated gene duplication is shown in Figure 2.
Depending on whether genes of both lineages can be found in the same species as their duplicates we introduce the distinction among nodes of the gene tree.

Definition 4.0.2 (Partition of V_G)

Genes which are not duplicated.
$$V_0(T_G, T_S) := \{a \in Vi_G \mid M(a) = M(a_c) \wedge M(a) = M(a_{\bar{c}})\},$$

Duplicated genes with only one known copy.
$$V_1(T_G, T_S) := \{a \in Vi_G \mid M(a) \neq M(a_c) \dot{\vee} M(a) \neq M(a_{\bar{c}})\}$$

Duplicated genes with two known copies.
$$V_2(T_G, T_S) := \{a \in Vi_G \mid M(a) \neq M(a_c) \wedge M(a) \neq M(a_{\bar{c}})\}$$

We will denote the set of all duplications as $D(T_G, T_S) := V_1(T_G, T_S) \cup V_2(T_G, T_S)$.

The measure of inconsistency between trees that seems most accessible to computation by a fast algorithm is due to Guigó et al. [6] (for further study of see Mirkin et al. [10]). We proceed to describe their measure. It is defined as the sum of the number of duplications and the following path lengths:

Definition 4.0.3 (L) *Let $e := (a, a_c) \in E_G$ and $p(e) := (x_0, \cdots, x_m)$ be the path in T_S with $x_0 = M(a)$, $x_m = M(a_c)$ and $m \in \mathbb{N}_0$. The path length of p not counting x_0 and x_m is defined as follows.*

$$|p(e)| := \begin{cases} m-1 & m > 1 \\ 0 & m = 0, 1 \end{cases}$$

For a node $a \in V_G$ and its children $a_c, a_{\bar{c}} \in V_G$ we then define $l : V_G \mapsto \mathbb{N}$:

$$l(a) := \begin{cases} |p((a, a_c))| + |p((a, a_{\bar{c}}))| & a \in V_0 \\ |p((a, a_c))| + |p((a, a_{\bar{c}}))| + 1 & a \in V_1 \\ 0 & a \in V_2 \end{cases}$$

$L : \mathbb{T}_n \times \mathbb{T}_n \mapsto \mathbb{N}_0$:

$$L(T_G, T_S) := |D(T_G, T_S)| + \sum_{a \in V_G} l(a)$$

5 A Linear–Time Algorithm to compute L

The goal is to calculate the tree mapping $M(T_G, T_S)$ in $O(n)$ time and space. This clearly falls into two tasks. First one needs to compute for all nodes in the gene tree their image under the mapping function M. This will be dealt with in Section 5.2. The number of duplications then follows trivially. Calculating L, although not quite as obvious, is also easily achieved in $O(n)$ time as follows.

5.1 Calculating L in time and space $O(n)$ if the tree mapping M is known

In a preorder run through T_S we label each node $s \in V_S$ with its depth $d(a)$ where $d(s)$ is defined as the path length from the root of T_S to s. Then we calculate L in time $O(n)$ by a preorder run in T_G as follows. We initialize $\mathcal{L} := 0$. For a node $g \in V_G$ and its children $g_c, g_{\bar{c}} \in V_G$ we calculate in time $O(1)$:

$$\mathcal{L} := \begin{cases} \mathcal{L} + d(M(g)) - d(M(g_c)) - d(M(g_{\bar{c}})) - 2 & M(g) \neq M(g_c), M(g) \neq M(g_{\bar{c}}) \\ \mathcal{L} + d(M(g)) - d(M(g_c)) & M(g) = M(g_{\bar{c}}) \neq M(g_c) \\ \mathcal{L} + d(M(g)) - d(M(g_{\bar{c}})) & M(g) = M(g_c) \neq M(g_{\bar{c}}) \\ \mathcal{L} & M(g) = M(g_c) = M(g_{\bar{c}}) \end{cases}$$

Obviously we have $\mathcal{L} = L(T_G, T_S)$. Thus we need $O(n)$ time and space to calculate $\mathcal{L}$ when the tree mapping M is given.

5.2 Calculating the Tree Mapping M in time and space $O(n)$

The key task of calculating L is to calculate the tree mapping M in linear time. The program given by Page [12] requires $O(n^2)$ time to compute the tree mapping M. His method does not make efficient use of the structures T_G and T_S. Since M is essentially a least common ancestor (lca) our algorithm merges ideas similar to lca computation (see Harel and Tarjan [7], Schieber and Vishkin [8] and an improved version of this algorithm is outlined by Lisa Legrand [9]). The tree mapping problem can be reduced in linear time and space to the lca problem. Thus the asymptotic performance of these algorithms is the same as ours. However, our algorithm is less technical than these algorithms and therefore can be easily implemented. It is especially designed to calculate M and in an extended version it has several features used to carry out tasks of practical purposes. In difference to the lca algorithms of Tarjan *et al.* and Vishkin *et al.* our algorithm has neither a preprocessing step nor packs several vertices into a single $O(\log(n))$ bits number to calculate least common ancestors. Our algorithm uses a different method based on a disjoint–set data structure that is very similar to the disjoint–set data structure described in Cormen *et al.* [1].
The key idea of the algorithm is to find for a subset $S \subseteq \{1, \cdots, n\}$ all nodes of V_G that map under M to nodes of $2^S \cap V_S$. These are all nodes of V_S that represent subsets of S. Initially the algorithm starts with $S = \{s\}$ for a certain $s \in \{1, \cdots, n\}$. By definition we know $M(s) = s$. For the general step we assume to have a subset $S \subset \{1, \cdots, n\}$ with all nodes of V_G known that map under M to nodes of $2^S \cap V_S$. Further we assume to know the image under M for these nodes. Now S is extended by a new element $t \in \{1, \cdots, n\} \setminus S$. The algorithm finds all nodes in V_G that map to a node in $2^S \cap V_S$ containing the new element t and calculates their tree mapping.
The entire algorithm consists mainly of the procedure DFS and the procedure GENE_TREE_WALK. Procedure DFS visits only nodes of T_S and procedure GENE_TREE_WALK visits only nodes of T_G. The procedure DFS determines the subset $S \subseteq \{1, \cdots, n\}$ for which the algorithm finds all nodes of V_G that map under M to nodes of $2^S \cap V_S$. Every time the DFS procedure visits a new leaf l this subset is extended by l. Let l be the node by which the subset is extended and S' the new subset. To find all nodes in V_G that map to a node in $2^{S'} \cap V_S$ containing the new element l the procedure GENE_TREE_WALK($M^{-1}(l)$) is called.
The procedure GENE_TREE_WALK($M^{-1}(l)$) walks up the gene tree from $l = M^{-1}(l)$ to the first node that was never visited before. This node is then marked as visited by the variable VISITED_NODE. Thus every inner node in the gene tree is visited twice. The first time a node is marked as visited and the second time the node is passed by a search for another unvisited node. Is a node visited the second time by the procedure GENE_TREE_WALK($M^{-1}(l)$) it has its image in $2^{S'} \cap V_S$ and M is calculated.
For the calculation of the tree mapping M the algorithm uses a *disjoint–set* data structure (similar to [1]). This data structure maintains a collection of disjoint subsets from V_G. A subset is represented by a *basket* that is affiliated to a node $s \in V_S$. We denote a basket affiliated to s by $basket(s)$. The following operations are defined for the data structure.

1. Operations only performed by the procedure DFS
   (a) CREATE_BASKET($g : V_G, s : V_S$): Creates $basket(s)$ that contains the node g.
   (b) MOVE_BASKET($s_1, s_2 : V_S$): Changes the affiliation of $basket(s_1)$ from s_1 to s_2.

(c) MERGE_BASKET(s_1, s_2 : different nodes on the same path in T_S):
Merges $basket(s_1)$ with $basket(s_2)$. The resulting basket is assigned to either s_1 or s_2 depending on which one has the lower depth.

2. Operations only performed by the procedure WALK_GENE_TREE

(a) INSERT_INTO_BASKET($g : V_G, s : V_S$): Inserts g into $basket(s)$.

(b) DELETE_FROM_BASKET($g : V_G, s : V_S$): Deletes g from $basket(s)$.

(c) FIND_BASKET($g : V_G$): Returns the affiliation of the basket that contains the element g.

The pseudo code of the algorithm is given in Figure 3. An example for the algorithm is given in the Appendix.

Baskets have special properties that are determined by the procedures DFS and GENE_TREE_WALK. We first introduce two general observations.

Observation 5.2.1 *Let $g \in V_G$ be contained in $basket(s)$ with $s \in V_S$. If $\mathcal{M}(g) = M(g)$ we have $M(g) \subseteq s$.*

PROOF: g can only be a node in $basket(s)$ if it was initially inserted into a basket, i.e. $basket(k)$. And g can only be moved from $basket(k)$ to $basket(s)$ by the operations MOVE_BASKET in line 12 or the operation MERGE_BASKET in line 14. These operations never move elements from $basket(x)$ to $basket(y)$ with DEPTH(x) < DEPTH(y). Consequently we have DEPTH(k) $\geq$ DEPTH(s) or in set terminology $k \subseteq s$.
g can only be initially inserted into $basket(k)$ by the operations CREATE_BASKET in line 16 or INSERT_INTO_BASKET in line 32. If g is inserted by the operation CREATE_BASKET we know from the procedure INIT $k = \mathcal{M}(g)$. If g is inserted by the operation INSERT_INTO_BASKET we know from lines 29 and 31 $k = \mathcal{M}(g)$. By the precondition we know $\mathcal{M}(g) = M(g)$ and conclude $k = M(g)$.
Summing up we have $M(g) = k \subseteq s$. □

Observation 5.2.2 *Let $s \in V_S$ and $s_l \in V_S$ be the left child of s. s has a basket assigned only if the operation* MOVE_BASKET *after* DFS(s_l) *has been performed and the last operation either* MOVE_BASKET *or* MERGE_BASKET *of* DFS(PARENT(s)) *is not yet performed.*

PROOF: Baskets are only created by the operation CREATE_BASKET in line 16 for leafs of T_S. Since s is not a leaf the operation CREATE_BASKET is not invoked for s. The only possibility for s to have a basket assigned is after the procedure DFS(s_l) finishes, when DFS(s_l) has assigned a basket to s_l. Then by the operation MOVE_BASKET in line 12 $basket(s_l)$ is moved to its new affiliation s. Consequently the only time s is assigned a basket is when the operation MOVE_BASKET after DFS(s_l) is performed. The only operation to change the affiliation of $basket(s)$ is to move it up to its parent node. The basket can only be moved away from s by an operation either MOVE_BASKET or MERGE_BASKET. Such an operation is only performed shortly before DFS(PARENT(s)) finishes. □

00 CREATE_TREEMAPPING($T_G, T_S : \mathbb{T}_n, \mathcal{M} : V_G \rightarrow V_S$);
01 $\triangleright$ Input: T_G, T_S;
02 $\triangleright$ Output: The calculated tree mapping $\mathcal{M}$ for T_G and T_S;

03 **procedure** INIT();
04 $\triangleright$ At the beginning all nodes in V_G are unvisited, so we initialize:
05 $\forall g \in V_G$: VISIT_NODE(g) := FALSE;
06 $\triangleright$ We know by assumption $M(l) = l = M^{-1}(l)$ and therefore $\mathcal{M}(l)$ is initialized by:
07 $\forall g \in V_G$ with g is leaf: $\mathcal{M}(g) = g$;

08 **procedure** DFS($s : V_S$)
09 **begin**
10   **if** (s is not leaf)**then**
11   DFS(LEFT_CHILD(s));
12   MOVE_BASKET(LEFT_CHILD(s), s);
13   DFS(RIGHT_CHILD(s));
14   MERGE_BASKET(RIGHT_CHILD(s), s);
15   **else**
16   CREATE_BASKET(($\mathcal{M}^{-1}(s)$), s);
17   GENE_TREE_WALK($\mathcal{M}^{-1}(s)$);
18         $\triangleright$ $\mathcal{M}^{-1}(s)$ is the leaf in V_G that has the same label as s in V_S.
19 **end**
20 **procedure** GENE_TREE_WALK($g : V_G$)
21 *species_l*, *species_r* : V_S;
22 **begin**
23   **while** ($g \neq$ROOT(T_G) and VISIT_NODE(PARENT(g))) **do**
24   g :=PARENT(g);
25   *species_l* :=FIND_BASKET(LEFT_CHILD(g));
26   *species_r* :=FIND_BASKET(RIGHT_CHILD(g));
27   **if** (DEPTH(*species_l*) $<$
28   DEPTH(*species_r*)) **then**
29             $\mathcal{M}(g)$:= *species_l*;
30   **else**
31             $\mathcal{M}(g)$:= *species_r*; **fi**;
32   INSERT_INTO_BASKET($g, \mathcal{M}(g)$);
33   DELETE_FROM_BASKET(LEFT_CHILD(g), *species_l*);
34   DELETE_FROM_BASKET(RIGHT_CHILD(g), *species_r*); **od**;
35   **if** ($g \neq$ROOT(T_G)) **then**
36   VISIT_NODE(PARENT(g)) :=True; **fi**;
37 **end**

38 **begin**
39   INIT();
40   DFS(ROOT(T_S));
41 **end**

Figure 3: TREE MAPPING ALGORITHM

Definition 5.2.1 *Let $l \in V_S$ be a leaf that is currently visited by the* DFS *procedure. $lca(l)$ is the least common ancestor of all leaves in V_S that have been visited by the* DFS *procedure up to the moment l is visited (including the visit of l). We define $P(l)$ as the path in T_S from $lca(l)$ to l.*

The role $P(l)$ plays is shown in the following two observations.

Observation 5.2.3 *Let $l \in V_S$ be a leaf currently visited by the* DFS *procedure. All currently existing baskets are affiliated to a node on $P(l)$ that has its left child completely processed by the* DFS *procedure.*

PROOF: Obviously all nodes in T_S that have a left child completely processed must be nodes in $T_S(lca(l))$. By Observation 5.2.2 we conclude that only the nodes in $T_S(lca(l))$ can currently be assigned to a basket. Now we consider $P(l)$. A node $s \neq l$ on $P(l)$ has exactly one child s_c on $P(l)$ and the other child $s_{\bar{c}}$ not on $P(l)$. Since l is not completely processed we know s_c is not completely processed by the DFS procedure. For $s_{\bar{c}}$ we now have to distinguish whether it is the left or right child of s. Is $s_{\bar{c}}$ the left child DFS($s_{\bar{c}}$) is finished and the operation MOVE_BASKET($s_{\bar{c}}, s$) has been performed. Observation 5.2.2 tells us that $T_S(s_{\bar{c}})$ does not contain a node assigned to a basket, but s is assigned to a basket. When $s_{\bar{c}}$ is a right child it is not yet visited by the DFS procedure. Observation 5.2.2 tells us that neither a node in $T_S(s_{\bar{c}})$ nor s itself is assigned to a basket. □

Observation 5.2.4 *Let $l \in V_S$ be a leaf that is currently visited by the* DFS *procedure. And let $basket(s)$ be an existing basket at this time and s_r the right child of s. If $\mathcal{M}(g) = M(g)$ we have $M(g) \nsubseteq s_r$ for any $g \in V_G$ with g in $basket(s)$.*

PROOF: Assume we have $M(g) \subseteq s_r$ for any g in $basket(s)$. g was initially inserted into $basket(\mathcal{M}(g))$. By precondition we have $\mathcal{M}(g) = M(g)$. Thus g was initially inserted into $basket(M(g))$ with $M(g) \subseteq s_r$.
To have g in $basket(s)$ it must first have been inserted into $basket(s_r)$ by DFS(s_r). After DFS(s_r) finished $basket(s_r)$ has been merged into $basket(s)$. Finally all leaves of $T_S(s)$ were visited by DFS(s_r) and we can conclude that s is not on the path $P(l)$. Observation 5.2.3 tells us then that no basket is affiliated to s, which is a contradiction.

Observation 5.2.5 *Let $g \in V_G$. If the algorithm has calculated $\mathcal{M}(g)$ it holds $M(g) = \mathcal{M}(g)$.*

PROOF: (By induction) Obviously we have $M(g) = \mathcal{M}(g)$ for a leaf $g \in V_G$ by the procedure INIT.
Consider now the computation of $\mathcal{M}(g)$ for an inner node $g \in V_G$. We make the inductive assumption that we have $M(g_c) = \mathcal{M}(g_c)$ and $M(g_{\bar{c}}) = \mathcal{M}(g_{\bar{c}})$ for the children g_c and $g_{\bar{c}}$ of g. $\mathcal{M}(g)$ is computed in lines 27 – 31 by the procedure GENE_TREE_WALK(l) for a certain $l \in V_G$. By the while loop in lines 23 – 34 GENE_TREE_WALK(l) visits the nodes on the path from g to l in inverse order. Thus a child of g, i.e. g_c must be on this path. So GENE_TREE_WALK(l) calculates $\mathcal{M}(g_c)$ and inserts g_c into $basket(\mathcal{M}(g_c))$ before g is visited. Consequently, $basket(\mathcal{M}(g_c))$ is not moved until g is visited.
$\mathcal{M}(g)$ is computed thus g must have been visited via $g_{\bar{c}}$ by a prior GENE_TREE_WALK(l') run, with $l' \neq l$. Consequently $\mathcal{M}(g_{\bar{c}})$ has been calculated and $g_{\bar{c}}$ is currently contained in

a basket.
Summing up, when g is visited by GENE_TREE_WALK(l) we have: g_c and $g_{\bar{c}}$ are currently contained in baskets, g_c is contained in $basket(\mathcal{M}(g_c))$. By our assumption g_c is contained in $basket(M(g_c))$ and $g_{\bar{c}}$ is contained in $basket(s)$ for a certain $s \in V_S$.
We know GENE_TREE_WALK(l) is called from the DFS procedure when it visits l. Thus Observation 5.2.3 tells us that currently all baskets are affiliated to nodes on $P(l)$. So we have either $s \subseteq M(g_c)$ or $M(g_c) \subset s$.

1. $s \subseteq M(g_c)$: Lines 27 – 31 calculate $\mathcal{M}(g) = M(g_c)$. We now show $M(g_c) = M(g)$. By assumption we can apply Observation 5.2.1 for s which tells us $M(g_{\bar{c}}) \subseteq s$. Thus we have $M(g_{\bar{c}}) \subseteq M(g_c)$ and can conclude $M(g) = M(g_c) = \mathcal{M}(g)$.
2. $M(g_c) \subset s$: GENE_TREE_WALK(l) calculates $\mathcal{M}(g) = s$ in line 27 – 31. We show now $M(g) = s$. Let s_l be the left and s_r be the right child of s. We know s is assigned a basket (containing $g_{\bar{c}}$). Observation 5.2.2 then tells us that DFS(s_l) has been finished. Consequently s_l is not on $P(l)$ and we have $M(g_c) \subseteq s_r$. Observation 5.2.4 applied for s tells us $M(g_{\bar{c}}) \nsubseteq s_r$ and Observation 5.2.1 tells us $M(g_{\bar{c}}) \subseteq s$. So we have either $M(g_{\bar{c}}) = s$ or $M(g_{\bar{c}}) \subseteq s_l$.
   For $M(g_{\bar{c}}) = s$ we have $M(g_c) \subseteq s_r \subset s = M(g_{\bar{c}})$ and conclude $M(g) = s = \mathcal{M}(g)$. For the case $M(g_{\bar{c}}) \subseteq s_l$ we have $M(g_c) \subseteq s_r$ and conlude $M(g) = s = \mathcal{M}(g)$.

This proves the induction step and the observation is shown. □

Every node in T_G is visited by the procedure GENE_TREE_WALK and thus $\mathcal{M}$ is calculated for each node. Consequently when the algorithm terminates Observation 5.2.5 tells us $\mathcal{M} = M$.

Time and space complexity
The running time of the algorithm depends on the representation of the disjoint–set data structure. We use the *disjoint–set forest* as described in [1] to represent the baskets. A basket is represented by a rooted tree in the disjoint–set forest. Each tree in this forest is affiliated to a node $s \in V_S$ by a link between its root and s. An element in a tree represents a node in the basket. Therefore each node in a tree is linked with the node in V_G that is represented by it. By the operation CREATE_BASKET(g, s) a tree consisting only of its root is generated. This root is linked with the node s in V_S and with the node g in V_G. Thus the operation CREATE_BASKET(g, s) needs constant time and space. The MERGE_BASKET operation is performed by the method union by rank (see [1]). This method creates a link pointing from the root of the tree with fewer nodes to the root of the tree with more nodes. Thus we can do a MERGE_BASKET operation in constant time and space. And a tree with k nodes in the disjoint–set forest has $O(\log(k))$ height.
For the FIND_BASKET(g) operation we use the method path compression (see [1]). The method uses two passes. In the first pass the method finds for the input node g its representation in a tree T of the disjoint–set forest. Then the method walks up the tree T to its root and thereby finds the node in V_S representing the basket that contains g. In the second pass the method again walks from g to the root of T and makes each visited node point to the root of T. During the execition of the whole algorithm for each node in V_G exactly one FIND_BASKET(g) operation is performed. Consequently each edge in the disjoint–set forest is visited twice from FIND_BASKET operations during a complete run of the algorithm. The disjoint–set forest has maximally $O(n)$ edges. Hence all FIND_BASKET operations runs in $O(n)$ time. Obviously one FIND_BASKET(g) operation needs constant space.

The MOVE_BASKET operation is performed for a tree in the disjoint–set forest. It simply changes the link from the root to a node s in V_S to the parent node of s. This needs constant time and space.
Summing up the results, the operations CREATE_BASKET, MERGE_BASKET and MOVE_-BASKET need constant time and space. We have each of this operations executed $O(n)$ times for a run of the algorithm. Hence these operations runs in $O(n)$ time and space. We have seen that all FIND_BASKET operations need $O(n)$ time. Since all FIND_BASKET operations need $O(n)$ time and one FIND_BASKET operations needs constant space we need $O(n)$ space to perform all FIND_BASKET operations.
Finally we need linear space and time to perform all operations for the disjoint–set data structure in one run of the algorithm. Since the algorithm visits $O(n)$ nodes our algorithm runs in linear time and space.

6 Generalizations to model and algorithm

We have so far followed the convention of Mirkin *et al.* [10] to assume that there is exactly one gene from each species given. However, this is not realistic in practical applications [12]. In fact, the above algorithm does not depend on this assumption at all. Generalizing it to many genes per species only requires keeping a list of genes for each leaf in V_S. Thus the list represents genes that are contained in the species–leaf. The algorithm is generalized to handle this by modifying the DFS procedure. When the DFS procedure visits a leaf GENE_TREE_WALK(g) is performed for each gene $g \in V_G$ of the leaf's list. Another valuable generalization that does not alter the basic algorithm and still makes it much more valuable in practice concerns the binary nature of gene and species trees. Frequently, tree reconstruction programs cannot decide about the order of a series of bifurcations and it is safer to assume a multi-furcating tree. Our algorithm can handle this situation as well.

7 Discussion

The inconsistency measure for which we just gave a linear time and space algorithm is only one of several such measures introduced by Page [11], Guigó *et al.* [6], and Mirkin *et al.* [10]. Especially Page [12] is making extensive use of his measure in biological applications. In Eulenstein and Vingron [2] we have shown the equivalence of L and another measure used in Mirkin *et al.* [10]. Furthermore, the measure used by Page is equivalent to these two as well (manuscript in preparation). Page presents in [12] a program to compute his inconsistency measure. Analyzing this code reveals a run time of $O(n^2)$. The other equivalent measures from [10] are given in mathematical form which, if translated naively into an algorithm, would yield an $O(n^3)$ time complexity. The value of our algorithm thus lies in providing a means of computing the inconsistency between a gene tree and a species tree in linear time in general. Our algorithm has been implemented and is currently being integrated into a program package for the analysis of gene trees. The program allows many genes per species and multi-furcating species trees.

8 Acknowledgments

I am indebted to Martin Vingron and Benno Schwikowski for the helpful discussions about the algorithm and the careful reading of this extended abstract.

References

[1] T. H. .Cormen, C. E. Leiserson, and R. L. Revest, Introduction to Algorithms, (MIT Press, 1990).

[2] O. Eulenstein and M. Vingron, On the equivalence of two tree mapping measures. "Arbeitspapiere der GMD", No. 936, Germany.

[3] W. Fitch and E. Margoliash, Construction of phylogenetic trees, Science. 155 (1967) 279 – 284.

[4] 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, Syst.Zool.. 28 (1979) 132 – 168.

[5] A. Gordon, A review of hierarchical classification, Journal of Royal Statistical Society. 150 (1987) 119 – 137.

[6] R. Guigó, I. Muchnik, and T. F. Smith, Reconstruction of ancient molecular phylogeny. Mol. Phylog. Evol. ., to appear 1996.

[7] D. Harel and R. E. Tarjan, Fast Algorithms for Finding Nearest Common Ancestors. SIAM J. Comput.. 13 (1984) 338 – 355.

[8] B. Schieber and U. Vishkin, On Finding Lowest Common Ancestors: Simplification and Parallelization. SIAM J. Comput.. 17 (1988) 1253 – 1262.

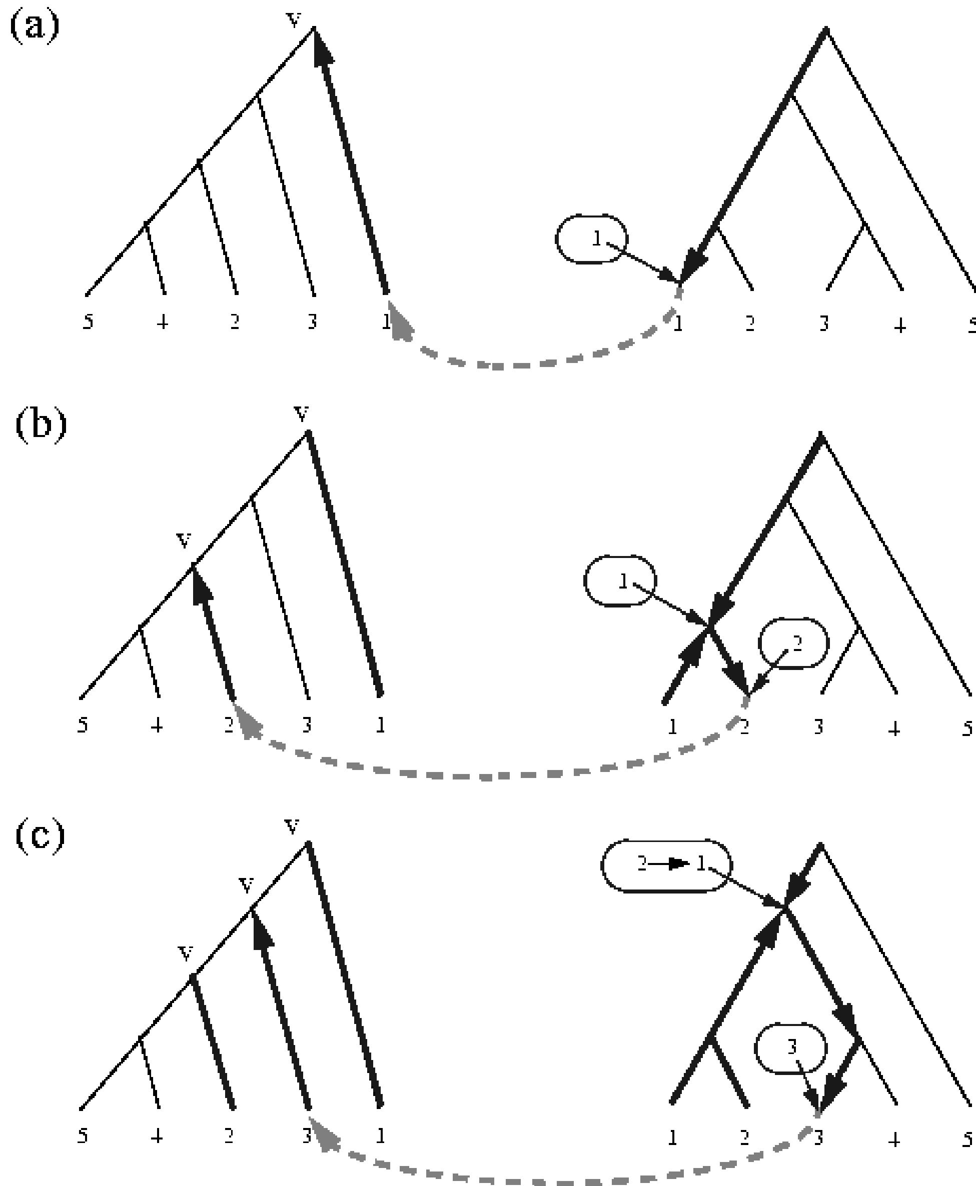
[9] L. Legrand, A Further Improved LCA Algorithm, University of Minneapolis Technical Report Archive. TR90-01 (1990).

[10] B. Mirkin, I. Muchnik, and T. F. Smith, A Biologically Consistent Model for Comparing Molecular Phylogenies, Journal of Computational Biology. 2 (1996) 493 – 507.

[11] R. D. Page, Component analysis: A valiant failure? Cladistics. 6 (1990) 119 – 136.

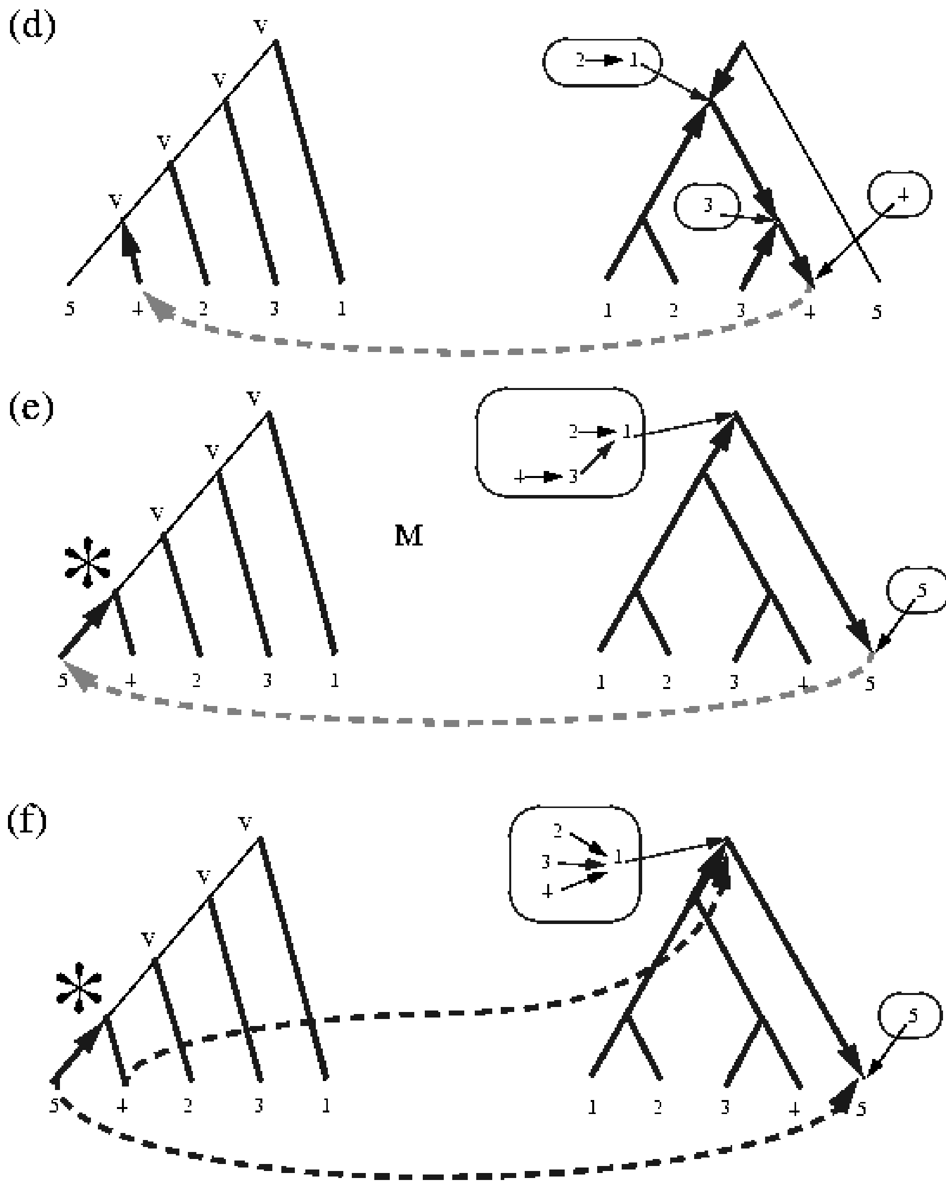
[12] R. D. Page, Maps between trees and cladistic analysis of historical associations among genes, organisms, and areas, Systematic Biology. 43 (1994) 58 – 77.

Appendix Example for the algorithm



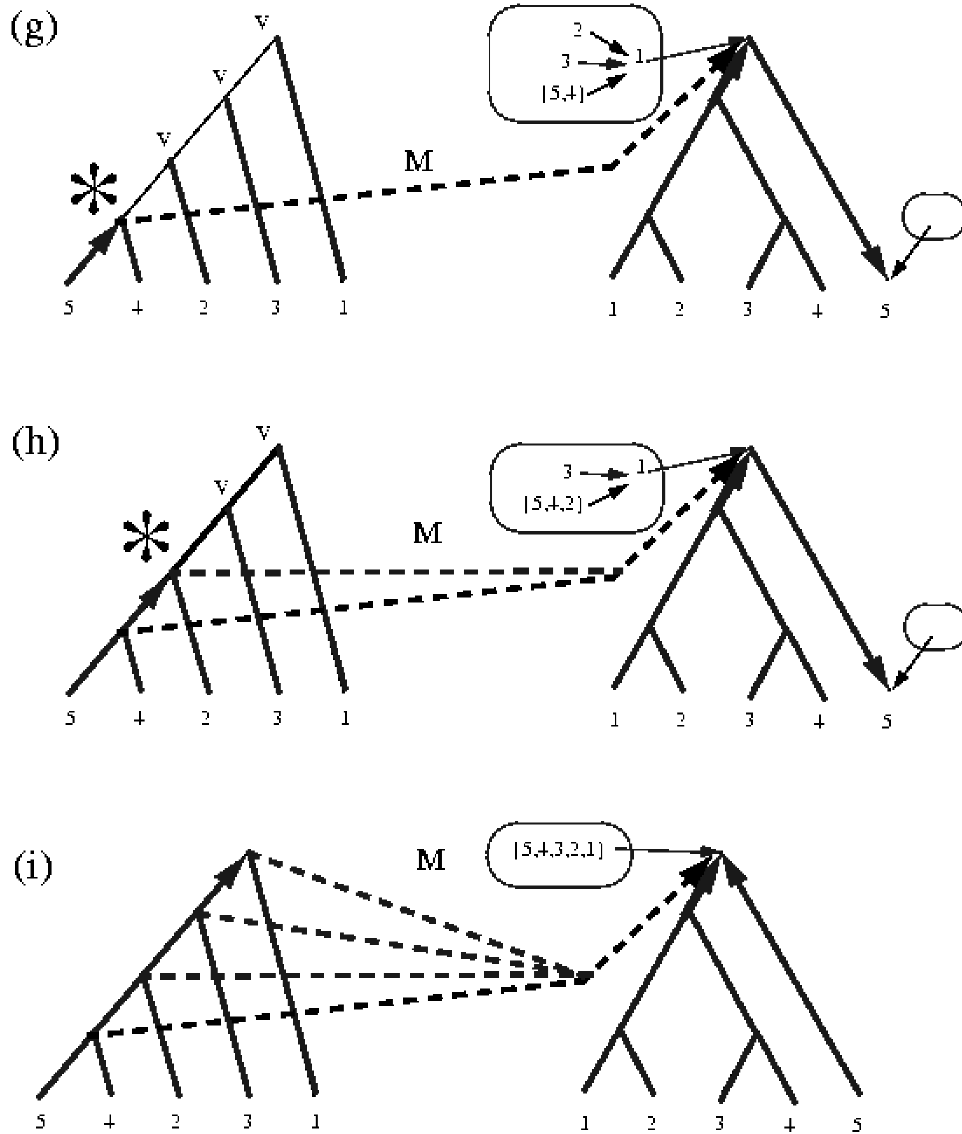
An example for the algorithm calculating the tree mapping $M(T_G, T_S)$ is given. In the pictures (a),(b),(c),(d) and (e) a snapshot of the algorithm is shown while the DFS procedure is visiting a leaf (see line 15 in the algorithm). The leaves are visited by the following DFS calls: Picture (a) DFS(1), Picture (b) DFS(2), etc. The arrows in T_S show the current status of the preorder walk for the DFS procedure. Arrows in T_S directed to the root point to a node for that we have finished the DFS procedure for the right child but not for the left child. Arrows in T_S directed away from the root point to a node for that the DFS procedure is still in progress. For the tree T_G the progressing of the procedure Walk_Gene_Tree is shown. An arrow here shows the nodes that are visited by

the procedure WALK_GENE_TREE when called by the DFS procedure. Edges that were visited by the procedure WALK_GENE_TREE in a prior DFS call are marked bold. An inner node is marked by V if it was visited by the procedure WALK_GENE_TREE which means its variable VISIT_NODE is set to true. Bold edges in T_G were visited by prior WALK_GENE_TREE calls.



Picture (d) is a good example for Observation 5.2.3. Leaf 4 in T_S is currently visited by a DFS procedure. All leaves in T_S that have been visited by the DFS procedure up to this moment are $1, 2, 3$ and 4. Thus we have $lca(1, 2, 3, 4) = \{1, 2, 3, 4\}$ and therefore $P(4) = \{\{1, 2, 3, 4\}, \{3, 4\}, 3, 4\}$. Each inner node in this example has its left child completely processed by the DFS procedure and is assigned by a basket.
Pictures (e) and (f) show a small example for the path compression in the basket affiliated

to the root of T_S. In Picture (e) the node $\{3,4\}$ marked by a star is visited the second time. Thus in Picture (f) the operations FIND_BASKET(5) and FIND_BASKET(4) are performed for calculating $\mathcal{M}(\{3,4\})$. In Picture (d) the FIND_BASKET(4) operation walks up the path from 4 to 1 in $basket(\{1,2,3,4,5\})$. Finally all nodes on the path are linked to node 1 in the basket as can be seen in Picture (f) after FIND_BASKET(4) is completed with the path compression.



The calculation of $\mathcal{M}(\{3,4\})$ is shown in Pictures (f) and (g). The FIND_BASKET operations performed for the children of $\{3,4\}$ return the nodes 5 and $\{1,2,3,4,5\}$ in T_S. And we have node $\{1,2,3,4,5\}$ which is of lower depth than node 5. Thus the algorithm calculates $\mathcal{M}(3,4) = \{1,2,3,4,5\}$. We know $M(5) = 5$ and $M(4) = \{1,2,3,4,5\}$ and can conclude $M(\{3,4\}) = \{1,2,3,4,5\} = \mathcal{M}(3,4)$. Pictures (h) and (i) show the calculation of the remaining tree mappings as performed by GENE_TREE_WALK(5).