## 5 Conclusions

We have given efficient algorithms for two natural combinatorial problems, and have demonstrated that these algorithms can have a significant impact on fundamental problems in computational evolutionary biology.

## References


[1] A.V.Aho, Y. Sagiv, T.G. Szymanski, and J.D. Ullman. *Inferring a tree from lowest common ancestors with an application to the optimization of relational expressions*, SIAM J. of Computing, Vol 10, No. 3, 1981, pp. 405-421.

[2] D. Eppstein, Z. Galil, and G. F. Italiano. *Improved Sparsification*, Tech. Report 93-20, Department of Information and Computer Science, University of California, Irvine, CA 92717.

[3] S. Even and Y. Shiloach, *An on-line edge deletion problem,* J. ACM, vol. 28, No. 1 (1981), pp. 1-4.

[4] J. Felsenstein, *Numerical methods for inferring evolutionary trees,* The Quarterly Review of Biology, Vol. 57, No. 4, (1982), pp. 379-404.

[5] W. Fitch, personal communication.

[6] G. N. Frederickson, *Data Structures for On-line Updating of Minimum Spanning Trees* SIAM J. Comput. 14:781 – 798, 1985.

[7] M. Henzinger and V. King, *Randomized Dynamic Algorithms with Polylogarithmic Time per Operation*, Proceedings, STOC '95.

[8] D. Hillis, personal communication.

[9] S. Kannan, E. Lawler, and T. Warnow, *Determining the Evolutionary Tree*, to appear, Journal of Algorithms.

[10] S. Kannan, T. Warnow, and S. Yooseph, *Computing the local consensus of trees*, Proceedings, SODA '95.

rooted trees, such each tree is bijectively labelled by the species in a set $S$. Clearly, no *consensus* exists according to our definition above unless these trees are identical. Thus, the question of inferring the consensus of a set of different trees which have the same leaf set depends upon a model for handling disagreements between trees.

This is an important question in computational evolutionary biology because it is frequently the case that the biologist can not determine one single tree for a data set, but instead computes many, each of which is equally plausible. This may arise because different methods are used to construct the trees, each based possibly on a different data set for the same species set (some could be biomolecular while others could be distances). Even when one method is chosen, there can be many different equally good trees for the optimization criterion upon which the method is based, so that the output is not one best tree, but many trees, each of which has a good score for the optimization criterion. The number of different trees can be quite large.[1]

In [10], a model for problem of computing consensus trees was presented, based upon the concept of a local consensus rule. A *local consensus rule* determines the exact form of (possibly) each rooted subtree of every triple of species drawn from $S$, based upon how the trees in the input profile resolve that triple, and a *local consensus tree* is then required to have the form specified by the local consensus rule for each triple $a, b, c$ of species.

When the local consensus rule is *entire*, so that the form on each triple is specified, it is possible to construct the local consensus tree by using the algorithms of [9]. Because a resolved subtree on $a, b, c$ indicates a definite hypothesis about the evolution of $a, b, c$ from a common ancestor, most of the biologically relevant local consensus rules will not specify the form of the local consensus tree on a triple unless the evolution of that triple from a common ancestor can be determined with high probability, in which case the form is a resolved (i.e. binary) subtree. Thus, the real problem we have for most realistic local consensus rules is that they return a set of $m$ resolved triples, for $m < n^3$. It is this case which is relevant to the subtree consistency problem.

Our algorithm is as follows.

**Step 1:** We apply the local consensus rule to each triple of species. (This is necessary, since the local consensus rule can be arbitrary, and the form of the rooted tree on a triple $a, b, c$ may not be determinable by examining all the remaining triples.)

**Step 2:** We now pass this subset of $m \leq n^3$ rooted triples to a consistency algorithm, and determine if a tree consistent with each of these rooted triples exists, and construct it if it does.

**Theorem 6** *We can construct the local consensus tree of $k$ trees (assuming all returned triples are resolved) in $O(kn^3 + min\{n^2 \log n, m\sqrt{n}\})$.*

**Proof:** Step 1 takes $O(kn^3)$ time, since we must examine every triple of species $a, b, c$ in the worst case (the local consensus tree for some local consensus rules can be computed faster, but in general this is not the case). This returns a subset of $m \leq n^3$ rooted triples, which can then be passed to a subtree consistency algorithm. The determination of subtree consistency then takes $O(min\{m + n^2 \log n, m\sqrt{n}\})$. ∎

Note that by contrast, if we use the Aho et al algorithm for the subtree consistency, this will result in an $O(kn^3 + mn)$ algorithm. Thus, our algorithms for the consistency of rooted triples can be used in computing the local consensus of trees significantly faster than could have been achieved without these algorithms.

[1] Walter Fitch has reported finding 61 equally parsimonious trees on a data set.

or perhaps only have high confidence in certain rooted triples, and it is therefore necessary to have an efficient method by which the rooted triples can be resolved into a single rooted tree.

This general approach of computing the rooted trees for subsets of species has strengths in that small subproblems can sometimes be solved quickly, using techniques appropriate for the particular subsets in question. When all triple queries can be answered correctly (i.e. consistently with the true evolutionary tree $T$), it is possible to construct $T$ in only $O(n^2)$ time (if the rooted subtrees are not restricted to be binary), and $O(n \log n)$ time if the rooted subtrees are restricted to be binary[9]. When perhaps not all triples are able to be resolved with complete confidence, the biologist will need methods for reconstructing the evolutionary tree which consider only those triples for which we have complete (or high enough) confidence. It is this case which is applicable to the problem we address here. We direct the interested reader to [4] for a review article in the problem of inferring evolutionary trees under a number of different methods.

The maximum likelihood method assigns a likelihood value to a hypothesized evolutionary tree. The strength of this method is that it is based upon stochastic models of molecular evolution, and many (perhaps most) biologists have greater faith in trees supported by maximum likelihood analyses than by other methods or optimization criteria[4]. The problem with using maximum likelihood as a method for finding the best evolutionary tree is twofold:

- There is currently no algorithm or heuristic which can find optimal (or locally optimal) trees quickly, with respect to maximum likelihood.
- The cost of computing the likelihood value of a fixed tree grows prohibitively fast in the number of species (the number of species corresponds to the number of nodes in the tree, and each node is associated with a parameter which must be estimated).

As a consequence, current methods for finding "optimal" trees with respect to maximum likelihood essentially boil down to generating just a few trees using some other method, and computing the likelihood score of the these trees. The tree with the best likelihood value is then the "optimal" tree.

There are some strengths to the maximum likelihood approach to tree construction, however, when the number of taxa is very small. In this case, studies have shown that when four sequences are generated with respect to a tree $T$ and a given stochastic process, then maximum likelihood methods will with very high probability reconstruct the correct tree for those four sequences[8]. By comparison, the same studies have shown that parsimony will not perform as well on such data.

This leads us to suggest the following approach to evolutionary tree construction.

Given a set $S$ of species, pick a species $s_0$ which is an outgroup for $S$; hence, in the true evolutionary tree for $S$, the leaf for $s_0$ will be a child of the root. For every triple of species, $a, b, c$, determine, through the use of maximum likelihood analysis, the correct topology of the rooted tree on $s_0, a, b, c$. (This can be computed either through using the rootedness of the tree on $s_0, a, b, c$, or by generating the unrooted topology and rooting it on the edge leading to $s_0$.) Thus, we can infer the rooted topology for $a, b, c$. For each triple $a, b, c$ for which one rooted topology has a significantly high likelihood scores as compared to the other rooted triples, pass this rooted triple along to the consensus tree algorithm. The consensus tree, if it exists, is returned.

### 4.2 Constructing the Local Consensus of $k$ Trees

The question that is addressed here is somewhat different from the question we addressed above in a number of ways. First, we are presuming that we are given a set $S$ of species and a set $\mathcal{T}$ of

For every internal node $x$ of a tree, we create a node $g(x)$ in the directed graph which is labeled with a "label-pair", the labels of the two children of $x$. The digraph edges consist of all edges of the form $(g(x), g(y))$, where $x$ is a child of $y$ in some $T_i$. A *maximal* node in $D$ is a node with no outgoing edges. An edge $e$ is a special edge of a subtree $T_i$ iff $e$ is the label of a maximal node in $D$. This leads to the following algorithm.

The algorithm starts with all edges of $U$ colored yellow and runs in phases until no edges are left, where a single phase is as follows:

1. Mark as red all edges in $U$ which correspond to the maximal nodes in $D$.
2. Determine if a new yellow connected component of $U$ has been created.

   If so, we identify the nodes in some new component and find and discard all red edges with exactly one endpoint in this component. For each such red edge, remove the node from $D$ whose label consists of both its endpoints.

   If not, we stop and output that no consensus tree exists.

The consensus tree is defined recursively as follows. If a connected component in $U$ is a node, then the consensus tree of the component is this node. Let $C_1$ and $C_2$ be the consensus trees formed from the new component and the remainder of the graph, respectively. The consensus tree for $U$ is the tree formed by making the root of $C_1$ and of $C_2$ children of a new node, which is the root of the consensus tree.

# 4 Applications

We will show in this section two ways in which our efficient algorithms for constructing a tree consistent with a set of rooted binary subtrees can be used to assist evolutionary biologists. These are:

- To construct trees directly from biomolecular data, such as DNA, RNA, or amino acid sequences.
- To resolve different rooted trees on a species set into a local consensus tree [10].

## 4.1 Constructing Trees from Biomolecular Data

One of the fundamental problems in computational biology is the inference of evolutionary history for a set of species. Most approaches in favor are based upon solving optimization problems, which are unfortunately *NP-hard*. Because of the fundamental importance of solving these evolutionary tree problems, there is nevertheless an abundance of heuristic methods for constructing trees.

One of the most favored methods is the maximum likelihood method, which assigns a likelihood to each tree. Unfortunately, current methods for finding the maximum likelihood tree are really only tractable when the number of species is kept very small. For example, much attention has been paid to the problem of inferring the maximum likelihood tree for only four species (when the tree is unrooted, and the stochastic model is symmetric), or three species (for rooted trees, so that the stochastic model may be asymmetric). As a consequence, the biologist may be able to determine with high confidence the correct form of each rooted triple (or unrooted quartet),

**Proof:** When we delete a batch of edges, for each edge $(a,b)$ deleted such that $a$ and $b$ are now in separate yellow components, we can search from the smaller of the two components found for all red edges that are incident to vertices in that new component. Each separable red edge that is found is deleted from the graph, for a total cost of $m$ over the course of the algorithm. The non-separable edges are each visited at most $\log n$ times, for a total contribution of $O(m \log n)$ to the cost of the algorithm. This additional work of finding the separable red edges over the course of the algorithm is less than cost of the updates and queries given above, so that the running times of the two variants (deterministic or randomized) of the algorithm are as stated. ∎

The algorithm we have just given is faster than the Aho et al algorithm; however, we can also do better by using Algorithm $B$ for batch deletions, and incorporating into the algorithm a search for (not necessarily all) separable red edges in such a way that we do not increase the running time.

The following theorem can be proved.

**Theorem 4** *Each of these algorithms returns a tree $T$ for input $\mathcal{T}$ if and only if $T$ is a consensus tree for $\mathcal{T}$.*

**Theorem 5** *Using Algorithm B, we can determine consistency of $m$ rooted subtrees on $n$ leaves and construct the consensus tree if it exists, in $O(m + n^2 \log n)$ time.*

**Proof:** Note that when the subtrees are consistent, each batch of deletions has to split each previous component; thus, each batch of deletions will result in the creation of a new component. When the subtrees are inconsistent, then after the first batch in which no new component can be found, the algorithm terminates and returns a null-tree. Hence, we note that when using algorithm B for batch deletions for the subtree consistency problem, $b_0 \leq 1$, so that the batch deletion problem can be solved in $O(m + n^2 \log n)$ time. The remaining work, as we have shown above, is therefore in finding all non-separable edges.

Each non-separable edge $e$ found goes between vertices in the same component, and thus the discovery of all non-separable edges found after discovering a new component $C$ costs us $O(|C|^2)$. Since each new component found is a *smaller* component, the total cost to us for considering the non-separable edges is $O(n^2 \log n)$. ∎

### 3.6 Determining consistency of general trees

We now describe how we generalize this implementation to arbitrarily sized binary subtrees. Label each node in each subtree $T_i$ as follows: The label of a leaf is the number of the leaf. The label of an internal node is the label of its right child. Let $U = (V, E)$ be the undirected graph with $V = \{1, 2, ..., n\}$ and $E = \{(j, j') | \ \exists T_i$ with an internal node whose children are labeled $j, j'\}$. The graph $U$ consists of the nodes $V$ with all leaves of a subtree $T_i$ connected by a chain of edges in the order in which they appear in $T_i$ for all $i$. Thus, it is a generalization of the above graph $U$.

The algorithm for computing the consensus tree is the same as the above with the edges $(q, r)$ replaced by an edge $(a, b)$ for each subtree $T_i$, where $(a, b)$ is the unique edge in the "chain of leaves" of $T_i$ that connects an edge of the left subtree of the root of $T_i$ with a leaf of the right subtree, (called *special edge*).

To find all special edges efficiently, we define the following directed graph $D$.

### 3.3 The directed graph $D$

We now define our data structures.

**Data Structures:** We have two data structures: a directed graph $D$ and an undirected graph, $U$.

- $D = (V, A)$ with $V = \{(a,b)\} : \exists((a,b),c) \in \mathcal{T}$ or $\exists((a,x),b) \in \mathcal{T}\}$. Nodes $(a,c)$ and $(b,c)$ are identified. $A = \{(a,b) \rightarrow (a,c)$.
- $U = (S, E)$ with $S$ the species set and $E = \cup_{((a,b),c)\in\mathcal{T}}\{(a,b),(a,c),(b,c)\}$.

### 3.4 Structure of our algorithms

The structure of each of our algorithms is as follows:

**Step 1:** Construct $U$ and $D$. Color all nodes in $D$ and edges in $U$ *yellow*. Compute yellow components in $U$.

**Step 2:** Identify maximal nodes in $D$. Recolor these nodes and the corresponding edges of $U$ *red*.

**Step 3:** If $U$ has no edges, then return the consensus tree $T$ with a root and all nodes in $U$ children of the root. Otherwise, recompute yellow components of $U$ using one of the algorithms presented in Section 2. If no new component is found, then return *Null-tree*, else let $C_1, C_2, \ldots, C_k$ be the new component(s) found. Make a tree $T$ with root $r$; the subtrees of $T$ will be the trees constructed on each of the components $C_1, C_2, \ldots, C_k$. For each new component found, identify the set $E_{sep}$ of separable red edges incident to that new component. Delete these edges from $U$ and the corresponding nodes from $D$. Recompute maximal elements in $D$, and color these maximal elements and the corresponding edges in $U$ red. Repeat Step 3.

It is clear that the algorithm's primary need is the determination of the set of red edges whose endpoints are in different yellow components (which we call *separable red edges*), and that this is the singly most important implementation detail we need to establish. Because of this need to determine separable red edges, we will modify the batch deletion algorithms of Section 2 to find the separable red edges.

### 3.5 Analysis of running time

The construction of $U$ and $D$ takes time proportional to $m = \sum_i |T_i|$. The overall cost of determining the maximal nodes of $D$ is proportional to the size of $D = O(m)$. We use one of the batch deletion algorithms to find a new component in each phase. While discovering a new component $C$, we also examine and identify the red edges incident to it. Each separable edge $e$ is then discarded from $U$, for a total cost of $O(m)$. The cost of using batch deletion algorithms to compute the consensus tree then depends upon the cost of the batch deletion algorithm itself, and the number of times non-separable edges are visited.

**Theorem 3** *Using Algorithm A, we can construct the minimal consensus tree of $m$ rooted triples on $n$ species in $O(m\sqrt{n})$ time, or in $O(m \log^3 n)$ expected time.*

**Input:** Set $\mathcal{T}$ of rooted triples $((a,b)c)$, indicating that the lca of $a,b$ lies below the lca of $a,c$.

**Output:** Tree $T$ consistent with the rooted triples if such a tree exists.

### 3.1 Previous results

In 1981, Aho, Sagiv, Szymanski, and Ullman [1] addressed this problem of consistency of homeomorphic subtrees (each of which is binary) in a paper motivated by an application to relational databases. They presented an $O(nm)$ algorithm to find (when possible) a single tree $T$ such that each $T_i$ is a homeomorphic subtree of $T$, where $n$ is the number of leaves in $T$ and $m = |\mathcal{T}|$.

Given the set of subtrees, $\mathcal{T}$, the algorithm in [1] constructs a graph $U = (V, E)$ as follows. $V = \{1, 2, \ldots, n\}$, and $E = \{(p,q) : \exists((p,q),r) \in \mathcal{T}\}$. The connected components of $U$ are computed using any linear time algorithm. If $U$ is connected, then there is no consensus tree; otherwise, let $G_1, G_2, \ldots, G_k$ be the distinct components of $U$. For each $G_i$, and for each tree $T \in \mathcal{T}$, determine if $T$ applies to $G_i$ ($T$ applies to $G_i$ if all of the leaves in $T$ are in $V(G_i)$), and recurse on each subproblem $(G_i, \mathcal{T}^i)$, where $\mathcal{T}^i$ is the set of trees in $\mathcal{T}$ which apply to $G_i$. A consensus tree exists if and only if each subproblem has a solution. Let $T'_1, .., T'_k$ be the consensus trees generated for the $k$ subproblems. Then the consensus tree for the input is given by adding edges from the roots of the $T'_i$ to a new parent node. The proof of correctness of the Aho et al. paper may be found in their paper [1].

Note that, as stated, the algorithm may require $\Theta(nm)$ time since it repeatedly computes connectivity on a graph of up to $m$ edges. Since each computation results in at least one new component, or the computation stops, the connectivity computation is repeated no more than $n$ times.

### 3.2 Faster algorithms for subtree consistency

In this section, we will show how to use the fast batch deletion algorithms of Section 2 to produce faster algorithms for the problem of testing consistency of rooted resolved triples. As we have said, this problem is equivalent to the problem of testing consistency of rooted binary trees.

The algorithms we will present in this section will have the same basic structure as the Aho et al algorithm, in that components are found in the graph $U$, and the consensus tree returned is formed by combining the trees constructed for the components in the same way as the Aho et al algorithm. Thus, the only differences we will introduce is (a) the manner in which the components are found, and (b) whether all components are found after each batch of deletions, or just at least one.

One way to speed this up is to incorporate dynamic data structures for connectivity. We will present different algorithms (two deterministic and one randomized) for the problem of determining consistency of the rooted triples. These algorithms use dynamic batch deletion algorithms to determine connectivity, and a directed graph, $D$, which allows us to determine the new batch of edges to be deleted. When the batch deletion algorithm determines *all* new components, the minimal tree is returned, but when the batch deletion algorithm only determines some single new component, then a binary tree is returned which will be consistent with the constraints but not necessarily minimal. Depending upon the particular application, the minimality may or may not be important (it turns out to be important for the database problem, but not necessarily important for the biological problem).

**Find a (newly created) smaller undiscovered component:** After a batch is deleted, compute the connected components in each $G_i$ starting with $G_1$ until a graph is found which contains a (newly created) undiscovered component C which contains only white nodes. Any connectivity algorithm which runs in time linear in the number of edges of the modified component may be used.

A component is determined to be undiscovered and smaller by comparing its actual size to the size in the label of one of its nodes. A component is determined to be newly created by checking the subset of edges most recently deleted which are incident to the component to see if one of the edges' endpoint lies in a different component. Adjust the label of the component which previously contained C by subtracting out the size of this undiscovered component. Relabel the nodes in this undiscovered component.

Output C.

### 2.2.1 Analysis of the Batch Deletion Algorithm

We start with the analysis of the smaller undiscovered component algorithm. The initialization and adjustments take $O(n^2)$ time since there are $\lg n$ graphs and the total number of edges in $G_i$ is $O(2^i n)$.

We use the following observation:

**Lemma 1** *A set of nodes in G is a component of size no greater than $2^i$ if and only if it appears as a component in $G_i$ which contains no blue nodes.*

**Theorem 2** *This algorithm has running time $O(n^2 \log n + b_0 min\{n^2, m \log n\})$, where $b_0$ is the number of batches which do not result in the creation of a new component.*

**Proof:** A component of size no greater than $2^i$ will always be found in some $G_j$, for $j \leq i$, by Lemma 1. Since $G_j$ has size $O(2^j n)$, it takes time $O(2^i n)$ to find a new component of size no greater than $2^i$.

During the execution of the algorithm, for each $i$, a node can only be in one smaller component of size greater than $2^{i-1}$ and no greater than $2^i$. Thus, there are no more than $n/2^{i-1}$ such components during the course of the algorithm. Thus the total cost of finding all smaller components of size greater than $2^{i-1}$ and no greater than $2^i$ during the course of the algorithm is $O(n/2^{i-1} * 2^i n) = O(n^2)$. Summing over all $i$, $1 \leq i \leq \log n$, the total cost of all searches which end in the discovery of a smaller component during the course of the algorithm is $O(n^2 \log n)$. Any search which does not end in the discovery of a smaller component takes $O(\min(n^2, m \lg n))$ time. ∎

Note that we can extend this analysis to the newly created option, by noting that the test to determine if a undiscovered component is newly created takes time proportional to the number of edges which were just deleted and which are incident to that component. Therefore, these tests involve looking at each edge at most twice during the algorithm. So the cost of these tests over the whole execution of the algorithm is proportional to $m$. The same analysis as above holds.

## 3 The subtree consistency problem

Recall that the problem statement is as follows:

or insertions of edges. The queries can be answered in constant time. Adjustments to the data structure following each insertion or deletion can be done $O(\sqrt{n})$ time per edge deterministically, or in $O(\log^3 n)$ expected time per edge, using the randomized algorithm of [7].

**Theorem 1** *We can solve the batch deletion problem with the requirement that we find all components after each batch, in $O(m\sqrt{n})$ time, or in $O(m \log^3 n)$ expected time, where $m = |E|$ and $n = |V|$.*

**Proof:** We begin by determining the components of $G$, and then delete all edges in the first batch. If a new component has been created by the batch, then it must be the case that the endpoints of some deleted edge have become disconnected. By querying all deleted edges for connectivity, we can determine exactly which pairs of newly deleted edges have become separated. For each such newly separated pair, we proceed to search from each endpoint, alternating between the two searches, until one component is completely visited (this technique derives from [3]). It is not hard to see that the searches for components cost in total $O(m \log n)$ time during the course of the algorithm. If we use the best deterministic fully dynamic algorithm for connectivity, then the $m$ queries and $O(m)$ updates cost $O(m\sqrt{n})$ time; while if we use the randomized fully dynamic algorithm of [7], we can perform all the queries and updates in $O(m \log^3 n)$ time. ∎

## 2.2 Algorithm B: A faster batch deletion algorithm

In this section we will describe an algorithm which uses $O(n^2 \log n + b_0 min\{n^2, m \log n\})$ time for the batch deletion problem when we do not require that we find all components created, but only one, which must be newly discovered but need not be newly created.

Let $C$ be any connected component of $G$ which becomes disconnected after a batch of deletions. Then we call $C'$ a *smaller component* if it is a connected subset of $C$ of size no greater than half the size of $C$. Throughout the algorithm, each undiscovered component which is found is given a unique name and labeled with its size. A component is *undiscovered* if its current label is shared by a component which had previously been connected to it. A component is *newly created* if it has just been created by the most recent batch of deletions. Each node in a component has a pointer to its component label.

Note that if there is an undiscovered component then there is also an undiscovered smaller component. By definition, there must be two or more undiscovered components with the same label, and all but one must be smaller.

Similarly, if there is a newly created component, there is a newly created smaller component. The following algorithm finds an undiscovered smaller component after each batch of deletions. The algorithm may be modified, as explained below, to stop only when it finds an undiscovered smaller component which is also newly created.

**Batch deletion algorithm B:**

**Initialize:** Construct a sequence of graphs $G_1, G_2, ..., G_{\lg n} = G$ with $G_i = (V, E_i)$, and $E_i = \{(a,b) : min\{deg_G(a), deg_G(b)\} < 2^i\}$. Nodes with degree at least $d_i$ are colored blue in $G_i$, and the others are colored white. Label each node with the size of its component.

**Adjust the $G_i$:** Each time an edge is deleted adjust the appropriate $G_i$ by deleting this edge from all graphs which contain this edge. If the degree of a node falls from $2^i$ to below $2^i$, color the node white in $G_i$ and add its remaining edges to $G_i$.

When the tree $T^*$ exists, the set $\mathcal{T}$ is said to be a *compatible* set of subtrees, and the set $T^*$ is called the *consensus tree.*

Our deterministic algorithm runs in time $\min\{O(mn^{1/2}), O(m+n^2 \log n)\}$, where $m = \sum_i |T_i|$. If the randomized full dynamic connectivity data structure of [7] is used, the resulting algorithm takes time $O(m \log^3 n)$. By contrast, the best previous algorithm for this problem was given in the 1981 paper by Aho et al [1], and had running time $O(mn)$.

The efficiency of our algorithm results in part from an efficient solution to the following dynamic graph problem, called the *batch deletion problem.*

Given a graph G with $n$ nodes and $m$ edges and a sequence of $b$ batches of one or more edge deletions, then after each batch, either find a new component and output all its nodes, or determine that there is no such component. No information about the batches of deletions is given in advance of their occurrence.

For this problem, we have a simple algorithm with running time $O(n^2 \log n + b_0 \min(n^2, m \log n))$, where $b_0$ is the number of batches which do not result in a new component. For our particular application, $b_0 \leq 1$. Note that all previous algorithms for this problem require time $\Omega(d\sqrt{n})$, where $d \geq b_0$ is the total number of deletions. Hence if $d \geq n \log n + \min(n^{3/2}, m \log n/\sqrt{n})$, then the presented algorithm is more efficient than the best deterministic dynamic graph algorithms.

The rest of the paper is organized as follows. In Section 2, we present deterministic algorithms for the batch problem. We then present algorithms for testing the consistency of rooted binary subtrees in Section 3. We show how to use these algorithms for constructing evolutionary trees from biomolecular data, such as DNA, RNA, or amino-acid sequences, in Section 4. We summarize our results in Section 5.

## 2 The batch deletion problem

The batch deletion problem is as follows. We are given a graph $G$ on $n$ nodes and $m$ edges, and an (unknown) sequence of batches of edge deletions. After each batch we wish to either determine that the number of connected components has not changed, or to find a new component and output the set of nodes. One variant of the batch deletion problem requires that we identify only those components that are newly created by the previous batch of deletions. In each case we may wish either to find all such components, or simply just one.

The batch deletion problem may be done by brute force. After each batch, simply compute connectivity. If there were $b$ batches of deletions, this would result in $bm$ running time, regardless of the value of $b_0$.

A 1981 algorithm by Even and Shiloach [3] considered the dynamic graph problem for individual deletions only. Over the course of deleting every edge from a graph, their algorithm spends $O(m \log n)$ time processing individual deletions each of which results in a new component and $O(mn)$ time processing the remaining deletions. Thus, if every individual deletion in a graph resulted in a new component, their algorithm would have a $O(m \log n)$ running time. There does not seem to be any immediate way to extend their technique to derive a fast algorithm for the batch deletion problem in which a batch of deletions result in a new component.

### 2.1 Algorithm A: A fast batch deletion algorithm which find all components

We can use a fully dynamic connectivity data structure for this problem. This data structure answers queries concerning the connectivity between pairs of nodes, following individual deletions

# Constructing a Tree from Homeomorphic Subtrees, with Applications to Computational Evolutionary Biology

Monika Rauch Henzinger *
Valerie King †
and Tandy Warnow ‡

**Abstract**

We are given a set $\mathcal{T} = \{T_1, T_2, \ldots, T_k\}$ of rooted binary trees, each $T_i$ leaf-labeled by a subset $\mathcal{L}(T_i) \subset \{1, 2, \ldots, n\}$. If $T$ is a tree on $\{1, 2, \ldots, n\}$, we let $T|\mathcal{L}$ denote the subtree of $T$ induced by the nodes of $\mathcal{L}$ and all their ancestors. The *consensus tree problem* asks whether there exists a tree $T^*$ such that for every $i$, $T^*|\mathcal{L}(T_i)$ is homeomorphic to $T_i$. The consensus tree problem has applications in computational biology and in the theory of relational data bases.

We present algorithms that test if a given set of trees has a consensus tree and if so, constructs it. The deterministic algorithm takes time $\min\{O(mn^{1/2}), O(m+n^2 \log n)\}$, where $m = \sum_i |T_i|$ and uses linear space. The randomized algorithm takes time $O(m \log^3 n)$ and uses linear space. The previous best for this problem was an 1981 $O(mn)$ algorithm by Aho et al.

Our faster deterministic algorithm uses a new efficient algorithm for the following interesting dynamic graph problem: Given a graph G with $n$ nodes and $m$ edges and a sequence of $b$ batches of one or more edge deletions, then after each batch, either find a new component or determine that there is no such component. For this problem, we have a simple algorithm with running time $O(n^2 \log n + b_0 \min\{n^2, m \log n\})$, where $b_0$ is the number of batches which do not result in a new component. For our particular application, $b_0 \leq 1$. If all edges are deleted, then the best previously known algorithm requires time $O(m\sqrt{n})$ to solve this problem.
**Key words:** Algorithms, Data Structures, Evolutionary Biology, and Theory of Databases.

## 1 Introduction

We are interested in the following problem which arises in computational biology and the theory of relational data bases.

**Input:** A set $\mathcal{T} = \{T_1, T_2, \ldots, T_k\}$ of rooted binary trees, each $T_i$ leaf-labelled by a subset $\mathcal{L}(T_i) \subset \{1, 2, \ldots, n\}$.

**Question:** Does there exist a tree $T^*$ such that for every $i$, $T^*|\mathcal{L}(T_i)$ is homeomorphic to $T_i$, where $T^*|\mathcal{L}(T_i)$ denotes the subtree of $T^*$ induced by the leaves of $\mathcal{L}(T_i)$ and their ancestors?

*Department of Computer Science, Cornell University, Ithaca, NY. This work was done in part which visiting at the International Computer Science Institute, Berkeley, CA. Email: mhr@cs.cornell.edu.

†Department of Computer Science, University of Victoria, Victoria, BC. Email: val@csr.uvic.ca.

‡Department of Computer and Information Science, University of Pennsylvania Philadelphia, PA. Email: tandy@central.cis.upenn.edu. This work was supported in part by a National Young Investigator Award from NSF.