QuickTree: A Deterministic Algorithm for the Building of Binary Phylogenetic Trees in $O(hn)$ Time

James Bieron

September 24, 2014
Abstract

I present an algorithm, deterministic quicktree, which solves the Rooted Triplet Consistency Problem with dense, consistent input. I prove that this algorithms runs in $O(hn)$ time, where $n$ is the number of leaves in the tree and $h$ is the height of the resulting tree.

1 Introduction

In this paper I discuss a new algorithm for solving the Rooted Triplet Consistency Problem with dense and consistent input. Some may find some portions of this paper to be overly detailed, which I concede is a valid criticism. I recommend that those reading this who are already familiar with the Rooted Triplet Consistency Problem read all of the boxed algorithms first, and then read the proofs as necessary. This report is based off of a project done for an undergraduate course, and thus little is assumed to appear obvious to the reader. Section 2 presents the problem formally, and describes my notation. Section 3 presents the algorithm, called QuickTree, which uses top down partitioning of the set of leaf elements to find the solution in $O(hn)$ time where $h$ is the height of the resulting tree, assuming that two pivots can be found such that they are on opposite sides of the root. I shall prove that partitioning correctly at every step of QuickTree will produce a correct solution, by first proving it produces a tree, and then that the tree satisfies the input constraints. Then, I shall give a linear time algorithm for pivot selection, and prove it is correct and runs in $O(n)$ time. Finally, I shall conclude by proving that QuickTree has running time $O(hn)$ for all possible consistent, dense inputs. A brief discussion of further work appears just before the acknowledgements.

2 The Problem

For the purposes of this paper, let a p-tree be a binary tree with only leaf nodes labeled, and every non-leaf node having two children. Let $C$ be a set of constraints of the form $(ij|k)$, which implies that in a tree that satisfies the constraint, $i$ and $j$ must have a common ancestor that is a proper descendent of the lowest common ancestor of $i$, $j$, and $k$. Let $C$ be called consistent if there exists a tree that satisfies all of the constraints in $C$. Let $C$ be called dense if for every combination of $i,j,k$ there exists a constraint in $C$. Thus, a dense set of constraints for a p-tree with $n$ elements (leaves) has $\frac{n(n-1)(n-2)}{6}$, or $(\binom{n}{3})$, constraints. Let us consider the problem of building a p-tree with $n$ leaves given a dense, consistent set of input constraints $C$. It is known [1] that such a tree can be found in $O(n^3)$ time. As the input has $O(n^3)$ constraints, it is impossible to do better than $O(n^3)$ with an algorithm that examines every constraint $C$. Observe that if you can partition the elements into two sets, which represent the sets of elements on each side of the root, you can then solve the problem for those two sets independently, and each resulting tree will be a subtree of the root. This applies recursively. I shall prove this formally below.

3 The Algorithm - Definition and Correctness

Let $Elements$ be a list containing a list of labels for leaves, and let $Constraints$ be a consistent, dense set of constraints such that $C[i,j,k]$ returns whichever of $i,j,k$ is such that the lowest common ancestor of the other two is below the lowest common ancestor of the three.
Definition 1  Let the descendants of a node be defined as all of the elements in the subtree rooted at that node. Thus, a leaf node can be said to have only a single descendant, the element it is labeled with. An interior node’s descendants are the union of the descendants of the left and right children of the node.

Definition 2  Let a p-tree be a binary tree with only leaf nodes labeled, and every non-leaf node having two children which are each either a leaf or the root of a p-tree. Optionally, let the interior nodes contain a list of all leaves that are descendants of the node.

Definition 3  Let a constraint be said to cross the root if two of its elements are on one side of the root, and the third element on the other side of the root.

Note that ”crossing the root” can be applied to any root of a subtree in the resulting tree, not just the top root of the entire tree. For a constraint to ”cross the root,” all three elements pertaining to the constraint must be descendants of the ”root.”

```
QuickTree(Elements E, Constraints C){
    if(|E| == 1) return a leaf node labeled E[0];

    using an oracle partition E into E1, E2 so that:
    E1 is all elements on the left side of the root
    of the tree consistent with C made up of elements E,
    and E2 is all elements on the right side of the root
    of the tree consistent with C made up of elements E
    create node r;
    set the left child of r to be OracleQuickTree(E1,C);
    set the right child of r to be OracleQuickTree(E2,C);
    return r;
}
```

Algorithm 1: An oracle based QuickTree

Theorem 1  If QuickTree is run with E as the set of all elements in a p-tree with constraints C, and C is consistent and dense, then Quicktree(E, C) will return the root of the p-tree with leaves E and satisfying constraints C.

Proof: First, I claim that it creates a p-tree. I prove this by strong induction:

Inductive Hypothesis: Quicktree(E, C) creates a valid tree for all |E| < k

Base Case: If |E| is 1 then it will return the p-tree that is only a single leaf. This is a p-tree.

Inductive case: It must be shown that assuming that Quicktree(E, C) creates a p-tree for all |E| < n that Quicktree produces a p-tree when |E| = n + 1

Proof: If |E| is 1 then it will return the tree that is only a single leaf. This is a p-tree. Otherwise, Quicktree will partition E into two non-empty sets, L and R, where L is the elements that are descendants of the left child of the interior node and R is the elements that are descendants of the right child of the interior node. Because L and R are non-empty and partition E, a set of cardinality \( n + 1 \), L and R are of cardinality at most \( n \). Thus by the inductive hypothesis, Quicktree(L, C) and Quicktree(R, C) will be p-trees. Thus, Quicktree will return a interior node with two children each of with are roots of p-trees. Being an interior node with two children, each of which are the roots of valid trees is sufficient to be a p-tree. Thus, by the
principle of mathematical induction, it is proven that Quicktree returns a p-tree.
Secondly, I claim that the p-tree created by Quicktree is consistent with all constraints in C. I shall do this by proving that at every point, the algorithm chooses an option that does not violate any constraints. I prove this by recursive definition.
Proof: Consider the root. Consider that there must exist a partitioning into L and R such that all of the constraints that include elements on both sides of the root are satisfied. If this was not the case, then C was not consistent. Because we took our partitioning from an oracle, we assume that we we chose such a partitioning. Thus, all constraints that contain elements on opposite sides of the root are satisfied. Observe that constraints whose elements are all on the same side of the root are unaffected by whether the three elements are in L or R, so none of those constraints are violated by the partitioning done at the root. Now, recursively apply this to the children of the root. All constraints that cross each subchild of the root are satisfied for the same exact reasons. Now, observe that every constraint must cross over interior nodes by the definition of the p-tree, so every identity is satisfied.

Claim 1 There exists a linear time algorithm that can partition E into E_1 and E_2 where E_1 and E_2 partition E correctly so that E_1 is all elements on one side of the root, and E_2 all elements on the other side of the root.

Before I prove this claim, I must make a few very obvious but nonetheless important observations, and prove one claim.

Observation 1 Unless a node is a leaf, it has at least one leaf in its left subtree and one leaf in its right subtree.

Observation 2 For any node x in a p-tree of at least 2 elements there is at least one element that is on the other side of the root.

Observation 3 If x and y are on opposite sides of an interior node, and e is also a descendant of this interior node, then C[x, y, e] cannot be e.

Claim 2 Given any x ∈ E where |E| = n > 1 and a set of dense, consistent constraints C, an element y ∈ E can be found such that x and y are on opposite sides of the root of the p-tree made up of nodes E and consistent with C. Furthermore, this can be done in O(n) time.

Proof: Consider the following algorithm:

```python
pivotFinder(E, C){
    x = the first element of E;
    y = the second element of E;
    for each remaining element e in E
        if xy|e
            y=e
    return (x, y)
}
```

Algorithm 2: Linear time pivot selection
It is clear that pivotFinder runs in $O(|E|)$: it has only a single loop with $|E| - 2$ iterations and all operation require constant time. Thus, it only remains to show that $(x, y)$ are definitely on opposite sides of the root. First, note that at least one element in $E$ is on the opposite side of the root compared to $x$. Second, observe that if $y$ is on the opposite side of the root as $x$ then $xy|e$ will never be true, and thus when the loop terminates $(x, y)$ will be on opposite sides of the root. Finally, if $(x, y)$ are on the same side of the root, then $xy|e$ will be true if $e$ is on the opposite of the root as $x$, and thus $y$ will be set to $e$, an element on the opposite side of the root as $x$, and $(x, y)$ will remain the same for all following iterations. Thus, $(x, y)$ is a pair of elements on opposite side of the root.

Now I shall prove Claim 1.

```plaintext
partition(Elements E, Constraints C) {
    (x, y) = pivotFinder(E, C);

    let E1, E2 be sets of elements
    add x to E1
    add y to E2
    for each element e in E not including x or y
        if xe|y
            add e to E1
        if ye|x
            add e to E2
}
```

Algorithm 3: Partitioning in QuickTree

Proof: Consider the above algorithm. This algorithm clearly runs in linear time, as it contains pivotFinder and a single for loop with $|E| - 2$ iterations. Storing the constraints as a table allows constraint checks to be done in constant time. We assume that adding nodes to lists is constant time. Thus, it remains to prove that $L$ and $R$ are in fact a partitioning of $E$ and that this partitioning does not violate any constraints that cross over the root of p-tree being built by Quicktree. That is partitions $E$ into $L$ and $R$ that are non-empty is clear. $x$ is explicitly put into $L$ and $y$ into $R$ so the sets are both non-empty. Only one of $(\text{C}[x, y, e] == y)$ or $(\text{C}[x, y, e] == x)$ can be true, so no element is added to both sets. It cannot be the case that $(\text{C}[x, y, e] == e)$ by observation 3. Thus, it is a partitioning of $E$.

That it violates none of the constraints in $C$ is surprisingly easy to show. First, observe that because $C$ is consistent, there exists some partitioning such that all constraints crossing the root will be satisfied. Next, observe that $x$ and $y$ must be on opposite sides of the root by the correctness of pivotFinder. Now, for each element $e$ we must choose whether to put it with $x$ or $y$. We know that there must exist a correct decision because of the consistency of $C$. If we choose differently than the algorithm above, then we will definitely contradict the constraint $\text{C}[x, y, e]$. Therefore, the only option at any stage is to put it into the set $L$ or $R$ as the algorithm does. Now, assume that after the partitioning is done, one of the constraints has been violated. This implies that $C$ is inconsistent, because if we change any $e$ between $L$ and $R$ we will definitely violate the condition $C[x, y, e]$. Thus, the only alternative to the partitioning done by this algorithm are ones that definitely violate constraints in $C$, and thus either our partitioning violates no constraints or $C$ is inconsistent.
4 Deterministic Quicktree

```
Quicktree(Elements E, Constraints C)
if(|E| == 1) return a leaf node labeled E[0]

(x,y) = pivotFinder(E,C)

let E1, E2 be sets of elements
add x to E1
add y to E2
for each element e in E not including x or y
    if xe|y
        add e to E1
    if ye|x
        add e to E2
create node r;
set the left child of r to be OracleQuickTree(E1,C);
set the right child of r to be OracleQuickTree(E2,C);
return r;
}
```

Algorithm 4: Final version of QuickTree

We simply replaced the oracle from Theorem 1 with the algorithm from Claim 1. The proof in Theorem 1 still holds, because the oracle behaved exactly like the partitioning algorithm that I have shown. The use of the oracle allowed a simpler, cleaner proof for Theorem 1, but now we are ready to define QuickTree in a fully deterministic way. Its correctness follows directly from Theorem 1, and its time complexity will be the topic of Theorem 2.

**Theorem 2** Quicktree builds a p-tree with n leaves in \( O(h \cdot n) \) time where \( h \) is the height of the resulting p-tree.

**Proof:** The fact that it builds a consistent p-tree has been proven in Section 2. What remains is to prove that the running time is \( O(h \cdot n) \). I shall consider the cost of computation spread among each interior node. For any interior node, the cost of the evaluation of the recursive call associated with that node is exactly the number of elements in the subtree rooted at that node. This is clear, because the algorithm runs some constant time operations, a for loop with |E| iterations, and pivotFinder which runs in linear time with respect to the size of \( E \). Thus, at each interior node, we do at most \( O(n) \) work.

Consider that \( L \) and \( R \) form a partition of \( E \), and thus any element of \( E \) is in exactly one of \( L \) and \( R \). Thus, the total work done in the two children of the root can be at most equal to the work done at the root. This also applies recursively, so at any level of the tree, the work done is at most \( O(n) \) where \( n \) is total number of leaves in the tree. Thus, the total work across all nodes is \( O(h \cdot n) \) plus constant time work for each of the \( n \)
leaves. In total, that is $O(h \cdot n)$ time to build the tree.

It should be noted that $h$ ranges between $O(\log_2 n)$ and $O(n)$, so this algorithm will run in somewhere between $O(n^2)$ and $O(n \log_2 n)$. An example of $O(n \log_2 n)$ behavior would be building a balanced tree, whereas an example of $O(n^2)$ behavior would be building a completely unbalanced tree, where every level has all but one element in a single subtree.

5 Future Work

I hope to extend this algorithm to the non-dense case of the Rooted Triplet Consistency Problem. I theorize that this should be done by considering separately the case where there are more than $n^2$ triplets given, and the case when there are less than $n^2$ triplets given. It is my belief that for the case of nearly dense sets of input, some simple modification of the algorithm will yield average case $O(hn)$ and worst case $O(n^2)$ behavior, while for the case of very sparse (less than $n^2$ triplets) significant modifications will be needed.

Acknowledgements

I would like to thank Anke van Zuylen and Ian Sturdy for their advice and counterexamples to the many false conjectures I made along the way. I would also like to thank Nathan Schaaf for his help with editing. Lastly, I would like to thank the College of William and Mary Department of Mathematics for their guidance and direction.

Bibliography

References
