Efficient implied alignment

Alex J. Washburn* and Ward C. Wheeler

*Correspondence: academia@recursion.ninja
Division of Invertebrate Zoology, American Museum of Natural History, 200 Central Park West, 10024-5192 New York, NY, USA

Abstract

Background: Given a binary tree $T$ of $n$ leaves, each leaf labeled by a string of length at most $k$, and a binary string alignment function $\otimes$, an implied alignment can be generated to describe the alignment of a dynamic homology for $T$. This is done by first decorating each node of $T$ with an alignment context using $\otimes$, in a post-order traversal, then, during a subsequent pre-order traversal, inferring on which edges insertion and deletion events occurred using those internal node decorations.

Results: Previous descriptions of the implied alignment algorithm suggest a technique of “back-propagation” with time complexity $O(k^2 \ast n^2)$. Here we describe an implied alignment algorithm with complexity $O(k \ast n^2)$. For well-behaved data, such as molecular sequences, the runtime approaches the best-case complexity of $\Omega(k \ast n)$.

Conclusions: The reduction in the time complexity of the algorithm dramatically improves both its utility in generating multiple sequence alignments and its heuristic utility.

Keywords: Dynamic homology, Implied alignment, Multiple string alignment, Phylogenetics, Sequence alignment, Tree alignment

Background

Implied Alignment (IA) was proposed by [1] as an adjunct to Direct Optimization (DO) [2, 3] to be used in phylogenetic tree search to provide both verification and more rapid heuristic analysis. The method was originally implemented in later versions of MALIGN [4] and has been a component of POY [5–9] since its inception. A more formal description of the algorithm was presented in [6] and [2]. Although originally designed for alignment-free phylogenetic analysis (dynamic homology, [10]), the procedure was first used as a stand-alone multiple sequence alignment (MSA) tool by [11] in their analysis of skink systematics.

IA was originally described in the context of parsimony-based phylogenetic analysis and was later extended to probabilistic model-based approaches by [12] and its implementations were described by [9, 13]. Similar MSA approaches also based in probabilistic analysis have been described e.g by [14] and [15], and implemented in PRANK [16]. Whiting et al. found that IA was superior (in terms of tree optimality score) to other
MSA methods in both parsimony and likelihood analyses. This observation has been repeated multiple times (e.g. [17–20]; summarized in [21]). The use of IA as an MSA algorithm as well as its use in the “static approximation” procedure [22] benefits greatly from improvements in the time complexity we present in this paper.

In a broader context, IA is a heuristic solution to the NP-hard Tree Alignment Problem (TAP) defined by [23]. As such, any individual IA is not guaranteed to be either optimal or unique, with potentially an exponential number of equally optimal implied alignments for any given binary tree.

The IA algorithm presented in this paper takes a different intellectual approach to deriving alignments than earlier versions of IA. Previous algorithmic approaches relied on DO assigning median sequences to the graph vertices. These sequences were then consumed by IA to produce the full alignment. Here, we describe IA as assigning “preliminary contexts” to the graph vertices, and later consuming these contexts to produce the median sequences and the full alignment.

The algorithm uses the repeated application of a pairwise string alignment function to perform an efficient MSA for a given binary tree whose leaves are labeled by strings, i.e. the tree describes the relationship of those strings. The more similar the initial leaf labelings the better the algorithm performs. Thus, while this algorithm has general use for performing an MSA, it is especially well-suited for the alignment of biological sequences where the strings are highly similar and a binary tree describing the strings’ relationships can be provided. Below, we provide an example of the IA algorithm’s performance on biological data.

**Definition of the heuristic function**

In order for an MSA to be inferred, there are constraints on the heuristic alignment function used to decorate the tree prior to performing the IA algorithm. As long as these constraints are satisfied, the implementation details of the function are agnostic to the IA algorithm presented here.

Let $\Sigma$ be a finite alphabet of symbols such that $|\Sigma| \geq 3$. Let $(-) \in \Sigma$ be a gap symbol, which will have a special meaning in the context of an alignment. Let $\mathcal{P}_{\geq 1}(X)$ denote the powerset of $X$, minus the empty set. Let $\Sigma^r_1$ be the alphabet of the following symbols:

$$
\Sigma^r_1 = \begin{align*}
\text{BOTH} & \quad \mathcal{P}_{\geq 1}(\Sigma) \\
\text{LEFT} & \quad \mathcal{P}_{\geq 1}(\Sigma) \\
\text{RIGHT} & \quad \mathcal{P}_{\geq 1}(\Sigma) \\
\text{GAPPED} & 
\end{align*}
$$

That is, $\Sigma^r_1$ contains all pairs of elements of $\mathcal{P}_{\geq 1}(\Sigma)$ tagged as BOTH, all elements of $\mathcal{P}_{\geq 1}(\Sigma)$ tagged as LEFT, all elements of $\mathcal{P}_{\geq 1}(\Sigma)$ tagged as RIGHT, and an additional element GAPPED. This construction of $\Sigma^r_1$ extends the original alphabet $\Sigma$ to preserve alignment information in the algorithms presented below. Note that if $|\Sigma| = x$ then $|\Sigma^r_1| = 2^x$. This follows from the fact that $|\mathcal{P}_{\geq 1}(\Sigma)|$ is equal to one less than the size of the power set of $\Sigma$, due to $\mathcal{P}_{\geq 1}(\Sigma)$ disallowing the empty set.

Let $\Sigma^r_{1}^{*}$ be the set of all finite strings over the alphabet $\Sigma^r_1$. Let $\otimes : \Sigma^r_{1}^{*} \times \Sigma^r_{1}^{*} \rightarrow (\mathbb{R}_{\geq 0}, \Sigma^r_{1}^{*})$ be a heuristic function that returns a nonnegative alignment cost and an alignment result in $\Sigma^r_{1}^{*}$. It is required that $\otimes$ be commutative but it need not be associative.
Both of these constraints will be explored later in the “Discussion” Section. These constraints are necessary but not sufficient for a heuristically optimal implied alignment to be inferred on the alignment function.

It is worth noting the motivation of the constructions defined above. Most pairwise string alignment functions take two finite strings of symbols from the original alphabet and supply a new finite string of symbols from the original alphabet. We can represent this class of pairwise string alignment functions by letting $\Sigma^*$ be the set of all finite strings over the alphabet $\mathcal{P}_{\geq 1}(\Sigma)$ and letting $\odot : \Sigma^* \times \Sigma^* \to (\mathbb{R}_{\geq 0}, \Sigma^*)$. The results of $\odot$ can contain cases of ambiguity where it cannot be inferred which input elements correspond to which output elements, but the construction of $\otimes$ never produces these cases of ambiguity due to the tagging of each element. The preservation of this non-ambiguous relationship between input and output is required for the algorithmic improvements presented below.

**Overview of the implied alignment algorithm**

The IA algorithm provides an MSA for a binary tree $T$ of $n$ leaves, each leaf containing a string with symbols in $\Sigma$ and length at most $k$. To generate this alignment, we will traverse $T$ twice. First, we perform a post-order traversal—from the leaves to the root—assigning the results of $\otimes$ as a “preliminary context” decoration to each node. Second, we perform a pre-order traversal—from the root to the leaves—aligning each preliminary context with its parent to assign a “final alignment” decoration to each node.

Using a binary string alignment function (like $\otimes$) to produce an MSA efficiently relies on the output of binary operations combined across the “global scope” of $T$. However, at each step in the post-order traversal, the only information known at a given node is the information contained in its subtree. Therefore, information for the entirety of $T$ is only known at completion of the post-order traversal. When performing the subsequent pre-order traversal, we take the “complete” scope available at the root node and thread the information towards the leaves. At each pre-order step, we take the “complete” context threaded from the root and combine it with the preliminary context derived during the post-order pass to assign the final alignment on that node. Thus, we collect all requisite information for an MSA during the post-order traversal and then apply that information during the pre-order traversal to derive the MSA.

As noted above, the time complexity of the IA algorithm’s pre-order traversal in previous work was $O(k^2 \cdot n^2)$. We are able to improve this by, during the post-order pass, tagging each element of a string on a node $v_x$ with information that notates on which subtree of $v_x$ that element originated. We can lift each symbol in $\Sigma$ into $\Sigma^+_G$ through the alignment process. Upon completion of the post-order traversal, each node in $T$ will have a string in $\Sigma^+_G$. Each element of said strings are tagged with one of four options enumerated above, representing from which child node the information of that element originated, relative to that node. Those tagged BOTH originated from both subtrees, those tagged LEFT originated from the left subtree, those tagged RIGHT originated from the right subtree, and those tagged GAPPED originated from neither subtree (i.e., elsewhere in $T$). Because elements tagged GAPPED originated from neither subtree, GAPPED those elements cannot be created during the post-order, only derived during the pre-order.

This tag on each element provides contextual information that allows for an efficient processing of the elements in the pre-order traversal. During the pre-order traversal, a node’s preliminary context is “zipped” with the parent’s alignment in order to derive its
We will show that this tagging and “zipping” process is a substantial improvement over previous work, reducing the time complexity from quadratic to linear in the length of the strings. It is worth noting that this tagging can be represented as a succinct data structure per [24], requiring only two additional bits per element.

**An example heuristic function**

We will provide an example definition of ⊗ in Algorithm 1 sufficient for the IA algorithm, though there are other sufficient definitions of ⊗. The candidate function fitting the description of ⊗ we present will be defined as a extension of the Needleman-Wunsch [25] algorithm for pairwise string alignment. The algorithm is modified along the same the lines that DO modified the dynamic programming technique of [26], with an additional step taken to produce the tagged elements in the output alignment. Algorithm 1 (described below) is used to generate the results presented in the “Methods” section.

**Algorithm 1** Example ⊗ definition

*Require:* lhs, rhs ∈ Σ^n
*Result:* (R_0, Σ^n)

1. function ⊗(lhs, rhs)
   ▶ Conditional here are required to ensure commutativity of ⊗.
   2. if lhs = rhs then
   3. return lhs
   4. else if lhs < rhs then
   5. return MATRIXTRACEBACK(σ, lhs, rhs) ▶ See Algorithm 4
   6. else
   7. result ← MATRIXTRACEBACK(σ, rhs, lhs) ▶ See Algorithm 4
   8. return swapCONTEXTS(result) ▶ See Algorithm 2

**Algorithm 2** Swap the LEFT and RIGHT contexts of s ∈ Σ^n

*Require:* inputString ∈ Σ^n
*Result:* Σ^n

1. function swapCONTEXTS(inputString)
2. i ← |inputString| − 1
3. while i ≥ 0 do
4. switch inputString.i do
5. case LEFT v
6. inputString.i ← RIGHT v
7. case RIGHT v
8. inputString.i ← LEFT v
9. i ← i − 1
return inputString

First, we decide deterministically which of the two input strings is assigned to the top (columns) of the alignment matrix and which string is assigned to the left side (rows). We assign the input strings based on the data they contain. The longer string is assigned to the columns of the alignment matrix, the shorter string to the rows. If the strings are
the same length, we take the first string under the lexical ordering of their elements and
assign it to the columns and assign the second string to the rows of the alignment matrix.
In the case that the strings are identical, the alignment is trivial. If the first string sup-
pired to $\otimes$ was not assigned to the rows of the matrix, then we must swap the LEFT and
RIGHT tags of the resulting string alignment before returning the result. This consistency
in assignment ensures the commutativity of $\otimes$, which is necessary to enforce consistency
of the IA algorithm. Commutativity of $\otimes$ ensures that the IA algorithm provides the same
alignment results for isomorphic tree labeling (i.e. ensures label invariance).

We now apply a memoized update procedure [27], a common element of dynamic
programming algorithms such as the Needleman-Wunsch alignment. The subsequent
“traceback,” however, is notably modified from the original Needleman-Wunsch pro-
dedure. The upward, leftward, and diagonal directional arrows used to produce the
alignment are additionally used to tag each element as LEFT, RIGHT, or BOTH, re-
spectively. These tagged pairwise alignments will be consumed on the subsequent pre-order
traversal of $T$ when merging preliminary contexts. Storing this information for each ele-
ment of the pairwise alignment allows a more efficient generation of the subsequent
multiple string alignment, allowing for an asymptotic improvement over the previous IA
algorithm. This additional tagging detail is the key difference between previous alignment
methods and the one presented in this paper.

The example $\otimes$ presented in Algorithm 1 is of $\Theta(k^2)$ complexity in both time and space,
where $k$ is the length of the longer string. For clarity, while this example function is pre-
sented as a modification of the well understood Needleman-Wunsch algorithm (without
explicit memoization), this tagging approach can be incorporated into more sophisti-
cated pairwise string alignment algorithms. For instance, by using the method described
by [28], this algorithm’s time complexity could be improved to $O(k \times s)$, where $s$ is the
edit distance between the strings. Alternatively, by using the method described by [29],
this algorithm could be improved to use $O(k)$ space. Affine gap models [30] can also be
incorporated via the method of [2].

The operator $\sigma : \mathcal{P}_{\geq 1}(\Sigma) \times \mathcal{P}_{\geq 1}(\Sigma) \to (\mathbb{R}_{\geq 0}, \mathcal{P}_{\geq 1}(\Sigma))$ presented in Algorithms 1, 3,
and 4 represents a metric for determining the transition cost between symbols in $\mathcal{P}_{\geq 1}(\Sigma)$.
The metrics used in our data sets can be found in Table 1. The metrics presented in
Table 1 show the transition cost between elements of $\Sigma$. However, these metrics can
be expanded to define the transition costs between elements of $\mathcal{P}_{\geq 1}(\Sigma)$ in the manner
described by [2]. Note that $\sigma$ can also be a more complex metric than those presented
here, for instance a metric with affine or logarithmic affine gap costs, and be compatible
with the IA algorithm. For usage of $\sigma$, see Algorithms 1, 3, and 4.

| $\sigma_0$ | A | C | G | T | – | $\sigma_1$ | A | C | G | T | – | $\sigma_2$ | A | C | G | T | – |
|-----------|---|---|---|---|---|-----------|---|---|---|---|---|-----------|---|---|---|---|---|
| A         | 0 | 1 | 1 | 1 | 2 | A         | 0 | 3 | 3 | 3 | 1 | A         | 0 | 1 | 1 | 1 | 1 |
| C         | 1 | 0 | 1 | 1 | 2 | C         | 3 | 0 | 3 | 3 | 1 | C         | 1 | 0 | 1 | 1 | 1 |
| G         | 1 | 1 | 0 | 1 | 2 | G         | 3 | 3 | 0 | 3 | 1 | G         | 1 | 1 | 0 | 1 | 1 |
| T         | 1 | 1 | 1 | 0 | 2 | T         | 3 | 3 | 3 | 0 | 1 | T         | 1 | 1 | 1 | 0 | 1 |
| –         | 2 | 2 | 2 | 2 | 0 | –         | 1 | 1 | 1 | 1 | 0 | –         | 1 | 1 | 1 | 1 | 0 |

Expansion of the metrics presented in Table 1 is described by [2].
Description of post-order traversal

The post-order traversal (leaves to the root) of the binary tree $T$ is a straightforward procedure, see Algorithm 6. We assign preliminary contexts and costs to each node, $v_x$, of $T$. These preliminary contexts will be consumed to assign a final alignment in the subsequent pre-order traversal of the tree. The post-order traversal described here is very similar to the DO post-order traversal described by [1], differing only in the use of $\otimes$ which captures the preliminary context of a subtree, instead of generating a preliminary median string assignment.

First, for each leaf node, $v_x$, we set $v_x.cost$ to 0. Additionally, if $v_x$ is of type $\Sigma^*$ and not of type $\Sigma_x^i$—i.e. if it has been decorated with a finite string of symbols from the alphabet $\Sigma$, and it is not decorated with a finite string of preliminary contexts over the alphabet $\Sigma_x^i$—then we call $\text{INITIALIZESTRING}(v_x, \text{prelimString})$ to apply the transformation $\Sigma^* \rightarrow \Sigma_x^i$.

On each internal node, $v_y$ with children $v_l$ and $v_r$, of $T$, we call $v_l \otimes v_r$. The resultant $\text{prelimString}$ is assigned to $v_y, \text{prelimString}$, and the sum of the $v_l.cost$, $v_r.cost$, and the alignment cost of $v_l \otimes v_r$ is assigned to $v_y.cost$. By performing this operation in a post-order traversal over $T$, we propagate the preliminary contexts and costs returned from the calls to $\otimes$ from the leaves to the root.

Algorithm 3: Generate the alignment matrix of $\otimes$, presented without explicit memoization

```
Algorithm 3 Generate the alignment matrix of $\otimes$, presented without explicit memoization

Require: lesser, longer $\in \Sigma_x^i$
 Require: $\sigma : \mathcal{P}_2(\Sigma) \times \mathcal{P}_2(\Sigma) \rightarrow (\mathbb{R}_{\geq 0}, \mathcal{P}_2(\Sigma))$
 Require: $i \in [-1, |\text{lesser}|-1] \in \mathbb{Z}$
 Require: $j \in [-1, |\text{longer}|-1] \in \mathbb{Z}$
 Result: $(\mathbb{R}_{\geq 0}, \text{Dir}, \Sigma)$
 1: A suitable memoization strategy should be applied to avoid repeated work of shared sub problems
 2: function $\text{MATRIXDEFINITION}(\text{lesser, longer, } \sigma, i, j)$
 3:  if $i < 0 \land j < 0$ then $\triangleright$ Outside of matrix is infinite cost
 4:    return $(\infty, \infty, \infty)$
 5:  else if $i = 0 \land j = 0$ then $\triangleright$ Handle the origin
 6:    return $(0, 0, 0)$
 7:  else if $j = 0 \land |\text{longer}, i-1| = -$ then $\triangleright$ Preserve input gap
 8:    $(\text{leftCost}, \text{lessForm}) \leftarrow \text{MATRIXDEFINITION}(\text{lesser, longer, } \sigma, i-1, j)$
 9:    return $(\text{leftCost}, \text{lessForm}, \infty)$
 10:  else if $i = 0 \land |\text{lesser}, i-1| = -$ then $\triangleright$ Preserve input gap
 11:    $(\text{aboveCost}, \text{lessForm}) \leftarrow \text{MATRIXDEFINITION}(\text{lesser, longer, } \sigma, i, j)$
 12:    return $(\text{aboveCost}, \text{lessForm}, \infty)$
 13:  else $\triangleright$ General recursive case
 14:    $z \leftarrow \sigma(\text{longer}, \text{lessForm}, i-1)$
 15:    $y \leftarrow \sigma(\text{longer}, \text{lessForm}, \infty)$
 16:    $x \leftarrow \sigma(\text{lessForm}, \text{lessForm}, \infty)$
 17:    $(\text{minCost}, \text{minDir}, \text{minElem}) \leftarrow \text{GETMINIMAL}(x, y, z) \triangleright$ See Algorithm 5
 18:    if $(\text{minDir}, \text{minElem}) = (\infty, \infty) \text{ then}$ $\triangleright$ Aligned gap is insertion
 19:      return $(\text{minCost}, \infty, \infty)$
 20:  else
 21:    return $(\text{minCost}, \text{minDir}, \text{minElem})$
```
Upon completion of the post-order traversal, each internal node contains the preliminary context information and the cost for the corresponding subtree. Consequently, when the post-order traversal is complete, the root node contains the preliminary context information of the full leaf set of strings and the alignment cost for the entire tree $T$. In the pre-order traversal, we will consume this preliminary context to perform an (efficient) alignment on the strings.

Because the post-order traversal can be performed using any valid definition of $\otimes$, the complexity of the post-order traversal is dependent on the complexity of the heuristic alignment function used. Let the complexity of $\otimes$ be defined as $H(k)$, where $k$ is the maximum string length of the leaf labels of the tree $T$. Then post-order traversal runs in $O(H(k) \ast n)$ time and space, where $n$ is the number of leaves in the binary tree $T$. If we were to use Ukkonen’s method with the $\otimes$ described in Algorithm 1, the post-order traversal would run in $O(k \ast s \ast n)$ time and space, where $s$ is the maximum edit distance between any two strings.

**Description of pre-order traversal for final alignments**

The pre-order traversal (from the root to the leaves) of the binary tree $T$ consumes the preliminary context decorations on each node created in the post-order traversal in order to assign final alignment decorations of $\Sigma_T^*$ to each node, see Algorithm 7. First, the root node must be initialized for the pre-order traversal by assigning the root’s preliminary context to the root’s final alignment. By initializing the root node in this manner, the root node is consistent with the treatment of any other parent node when deriving the internal node alignments in Algorithm 8.

---

**Algorithm 4 Consume the alignment matrix of $\otimes$, returning the alignment and cost**

Require: $\text{lesser, longer } \in \Sigma_T^*$

Require: $\sigma : \mathcal{P}_n(\Sigma) \times \mathcal{P}_n(\Sigma) \rightarrow (R_{\delta_0}, \mathcal{P}_n(\Sigma))$

Result: $(R_{\delta_0}, \Sigma_T^*)$

1. A suitable memoization strategy should be applied to avoid repeated work of shared sub problems
2. function $\text{MATRX\_TRACEBACK(lesser, longer)}$
3. (i, j) $\leftarrow (|\text{lesser}| - 1, |\text{longer}| - 1)$
4. $(\text{alignedCost, ..}) \leftarrow \text{MATRX\_DEFINITION(lesser, longer, i, j)}$ $\triangleright$ See Algorithm 3
5. $\text{alignedStr } \leftarrow []$
6. while $(i, j) > (0, 0)$ do
7. $(.., \text{dirArrow}, \text{elem}) \leftarrow \text{MATRX\_DEFINITION(lesser, longer, i, j, .., dirArrow, elem)}$ $\triangleright$ See Algorithm 3
8. switch $\text{dirArrow}$ do
9. case $\leftarrow$
10. $(i, j) \leftarrow (i + 1, j - 1)$
11. $\text{nextElement } \leftarrow \text{DELETE elem longer}_{i,j}$
12. case $\uparrow$
13. $(i, j) \leftarrow (i - 1, j + 1)$
14. $\text{nextElement } \leftarrow \text{INSERT elem lesser}_{i,j}$
15. case $\wedge$
16. $(i, j) \leftarrow (i - 1, j - 1)$
17. $\text{nextElement } \leftarrow \text{ALIGN elem longer}_{i,j} \text{ lesser}_{i,j}$
18. $\text{alignedStr } \leftarrow \text{nextElement + alignedStr}$
19. return $(\text{alignedCost, alignedStr})$

---

Upon completion of the post-order traversal, each internal node contains the preliminary context information and the cost for the corresponding subtree. Consequently, when the post-order traversal is complete, the root node contains the preliminary context information of the full leaf set of strings and the alignment cost for the entire tree $T$. In the pre-order traversal, we will consume this preliminary context to perform an (efficient) alignment on the strings.

Because the post-order traversal can be performed using any valid definition of $\otimes$, the complexity of the post-order traversal is dependent on the complexity of the heuristic alignment function used. Let the complexity of $\otimes$ be defined as $H(k)$, where $k$ is the maximum string length of the leaf labels of the tree $T$. Then post-order traversal runs in $O(H(k) \ast n)$ time and space, where $n$ is the number of leaves in the binary tree $T$. If we were to use Ukkonen’s method with the $\otimes$ described in Algorithm 1, the post-order traversal would run in $O(k \ast s \ast n)$ time and space, where $s$ is the maximum edit distance between any two strings.
Algorithm 5: Get the minimal directional matrix context from the 3 inputs

Require: \((\text{diagCost, diagElem}) \in (\mathbb{R}_{\geq 0}, \Sigma_\text{C})\)

Require: \((\text{rightCost, rightElem}) \in (\mathbb{R}_{\geq 0}, \Sigma_\text{T})\)

Require: \((\text{downCost, downElem}) \in (\mathbb{R}_{\geq 0}, \Sigma_\text{T})\)

Require: Total ordering over directional arrows defined as: \(\text{\wedge} < \leftrightarrow < \rightarrow\)

Result: \((\mathbb{R}_{\geq 0}, \Sigma_\text{T})\)

1: function GETMINIMAL((diagCost, diagElem), (rightCost, rightElem), (downCost, downElem))

2: if diagCost \leq rightCost then
3:     if diagCost \leq downCost then
4:         return (diagCost, diagElem) \(\wedge\)
5:     else
6:         return (downCost, downElem \(\cup\) \(\rightarrow\))
7: else
8:     if rightCost \leq downCost then
9:         return (rightCost, rightElem \(\cup\) \(\leftrightarrow\))
10: else
11:     return (downCost, downElem \(\cup\) \(\rightarrow\))

Algorithm 6: Post-order Traversal

Require: A binary tree decorated with leaf labels inputString \(\in \Sigma_\text{T}^*\)

Result: A binary tree decorated with internal labels prelimString \(\in \Sigma_\text{T}^*\)

1: function POST-ORDER(node)
2: if isLeaf (node) then
3:     node.cost \leftarrow 0
4:     node.prelimString \leftarrow INITIALIZESTRING(node.prelimString)
5: else
6:     lhs \leftarrow POST-ORDER(node.children.first)
7:     rhs \leftarrow POST-ORDER(node.children.second)
8:     (alignCost, alignContext) \leftarrow lhs.prelimString \& rhs.prelimString \(\triangleright\) See Algorithm 1
9:     node.cost \leftarrow alignCost + lhs.cost + rhs.cost
10: node.prelimString \leftarrow alignContext

Require: inputString \(\in \Sigma_\text{T}^*\)

Result: \(\Sigma_\text{T}^*\)

11: function INITIALIZESTRING(inputString)
12:     \(i \leftarrow |\text{inputString}| - 1,\)
13:     while \(i \geq 0\) do
14:         inputString, \leftarrow \text{ALIGN}(\text{inputString},) \text{inputString}, \text{inputString},
15:         \(i \leftarrow i - 1,\)
16: return inputString

Algorithm 7: Pre-order Traversal

Require: A binary tree decorated with node labels prelimString \(\in \Sigma_\text{T}^*\)

Result: A binary tree decorated with node labels finalString \(\in \Sigma_\text{T}^*\)

1: function PREORDER(node)
2: if isRoot (node) then \(\triangleright\) Initialize the root node
3:     node.finalString \leftarrow node.prelimString
4: else \(\triangleright\) Derive alignment for node
5:     parentFinal \leftarrow node.parent.finalString
6:     parentPrelim \leftarrow node.parent.prelimString
7:     childPrelim \leftarrow node.prelimString
8:     \(v \leftarrow \text{DERIVEALIGNMENT}(\text{isLeft}(node), parentFinal, parentPrelim, childPrelim)\)
9:     node.finalString \leftarrow v
10: PREORDER(node.children.first)
11: PREORDER(node.children.second)
For each non-root node, \( v_c \), we first determine whether \( v_c \) is the left or right child of its parent. This is required because \textsc{LEFT}-tagged elements originate from alignments of the left subtree and \textsc{RIGHT}-tagged elements originate from alignments of the right subtree, and we must use this information when deriving the final alignment of \( v_c \).

The final alignment of the parent of \( v_c \), \( v_p \), will necessarily be of greater than or equal length to \( v_p \)'s preliminary context, because \( v_p \)'s final alignment contains all the information from the contexts of \( v_p \)'s subtrees as well as the information from the rest of the tree, that is, the contexts of all of the subtrees of every ancestor node to \( v_p \). The preliminary context of \( v_p \) is also of greater than or equal length to preliminary context of \( v_c \), due to the \( v_p \)'s context containing all information from \( v_c \)'s context, plus the addition of \( v_c \)'s sister subtree. The resulting value assigned to \( v_c \)'s final alignment will have the same length as the final alignment assigned to \( v_p \). Since this invariant length is maintained from the root node to the leaf nodes' final alignment assignments, all alignments will have the same length. This constitutes a simple inductive argument that the final alignment assignment of each node will be of equal length and constitute a genuine string alignment.

The final alignment for \( v_c \) is derived by performing a “sliding zip” over \( v_p \)'s final alignment, \( v_p \)'s preliminary context, and \( v_c \)'s preliminary context. \( v_p \)'s final alignment is used as the basis of the zip. The inputs to this alignment are the preliminary contexts of \( v_p \) and \( v_c \) and the final alignment of \( v_p \). At each step of the “sliding zip,” one element of \( v_p \)'s final alignment will be consumed and one element of \( v_c \)'s final alignment will be defined. Additionally, at each step of the zip, one of: an element from \( v_p \)'s preliminary context, elements from both \( v_p \)'s and \( v_c \)'s preliminary contexts, or no elements from either node's context, will be consumed. Finally, we define an element of \( v_c \)'s final alignment to be either an element from \( v_c \)'s preliminary context or a gap. The process is called a “sliding zip” because, due to the varying lengths of the three inputs, the elements of \( v_p \)'s and \( v_c \)'s
preliminary contexts do not have an immediately apparent index with which they correspond to \( v_p \)'s final alignment, which is used as the basis of the zip. Rather, the elements of \( v_p \)'s and \( v_c \)'s preliminary contexts “slide” through the zipping process, and their corresponding indices with \( v_p \)'s final assignment is deduced dynamically. The logic applied in the “sliding zip” is to propagate gaps from the final alignment of \( v_p \), which contains the gaps of the entire tree above \( v_c \), down to \( v_c \), and when not dealing with a gap propagated from an ancestor node to \( v_c \), to align the non-gap elements or introduce a new gap to be propagated. The “sliding zip” process is often easier to understand by stepping through the algorithm. An example alignment of this “sliding zip” process for two internal nodes is shown in Fig. 1.

There are five cases determining the derivation of each index of \( v_c \)'s final alignment. The cases are presented in the pseudocode of Algorithm 8, in Fig. 1, and described below.

- **Case 0:** When the element of \( v_p \)'s final alignment is “GAPPED,” then the next element of \( v_c \)'s final alignment is “GAPPED.”
- **Case 1:** When the sliding zip has consumed all elements of the \( v_c \)'s preliminary context, then the next element of \( v_c \)'s final alignment is “GAPPED.” Because we only define the next element of \( v_c \)'s final alignment to be either an element from \( v_c \)'s preliminary context or a gap, the latter is the only choice.
- **Case 2:** When the element of \( v_p \)'s final alignment is “BOTH,” then we consume the next elements of both \( v_p \)'s and \( v_c \)'s preliminary contexts, and the next element of \( v_c \)'s final alignment is \( v_c \)'s consumed preliminary context element. Because \( v_p \)'s final alignment element was marked as an alignment event, we know that \( v_c \) was aligned with its sister subtree at this index, and that \( v_c \)'s preliminary context element is the correct element for this index of the alignment.
- **Case 3:** When both \( v_p \)'s final alignment element and \( v_p \)'s preliminary context element are “LEFT,” and \( v_c \) is the left child of \( v_p \), then we consume the next element of each of \( v_p \)'s and \( v_c \)'s preliminary contexts, and the next element of \( v_c \)'s final alignment is \( v_c \)'s consumed preliminary context element. Because LEFT-tagged elements originate from the left subtree of a node, and \( v_c \) is the left child of \( v_p \), \( v_c \)'s preliminary context element is the correct element for this index of the alignment. If the same LEFT-tagged elements were encountered but \( v_c \) was the right child of \( v_p \), then \( v_c \)'s preliminary context element would not be the correct element for this index of the alignment, because LEFT-tagged elements originate from the left subtree of \( v_p \) and the LEFT-tagged element under consideration was encountered in \( v_p \)'s right subtree. In the case that a LEFT-tagged element is encountered in the right subtree of \( v_p \), we introduce a new gap into all the alignments of the subtree at this index to account for the aligned element in the \( v_c \)'s sister subtree. This is implicitly dealt with in Case 5.
- **Case 4:** Conversely to Case 3, when both \( v_p \)'s final alignment element and \( v_p \)'s preliminary context element are “RIGHT,” and \( v_c \) is the right child of \( v_p \), we consume the next element of both \( v_p \)'s and \( v_c \)'s preliminary contexts and assign to the next element of \( v_c \)'s final alignment \( v_c \)'s consumed preliminary context element. Because RIGHT-tagged elements originate from the right subtree of a node, and \( v_c \) is the right child of \( v_p \), \( v_c \)'s preliminary context element is the correct element for this index of the alignment. If the same RIGHT-tagged element was encountered but \( v_c \) was the left child of \( v_p \), then \( v_c \)'s preliminary context element would not be the correct
element for this index of the alignment, because RIGHT-tagged elements originate from the right subtree of \( v_p \) and the RIGHT-tagged element under consideration was encountered in \( v_p \)'s left subtree. In the case that a RIGHT-tagged element is encountered in the left subtree of \( v_p \), we introduce a new gap into all the alignments of the subtree at this index to account for the aligned element in \( v_c \)'s sister subtree. This is implicitly dealt with in Case 5.

- Case 5: When none of the conditions for Case 0, 1, 2, 3, or 4 hold, then we consume the next element of \( v_p \)'s preliminary context and the next element of \( v_c \)'s final
alignment is “GAPPED.” This handles the cases where either the two subtrees were not aligned at the current index or a new gap needed to be introduced at the current index because a LEFT-tagged or RIGHT-tagged element was encountered in $v_p$’s right or left subtree, respectively.

**Analysis of pre-order traversal**

Let $m = \frac{k}{a}$, where $k$ is the length of the longest input string, and $a$ is the length of the root node’s preliminary context. In the best case that a “perfect alignment” is derived, that is, that each element of all the input strings can be aligned with one of the elements of the longest input string, then $m = 1$. In the worst case that a “degenerate alignment” is derived, that is, that no element of any of the input strings can be aligned with any of the elements of the longest input string, and all input strings are of equal length, then $m = n$.

The improvement of the implied alignment algorithm presented here compared to the original algorithm is that the additional stored information allows us to determine the final assignments in $\Theta(k \times m \times n)$ instead of $O(k^2 \times n^2)$ time. The aforementioned $n^2$ component occurred in previous implementations due to the use of a “back-propagation” technique, which required that, at each pre-order step, each new gap found in the alignment was retroactively applied to every alignment derived in a previous pre-order step. The $k^2$ component in the previous implementation was due to using a Needleman-Wunsch string alignment between the current node and its parent node at each pre-order step in addition to the alignment already performed at each post-order step. By saving the requisite information on the nodes during the post-order traversal and then consuming this information with the “sliding-zip” technique, we eliminate the Needleman-Wunsch alignments during the pre-order, as well as the back-propagation, and replace these computationally expensive operations with a much more efficient algorithm.

In the pre-order traversal algorithm presented above, we generate an implied alignment in $\Theta(k \times m \times n)$ time. We must perform a “sliding-zip” operation on each node in the binary tree $T$, hence the factor of $n$. The “sliding-zip” accounts for the $k \times m$ factor.

The best case time complexity occurs when the length of the derived alignment is the length of the longest input string, an alignment with the minimal number of elements. In this case, $m = 1$ and the “sliding zip” performed on each node performs work equal to the length of the longest input string $k$. Hence, the best case time complexity of the implied alignment algorithm is $\Omega(n \times k)$, occurring when the input strings are highly correlated and $m = 1$.

The worst case time complexity occurs when the length of the derived alignment is equal to the sum of the lengths of the input strings, an alignment with the maximum number of elements. In the worst case, $m \gg 1$, and the “sliding zip” performed on each node performs work equal to the length of the longest input string, $k$, multiplied by the number of input strings, $n$. Hence, the worst case time complexity of the implied alignment algorithm is $O(k \times n^2)$, occurring when the input strings are independent of each other.

**Methods**

An example Haskell implementation of the implied alignment algorithm described above, the data sets used to generate the results, along with a script to replicate the results discussed below can all be found here: https://github.com/recursion-ninja/efficientimplied-alignment/replicate-results.sh
We ran the implied alignment algorithm described in this paper on a pathological data set that was constructed to illustrate the best and worst case performances of the implied alignment algorithm. The pathological data set consisted of balanced binary trees which repeatedly doubled in size. The smallest tree is a quartet tree, with the strings consisting of a single symbol from the alphabet \( \Sigma = \{A,C,G,T\} \) repeated \( k \) number of times. The lengths of the strings on each leaf were repeatedly doubled in size to scale the string length. The size of the tree was scaled by taking \( 2^n \) quartet trees and combining them together into a larger balanced binary tree of \( n \) leaves.

The time complexity scaling of this pathological data set was examined under two different metrics, \( \sigma_0 \) and \( \sigma_1 \). The former metric preferentially selects substitution events over insertion or deletion elements, thus producing the “perfect alignment.” Conversely, the latter selects for insertion or deletion over substitution, thus producing the “degenerate alignment.”

Additionally, to explore the performance of the pre-order traversal on “real world” data, the algorithm was run on the fungal and metazoan biological data sets described by [31] and [32] respectively. Both full data sets consisted of a preselected tree and predetermined string alignment (i.e. including gaps). The full leaf set of the tree was repeatedly halved to produce a data set of doubling leaf set sizes. The string alignment was repeatedly truncated, dropping the beginning and end of the alignment, taking the central slice of the current length from each string, and then removing all the gaps from the alignment slice. The pruned trees and truncated strings were used as progressively doubling inputs, to measure runtime scaling in terms of both leaf set size and string length. Both biological data sets used the discrete metric \( \sigma_3 \) and the alphabet \( \Sigma = \{A,C,G,T,\sim\} \).

After running the algorithm on each data set, we constructed an Ordinary Least Square (OLS) model with the running time in milliseconds as a function of dimensions \( k \) and \( n \). We took the binary logarithm, \( \log_2 \), of both input dimensions as well as the output. From there, we calculated the coefficients of each input in this equation: \( \log_2(\text{runtime}) = \beta_0 + \beta_1 \log_2(n) + \beta_2 \log_2(k) + \epsilon \), where \( \epsilon \) represents the estimation error. Note that, because the logarithm of the inputs was taken, we would expect \( \beta_1 \) to be close to 1 for linear performance with respect to that input variable and close to 2 for quadratic performance. See Table 2.

A direct runtime comparison between the \( \mathcal{O}(k^2*n^2) \) algorithm in POY and our improved algorithm was not readily achievable due to being implemented in different impure and purely functional languages, which come with confounding architectural designs. Instead we present the empirical runtime analysis of the pre-order traversal.

| Table 2 Regression coefficients of leaf-set size and string length on runtime |
|-------------------------------|-------------------|-------------------|-------------------|-------------------|
| Dependent variable: | log\(_2\) (Runtime) |                      |                      |                      |
| Best | Worst | Fungi | Metazoa |                      |
| log\(_2\) (String count \(n\)) | 1.057 | 2.021 | 1.236 | 1.511 |
| log\(_2\) (String length \(k\)) | 0.920 | 1.120 | 0.836 | 1.038 |
| Observations | 49 | 49 | 42 | 42 |
| Adjusted R\(^2\) | 0.990 | 0.996 | 0.987 | 0.995 |
above. We did not thoroughly explore the implemented post-order traversal, as it does not deviate substantially from the well-understood Needleman-Wunsch algorithm. We have provided the reader a convenient https://github.com/recursion-ninja/efficient-implied-alignment/replicate-results.sh script in the aforementioned code repository to conduct their own analysis of both the pre-order and post-order traversals.

Results
The pathological data sets show the stark difference between the best case $\Omega(n \times k)$ and worst case $O(k \times n^2)$ performances. The OLS model empirically supports the theoretical best and worst cases demonstrated by the two runs on the pathological data set as shown in Figs. 2 and 3.

The OLS model also anecdotally supports the supposition that time complexity scales well for the biological data sets. The fungal and metazoan sequence data sets demonstrate a near-linear time complexities with respect to the number of input strings and linear complexity with respect to string length. The fungal data sets lend support to the argument that some of “real world” use cases can perform close to the theoretical best case complexity (see Figs. 4 and 5).

Conclusions
The IA algorithm can be improved to run with $O(k \times n^2)$ and best case $\Omega(k \times n)$ complexity of time and space. The more similar the input strings are, the closer the performance will be to the best case. When the algorithm is applied to “real world” biological sequences, the performance tends strongly towards the best case. The improved algorithm presented in this paper offers immediate and significant gains to applications related to the TAP and MSA.

Fig. 2 Best case pre-order runtime
Discussion
The algorithm originally described by [1] was given the name implied alignment to differentiate it from other methods (e.g. sum-of-pairs alignment) unconnected to the vertex string assignments “implied” by the binary tree on a given leaf-set. However, it is worth articulating exactly how the alignment we derive is implied by the tree. In short, it is the requirement of commutativity and the lack of associativity.
For the purposes of this analysis we will ignore the cost returned from the $\otimes$ and consider only the resulting alignment context. Therefore let $\oplus : \Sigma^*_n \times \Sigma^*_n \rightarrow \Sigma^*_n$ be defined as $\otimes$, but ignoring the alignment cost of the result. If we are given a rooted binary tree $T = ((A, B), (C, D))$ with leaves $A, B, C, D \in \Sigma^*$ then the ancestral state of the root node defined by the heuristic function $\oplus$ would be $((A \oplus B) \oplus (C \oplus D))$. In fact, the ancestral state of any internal node defined by $\oplus$ can be calculated by applying $\oplus$ recursively to the subtree of the internal node. The binary structure of the tree directly implies the precedence of each application of $\oplus$ in the final result. Since $\oplus$ need not be associative, the tree $((A, (B, C)), D)$ evaluated as $((A \oplus (B \oplus C)) \oplus D)$, is likely to yield different results. However, since $\oplus$ is commutative, transposing any child nodes between the left and right positions of their parent will result in a tree that yields the same internal values. For example consider a transposed tree $T'$:

$$
eval(T') = \eval((D, C), (B, A))$$
$$= ((D \oplus C) \oplus (B \oplus A))$$
$$= ((C \oplus D) \oplus (B \oplus A))$$
$$= ((C \oplus D) \oplus (A \oplus B))$$
$$= ((A \oplus B) \oplus (C \oplus D))$$
$$= \eval((A, B), (C, D))$$
$$= \eval(T)$$

This commutative property and lack of an associative property precisely determines that the alignment is implied by the tree on the leaf-set under $\oplus$ and not the unique alignment on all trees for the leaf-set under $\ominus$. Clearly, a $\oplus$ that is both commutative and associative using the algorithm described in this paper would yield the same alignment on all trees for a given leaf-set.
Abbreviations
DO: Direct Optimization; IA: Implied Alignment; MSA: Multiple Sequence Alignment; OLS: Ordinary Least Square; TAP: Tree Alignment Problem

Acknowledgements
We would like to thank Eric Ford, Callan McGill, Katherine St. John, and Erilia Wu for insightful discussions. We would also like to thank the three reviewers for improvements to the manuscript.

Authors' contributions
AW developed the algorithmic improvements, implemented the prototype program, quantified the relationship between similar inputs and improved performance. WW provided empirical data sets, developed and implemented pruning methodologies for the scaling of the data sets, and performed literature review. Both authors read and approved the final manuscript.

Funding
This work was supported by DARPA SIMPLEX ("Integrating Linguistic, Ethnographic, and Genetic Information of Human Populations: Databases and Tools," DARPA-BAA-14-59 SIMPLEX TA-2, 2015-2018) and Robert J. Kleberg Jr. and Helen C. Kleberg foundation grant "Mechanistic Analyses of Pancreatic Cancer Evolution".

Availability of data and materials
The datasets generated and analysed in the study are available in the GitHub.com repository, https://github.com/recursion-ninja/efficient-implied-alignment

Ethics approval and consent to participate
Not Applicable.

Consent for publication
Not applicable.

Competing interests
The authors declare that they have no competing interests.

Received: 19 November 2019 Accepted: 10 June 2020 Published online: 09 July 2020

References
1. Wheeler WC. Implied alignment. Cladistics. 2003a;19:261–8.
2. Varón A, Wheeler WC. The tree-alignment problem. BMC Bioinforma. 2012;13:293.
3. Wheeler WC. Optimization alignment: The end of multiple sequence alignment in phylogenetics? Cladistics. 1996;12:1–9.
4. Wheeler WC, Gladstein DS. MALIGN. Unknown Month 1991. program and documentation available at http://research.amnh.org/scicomp/projects/malign.php. documentation by Daniel Janies and W.C. Wheeler.
5. Gladstein DS, Wheeler WC. POY version 2.0. New York: American Museum of Natural History; 1997. http://research.amnh.org/scicomp/projects/poy.php.
6. Wheeler WC, Aagesen L, Arango CP, Faivovich J, Grant T, D’Haese C, Janies D, Smith WL, Varón A, Giribet G. Dynamic Homology and Systematics: A Unified Approach. New York: American Museum of Natural History; 2006.
7. Wheeler WC, Gladstein DS, De Laet J. POY version 3.0. program and documentation available at http://research.amnh.org/scicomp/projects/poy.php (current version 3.0.11). documentation by D. Janies and W.C. Wheeler. commandline documentation by J. De Laet and W. C. Wheeler. New York: American Museum of Natural History; Unknown Month 1996.
8. Wheeler WC, Lucaioni N, Hong L, Crowley LM, Varón A. POY version 5.0. American Museum of Natural History; 2013. http://research.amnh.org/scicomp/projects/poy.php.
9. Wheeler WC, Lucaioni N, Hong L, Crowley LM, Varón A. POY version 5: Phylogenetic analysis using dynamic homologies under multiple optimality criteria. Cladistics. 2015;31:189–196.
10. Wheeler WC. Homology and the optimization of DNA sequence data. Cladistics. 2001;17:53–511.
11. Whiting AS, Sites JW, Pellegrino KC, Rodrigues MT. Comparing alignment methods for inferring the history of the new world lizard genus Mabuya (Squamata: Scincidae). Mol Phyl Evol. 2006:38:719–30.
12. Wheeler WC. Dynamic homology and the likelihood criterion. Cladistics. 2008;22:157–70.
13. Varón A, Vinh LS, Wheeler WC. POY version 4: Phylogenetic analysis using dynamic homologies. Cladistics. 2010;26; 72–85.
14. Loytynoja A, Goldman N. An algorithm for progressive multiple alignment of sequences with insertions. Proc Nat Acad Sci. 2005;102:10557–62.
15. Paten B, Herrero J, Fitzgerald S, Beal K, Flicek P, Holmes I, Birney E. Genome-wide nucleotide-level mammalian ancestor reconstruction. Genome Res. 2008;18:1829–43.
16. Loytynoja A, Goldman N. webPRANK: a phylogeny-aware multiple sequence aligner with interactive alignment browser. BMC Bioinforma. 2010;11(1):579. https://doi.org/10.1186/1471-2105-11-579.
17. Ford E, Wheeler W. Comparison of heuristic approaches to the general-tree-alignment problem. Cladistics. 2015;32: 452–60. https://doi.org/10.1111/cia.12142.
18. Lehtonen S. Phylogeny estimation and alignment via POY versus Clustal–PAUP: A response to Ogden and Rosenberg (2007). Syst Biol. 2008;57:653–7.
19. Lindgren AR, Daly M. The impact of length-variable data and alignment criterion on the phylogeny of Decapodiformes (Mollusca: Cephalopoda). Cladistics. 2007;23:464–476.
20. Wheeler WC, Giribet G. Phylogenetic hypotheses and the utility of multiple sequence alignment, pp. 95–104. In: Rosenberg MS, editor. Perspectives on Biological Sequence Alignment. Berkeley: University of California Press; 2009.
21. Wheeler WC. Systematics: A course of lectures. Oxford: Wiley-Blackwell; 2012. https://doi.org/10.1111/jzs.12009.
22. Wheeler WC. Search-based character optimization. Cladistics. 2003b;19:346–353.
23. Sankoff DM. Minimal mutation trees of sequences. SIAM J Appl Math. 1975;28:35–42.
24. Jacobson GJ. Succinct Static Data Structures. PhD thesis. USA: Carnegie Mellon University; 1988. AAI8918056.
25. Needleman SB, Wunsch CD. A general method applicable to the search for similarities in the amino acid sequences of two proteins. J Mol Biol. 1970;48:443–53.
26. Sankoff D. The early introduction of dynamic programming into computational biology. Bioinformatics. 2000;16:41–7.
27. Cormen TH, Leiserson CE, Rivest RL, Stein C. Introduction to Algorithms. 2nd edition. Cambridge: The MIT Press; 2001.
28. Ukkonen E. Algorithms for approximate string matching. Inf Control. 1985;64:100–118. International Conference on Foundations of Computation Theory.
29. Hirschberg DS. A linear space algorithm for computing maximal common subsequences. Commun ACM. 1975;18:341–3.
30. Gotoh O. An improved algorithm for matching biological sequences. J Mol Biol. 1982;705–8.
31. Giribet G, Wheeler WC. The position of arthropods in the animal kingdom: Ecdysozoa, islands, trees and the ‘parsimony ratchet’. Mol Phyl Evol. 1999;10:1–5.
32. Giribet G, Wheeler WC. Some unusual small-subunit ribosomal DNA sequences of metazoans. AMNH Novitates. 2001;3337:1–14.

Publisher’s Note
Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.