Algebraic Dynamic Programming on Trees

Sarah J. Berkemer\textsuperscript{a,b,*}, Christian Höner zu Siederdissen\textsuperscript{b}, Peter F. Stadler\textsuperscript{b,c,a,d,e,f,g}

\textsuperscript{a}Max Planck Institute for Mathematics in the Sciences, Inselstraße 22, D-04103 Leipzig, Germany
\textsuperscript{b}Bioinformatics Group, Department of Computer Science, and Interdisciplinary Center for Bioinformatics, University of Leipzig, Härtestraße 16-18, D-04107 Leipzig, Germany
\textsuperscript{c}German Centre for Integrative Biodiversity Research (iDiv) Halle-Jena-Leipzig, Competence Center for Scalable Data Services and Solutions, and Leipzig Research Center for Civilization Diseases, University Leipzig, Germany
\textsuperscript{d}Fraunhofer Institute for Cell Therapy and Immunology, Perlickstrasse 1, D-04103 Leipzig, Germany
\textsuperscript{e}Institute for Theoretical Chemistry, University of Vienna, Währingerstraße 17, A-1090 Wien, Austria.
\textsuperscript{f}Center for RNA in Technology and Health, Univ. Copenhagen, Grønegårdsvej 3, Frederiksberg C, Denmark
\textsuperscript{g}Santa Fe Institute, 1399 Hyde Park Rd., Santa Fe, NM 87501

Abstract

Where string grammars describe how to generate and parse strings, tree grammars describe how to generate and parse trees. We show how to extend generalized algebraic dynamic programming to tree grammars. The resulting dynamic programming algorithms are efficient and provide the complete feature set available to string grammars, including automatic generation of outside parsers and algebra products for efficient backtracking. The complete parsing infrastructure is available as an embedded domain-specific language in Haskell.

In addition to the formal framework, we provide implementations for both, tree alignment and tree editing. Both algorithms are in active use in, among others, the area of bioinformatics, where optimization problems on trees are of considerable practical importance.

This framework and the accompanying algorithms provide a beneficial starting point for developing complex grammars with tree- and forest-based inputs.

Keywords: Dynamic programming; Formal language; Tree editing; Tree alignment

1. Introduction

Dynamic programming (DP) is a general paradigm for solving combinatorial problems with a search space of exponential size in polynomial time (and space). The key

\textsuperscript{*}Corresponding author

Email address: bsarah@bioinf.uni-leipzig.de (Sarah J. Berkemer)
idea is that the optimal solution for the overall problem can be constructed by combining
the optimal solutions to subproblems (known as optimal substructure). During the re-
cursive deconstruction of the original problem instance, the same smaller subproblems
(known as overlapping subproblems) are required repeatedly and thus memoized. DP
is typically applied to optimization and (weighted) enumeration problems. Not all such
problems are amenable to this strategy however.

Bellman’s principle [1] codifies the key necessary condition, the intuition of which
can be expressed for optimization problems as follows: The optimal solution of any
relevant (sub)problem can be obtained as a suitable combination of optimal solutions of
its subproblems.

In Bioinformatics, DP algorithms are a useful tool to solve tasks such as sequence
alignment [2] or secondary structure prediction [3,4]. Here, the description of dynamic
programming algorithms as formal grammars is common practice [5]. Problems whose
natural input structure is a tree typically have no satisfactory solution using an encoding
as strings, because one “dimension” within the input is lost. A simple, yet often encoun-
tered, example is the structure of a text such as a book. Books are divided into chapters,
sections, subsections, and blocks of text, imposing a hierarchical structure in addition to
the linear order of words. Many other formats for data exchange are based on similar
structures, including json and xml. In bioinformatics tree structures encode, among
others, the (secondary) structure of RNA and phylogenetic trees. DP algorithms on trees
have a long history in computational biology. Well-studied problems include the com-
putation of parsimony and likelihood scores for given phylogenies [6], tree alignment
[7,8] and tree editing [9], reconciliation of phylogenetic trees [10], and RNA secondary
structure comparison [11,12]. All of these approaches use trees and/or forests as input
structures, however aims and methods differ. We argue here that a common high-level
framework will substantially simplify the design and development of such algorithms
and thus provide a very useful addition to the toolbox of someone working with tree-like
data. There are implementations for most of the problems mentioned above. However,
they use problem-specific choices for the recursions and memoization tables that seem
to be hard to generalize.

A particularly difficult challenge is the construction of DP algorithms that are not
only complete but also unambiguous. The latter property ensures that each configuration
is considered only once, so that the DP recursion correctly counts and thus can be used to
compute partition functions. A recent example is the approach of [7] to tree alignment.
In general, recognizing ambiguity for context-free grammars is undecidable [13, 14].
There are, however, quite efficient heuristics that can be used successfully to prove that
string grammars describing many problems in computational biology are unambiguous
[15]. The aim of the present contribution is to develop a general framework as a basis
for implementations of DP algorithms on trees and forests.

The most difficult issue in this context is to identify a sufficiently generic repre-
sentation of the search spaces on which the individual application problems are defined.
Following [16], a very natural starting point is a formal grammar $G$ as a rewriting system specifying a set of rules on how to generate strings or read strings of the corresponding language $L(G)$. The benefit of using formal grammars compared to, say, explicit recurrences, is mostly in the high-level description of the solution space defined by the problem at hand. For formal grammars describing dynamic programming problems on strings several approaches have been developed in recent years [16–20]. Formal grammars can describe tree languages just as well as string languages. This opens up the benefits of using formal languages (in abstracting away the minutia of implementation details) for this expanded input type [21, 22].

Our contributions are twofold. First, we provide a theoretical framework for describing formal languages on trees. We give a small set of combinators – functions that formalize the recursive deconstruction of a given tree structure into smaller components – for the design of such formal languages. Second, we show that an implementation in the ADPfusion variant of algebraic dynamic programming (ADP) [23] yields short and elegant implementations for a number of well-known algorithms on tree structures.

With this contribution ADPfusion now encompasses the most widely used types of inputs for dynamic programs, namely (i) strings [17, 18], (ii) sets [24], and (iii) trees (this work).

2. Algebraic Dynamic Programming (ADP)

Algebraic dynamic programming (ADP) [18, 25] is designed around the idea of higher order programming and starts from the realization that a dynamic programming algorithm can be separated into four parts: (i) grammar, (ii) algebra, (iii) signature, and (iv) memoization. These four components are devised and written separately and combined to form the solution. One advantage of ADP is that the decoupling of the individual components makes each individual component much easier to design and implement. Another advantage is that different components can be re-used easily. This is in particular true for the grammar, which defines the search space. Hence, it needs to be specified only once. The grammar only specifies the input alphabet and the production rules on how to split the input data structure in subparts. The algebra specifies the cost function and the selection criterion and thus defines formally what is considered as an “optimal” solution. The scope of the evaluation algebra is quite broad. It can be used e.g. to count and list prioritized (sub-)optimal solutions. The signature is used as a connecting element between grammar and algebra such that they can be re-used and exchanged independently from each other as long as they fit to the signature. In order to recursively use subsolutions to the problem, they are stored using a data structure easy to save and look-up the subsolution to the current instance. This is called memoization. In this section, the four parts of ADP will be explained using string alignment as an example.

Before we delve into dynamic programming on trees and forests, we recapitulate the
notations for the case of dynamic programming on strings. The problem instance is the
evenerable Needleman-Wunsch [26] string alignment algorithm. The algorithm expects
two strings as input. When run, it produces a score based on the number of matches,
mismatches, deletions, and insertions that are needed to rewrite the first into the second
input string.

One normally assigns a positive score to a match, and negative scores to mismatches,
deletions, and insertions. With the separation of concerns provided by algebraic dynamic
programming, the algebra that encodes this scoring becomes almost trivial, while the
logic of the algorithm, or the encoding of the search space of all possible alignments in
the form of a grammar, is the main problem. As such, we mostly concentrate on the
grammars, delegating scoring algebras to a secondary place.

A simple version of the Needleman-Wunsch algorithm [26] is given by:

\[
\begin{align*}
(X_X) & \rightarrow (X_X)^a | (X_X)(a) | (X_X)(a) \mid (S) \\
\text{match} & \quad \text{deletion} \quad \text{insertion} \quad \text{empty}
\end{align*}
\]  

(1)

The grammar has one 2-line symbol on the right-handed side, the non-terminal \((X_X)\). The
notion of two lines of \(X\) is indicative of the algorithm requiring two input sequences. This
notion will be made more formal in the next section. The operations of the grammar can
be explained for each symbol individually, while grouping into columns indicates that
the symbol on each line, and thereby input, operates in lockstep with the other symbols
in the column.

Here, \(X\) is a non-terminal of the grammar, \(a\) a terminal character, \$ the termination
symbol and the “no input read”-symbol −. Non-terminals such as \((X_X)\) are always regarded
as one symbol, and can be replaced by any of the rules on the right-handed side that have
said symbol on the left-handed side. Terminals, say \((a)\), act on individual tapes or inputs,
here not moving on the first input, but reading a character “into” \(a\) on the second input.

The grammar defines 4 different cases: (i) matching the current positions of the input
strings, (ii)-(iii) inserting a gap symbol in one or the other of the input strings or the case
(iv) when both input strings are empty. As the algorithm should return the best possible
alignment of the input strings, each of the production rules is assigned a scoring func-
tion in the scoring algebra. The scoring function will take the current score and return
an updated score including the weight for the current case (match, deletion or insertion).
Fig. 1 shows two different scoring algebras for the grammar of string alignment, with \(A\)
being the input alphabet, and \(S\) the set of scores. Thus, for each grammar, we can choose
a scoring algebra independently. The choice function corresponds to the objective func-
tion of the dynamic programming algorithm. As it is used to rank given (sub)solutions,
it is not part of the grammar but part of the scoring algebra and the signature, as seen in
Fig. 1 and Fig. 2.

The signature in ADP provides a set of function symbols that provide the glue be-
tween the grammar and the algebra. Thus, we define the type signatures for each of the
scoring functions, analogously to type signatures found in functional programming.
count

\[
\begin{align*}
\text{match}(s, (u v)) &= s \\
\text{deletion}(s, (u \ -)) &= s \\
\text{insertion}(s, (- v)) &= s \\
\text{empty}((\$ \ -$)) &= 1 \\
\text{choice}([x_1, \ldots, x_n]) &= x_1 + \ldots + x_n
\end{align*}
\]

with \( u, v \in \mathcal{A} \) and \( s \in \mathbf{S} \)

\[
\begin{align*}
\text{match}(s, (u v)) &= s + w(u, v) \\
\text{deletion}(s, (u \ -)) &= s + w(u, -) \\
\text{insertion}(s, (- v)) &= s + w(-, v) \\
\text{empty}((\$ \ -$)) &= 0 \\
\text{choice}([x_1, \ldots, x_n]) &= h(x_1, \ldots, x_n)
\end{align*}
\]

with \( u, v \in \mathcal{A} \) and \( s \in \mathbf{S} \)

Figure 1: Two different scoring algebras written for the grammar of string alignment. Here, \( s \in R \) is the score, \( \mathcal{A} \) the input alphabet, \(-\) the gap symbol and \$ the empty string. The choice function specifies the objective function applied in each recursion step. The scoring algebra count (left) returns the size of the search space whereas the scoring algebra eval (right) solves the problem of string alignment by applying the weight function and objective function.

Fig. 2 shows the signature for the Needleman-Wunsch algorithm. This signature fits to both scoring algebras and the grammar. Here, \( \mathcal{A} \) is the input alphabet, and \( \mathbf{S} \) the set of scores as above.

\[
\begin{align*}
\text{match} : \mathbf{S} \times (\mathcal{A} \mathcal{A}) &\rightarrow \mathbf{S} \\
\text{deletion} : \mathbf{S} \times (\mathcal{A} \ -) &\rightarrow \mathbf{S} \\
\text{insertion} : \mathbf{S} \times (- \mathcal{A}) &\rightarrow \mathbf{S} \\
\text{empty} : (\mathbf{S} \ -$) &\rightarrow \mathbf{S} \\
\text{choice} : [\mathbf{S}] &\rightarrow \mathbf{S}
\end{align*}
\]

Figure 2: Signature for the problem of string alignment. The signature lists the situations that need to be evaluated differently by specifying the input and output for each situation. Here, \( \mathbf{S} \) is the score, \( \mathcal{A} \) a letter from the alphabet, \(-\) the none symbol and \$ the empty symbol.

The last part needed for ADP is the memoization to store the subsolutions. For the Needleman-Wunsch algorithm we simply use a matrix of size \( O(s \times t) \) where \( s \) and \( t \) are the lengths of the input strings to store the subsolutions.

3. DP Algorithms on Trees and Forests

Algebraic dynamic programming (ADP) \[25\] in its original form was designed for single-sequence problems. Alignment-style problems were solved by concatenating the
two inputs and including a separator.

Originally, ADPfusion [18] was designed along the lines of ADP, but gained provisions to abstract over both, the input type(s), and the index space with the advent of generalized algebraic dynamic programming [17, 24, 27–30] inputs were generalized to strings and sets, as well as multiple tapes (or inputs) of those.

Trees as data structures sit “somewhat between” strings and sets. They have more complex structures, in that each node not only has siblings, but also children compared to lists, which only have siblings for each node in the list or string. Sets on the other hand are typically used on the principle of having edges between all nodes, with each edge having a particular weight. This alone makes trees an interesting structure to design a formal dynamic programming environment for. However, tree and forest structures have, as discussed above, important uses in (for example) bioinformatics and linguistics.

DP algorithms can be applied to just one input structure but also to two or more independent inputs. In the context of formal grammars describing DP algorithms, we denote DP algorithms applied to one input structure as single-tape and two or more inputs as two-tape or multi-tape algorithms.

We now formalize the notion of an input tape on which formal grammars act. For sequences, an input tape is a data type that holds a finite number of ordered elements. The ordering is one-dimensional. It provides the ability to address each element. For all algorithms considered here, the input tape is immutable. Sequences are the focus of earlier work [17, 18].

Input tapes on trees hold a finite number of elements with a parent-child-sibling relationship. Each element has a single parent or is a root element. Each element has between zero and many children. The children are ordered. For each child, its left and right sibling can be defined, unless the child is the left-most or right-most child. Input tapes on forests introduce a total order between the root elements of the trees forming the forest. These structures are discussed in this work. For sets, the tape structure is the power set and the partial order defined by subset inclusion. Set structure have been discussed earlier [24].

A multi-tape grammar, such as the example above, extends this notion to tuples of input tapes. We will formalize the notion of multi-tape inputs later in Defn. [10]. We note in passing that in the case of multiple context-free grammars the one-to-one correspondence between the i’th element of the tuple and the i’th row of each rule does not hold for interleaved symbols. We refer to [30] for the details.

In order to define formal grammars for DP algorithms on trees and forests, we have to define how to split a tree or a forest into its substructures. On strings, the concatenation operator is assumed implicitly to concatenate single symbols of the string. We make the operators explicit as there is more than one operator on trees and forests.

Definition 1 (trees and forests). Let T be a rooted, ordered tree. Then, either T is empty and contains no nodes, or there is a designated node r, the root of the tree. In
addition, for each non-leaf node, there is a given order of its children from left to right. Furthermore, let $F$ be the ordered forest of trees $T_1, \ldots, T_n$, where the order is given by assuming a global root node $r_0$, of which $r_1, \ldots, r_n$ are the direct, ordered children.

We now define the set of operations that will lead to dynamic programming operations on ordered trees and forests. Each operation transforms a tree or forest on the left-handed side of a rule ($\to$) into one or two elements on the right-handed side of a rule. Rules can fail. If a rule fails, it does not produce a result on the right-hand side. In an implementation, right-handed sides typically yield sets or lists of results, where a failure of a rule naturally leads to an empty set or an empty list.

**Definition 2 (empty tree or forest).** If $T$ is empty, then the rule $T \to \epsilon$ yields $\epsilon$. If $F$ is empty, then the rule $F \to \epsilon$ yields $\epsilon$.

**Definition 3 (single root).** If $T$ with root $r$ contains just the root $r$, then $T \to r$ yields $r$.

**Definition 4 (root separation).** Given non-empty $T$, the rule $T \to r \downarrow F$ yields the root $r$ of $T$, as well as the ordered forest $F = T_{r,1} \ldots T_{r,n}$ of trees. Each $T_{r,k}$ is uniquely defined as the $k$’th child of the root $r$ of $T$.

$T \to F \downarrow r$ is isomorphic to this rule.

**Definition 5 (leaf separation).** Given non-empty $T$, the left-most leaf $l = l_{leaff}(T)$ is defined as the leaf the can be reached by following the unique path from the root $r$ of $T$ via each left-most child until a leaf has been reached. The rule $T \to l \uparrow T'$ breaks up $T$ into $l$, and $T' = T - l$.

The rule $T \to T' \downarrow s$ likewise yields the right-most leaf.

**Definition 6 (forest separation).** The rule $F \to T \circ F'$ separates the non-empty forest $F$ into the left-most tree $T$, and the remaining forest $F'$. $T$ is uniquely defined by the ordering of $F$.

The rule $F \to F' \circ T$ likewise yields the right-most tree.

The following lemmas gives that both, tree and forest decomposition are well defined and always lead to a decomposition using at least one of the definitions above.

**Lemma 7 (tree decomposition).** Given any (empty or non-empty) tree $T$, at least one of the rules Defn.(2–5) yields a decomposition of the tree, where either $T$ is decomposed into “atomic” elements ($\epsilon$ or a single node), or into two parts, each with fewer vertices than $T$.

**Proof.** Every tree $T$ is either empty (Defn.2), contains just one node which then is the root (Defn.5), or contains one or more nodes (Defn.3), of which one is separated from the remainder of the structure.
Lemma 8 (forest decomposition). The argument for forest decomposition follows analogously to the one for trees, except that now at least one of Defn. 2 or Defn. 6 yield a decomposition.

Given those tree and forest concatenation operators, we are now able to define formal grammars to describe DP algorithms on trees and forests.

Lemma 9 (Totality). Given an arbitrary tree or forest, at least one of the operations specified in Defn. (2–8) can be applied. Each of these operations reduces the number of nodes associated with each non-terminal symbol or results in a single node or an empty structure. Thus any finite input tree of forest can be decomposed by a finite number of operations.

Proof. Let F be a, potentially empty, forest. For the case of the empty forest, we apply Def. 2. In the case of more than one tree, we apply Def. 6 which yields a single tree and another forest that are both smaller than the original forest. This is also shown in Lemma 8.

In case of Def. 3 the resulting forest structure can be empty such that the original forest only consisted of a single tree. Here, we continue with the possible tree structures.

A tree T can be empty, or consist either of a single node or more than one node. For the empty case, we apply Def. 2. In case of the tree consisting only of a single node, we apply Def. 3. If T consists of more than one node, we either apply Def. 4 or Def. 5. In case of Def. 4 the resulting structure consists of a single node (an atomic structure) and a forest which in total has a smaller number of nodes than the original tree.

In the case of Def. 5, one leaf is eliminated from the tree, thus this yields a tree that just differs from the original structure by one leaf.

For the case of algorithms working on two input structures, it is ensured that at least one of the current structures is decomposed such that the total number of nodes decreased within the next recursion step.

4. Single-tape DP on Trees and Forests

Dynamic programming algorithms on a single input tend to be concerned with the internal structure of the input. Single-tape problems and their associated grammars therefore deal with the structural decomposition of this one input. A well-known example in bioinformatics is the Nussinov algorithm [31] that is applied to one input sequence and returns the optimal secondary structure based on a scoring scheme (often maximal number of paired nucleotides).

4.1. Dynamic programming on single-tape inputs

Analogously to single-tape DP on strings, there exist single-tape DP algorithms on trees and forests. Here, problems include the search for an optimal partitioning of
the tree or detection of paths within the input structure. In contrast, two- or multi-tape DP algorithms are usually used to compare two or more input structure with each other with the aim of finding an optimal way of transforming one into the other or composing them into a consensus structure. We start with two examples of dynamic programming algorithms on single trees. Both solve well-known problems in computational biology but are also easy to state without reference to their usual applications. In the following section we will then address algorithms that take pairs of labeled trees as input.

4.2. The Minimum Evolution Problem

Given a tree $T$ with leaf set $X$, a labelling $\ell : X \rightarrow \mathcal{A}$, and a similarity function $w : \mathcal{A} \times \mathcal{A} \rightarrow \mathbb{R}$, find an extension $\tilde{\ell} : V(T) \rightarrow \mathcal{A}$ such that: (i) $\tilde{\ell}(v) = \ell(v)$ on $X$ and (ii) $w(\ell) := \sum_{(u,v) \in E(T)} w(u, v)$ is maximal. Maximizing similarity amounts to minimizing the number of evolutionary events that occur along the edges of the (phylogenetic) tree, explaining the traditional name of the problem.

A well-known solution to Minimum Evolution Problem problem, known as Sankoff’s Algorithm [32], consists of computing for each complete subtree of $T$, which by construction is rooted in a vertex $v \in V$, and each possible label $a \in \mathcal{A}$ of $v$, the score of the best scoring labeling of that subtree. Let us call this quantity $S_v(a)$. It clearly satisfies the recursion

$$
S_v(a) = \begin{cases} 
0 & \text{if } \ell(v) = a \\
-\infty & \text{otherwise}
\end{cases} 
$$

for all $v \in X$

$$
S_v(a) = \sum_{c \in \text{chd}(v)} w(a, \ell(c)) + S(c) \quad \text{for all } v \in V(T) \setminus X
$$

The grammar to Sankoff’s algorithm as described above is given by

$$
T \rightarrow x \downarrow F \\
F \rightarrow T \circ F \mid S
$$

We remark that the same grammar can also be use to describe Fitch’s small parsimony problem [33] for binary trees, which was later generalized by Hartigan [34] for arbitrary trees. Even though the same grammar is used, the scoring algebra is completely different in the two algorithms, see Appendix 9.2.

4.3. The Phylogenetic Targeting Problem

Here we are again given a tree $T$ with leaf set $X$, this time together with a weight function $w : X \times X \rightarrow \mathbb{R}$. The weight $w$ models the amount of information that can be gained by comparing a pair of two taxa, i.e., a pair of leafs. The task is to find a maximally informative set of mutually phylogenetically independent pairs of taxa.

Two pairs of taxa $u, v$ and $x, y$ are said to be phylogenetically independent if the paths from from $u$ to $v$ and from $x$ to $y$, respectively, have no edge in common [35]. A (partial) solution is thus a tree $T$ endowed with a set of disjoint paths that connect pairs
of leaves. There are two distinct types: in trees of type $U$ all paths between leaves in $U$ are confined to $U$. In the other case a path leaves the subtree tree though its root, which thus must be connected by a path to one of the leaves. We denote such complete subtrees with a leaving path by $W$. Of course the solution of the complete problem is of type $U$. It is not hard convince oneself that the child-trees of a type $U$ tree must be either both of type $U$ or both of type $W$. In the latter case a path runs through the root of $U$ and connects the two type $W$ children. Further, denote by $G$ a forest consisting of two trees of the same type, and let $H$ be a forest comprising two trees of different type. This yields the following grammar:

$$
U \rightarrow r \downarrow G
$$

$$
G \rightarrow U \cdot U \mid W \cdot W \mid $

$$
W \rightarrow r \downarrow H
$$

$$
H \rightarrow U \cdot W \mid W \cdot U \mid $\tag{4}

There are again several possible scoring algebras. As described in [36, 37], to obtain the best possible path system, the objective function maximizes over the best possible path systems of the subtrees given from the current instance of the program and add up the scores.

5. Two-tape and Multi-Tape DP on Trees and Forests

We begin with the formal definition of multi-tape inputs, where the type of input can be generic. However, for the purpose of this work we will deal with multiple trees or forests. Here, we also need to make an important distinction. When we discuss trees and forests, then a forest is composed of many trees, but the forest by itself is a single input or single tape.

Definition 10 (multi-tape inputs). A multi-tape input with $k$ inputs is a $k$-tuple of input tapes. In a multi-tape grammar operating on $k$ inputs, the $i$'th element of the tuple is operated on by symbols in the $i$'th row of each rule.

Now that we have a formal definition of multiple input tapes, we can also formalize the notion of DP algorithms on forests. We first note that any forest can be represented as a tree with the introduction of a root node $r_0$ of which the trees of the forest are the children. As such, a definition on trees is enough to cover forests as well. See [7] for a similar definition.

As examples for DP algorithms on trees and forests, we will use the following sections to describe two important, and often-used DP problems on trees, the tree editing [38, 39] (Sec. 9) and tree alignment [40] (Sec. 6) problems. Tree editing is concerned with finding the optimal edit script that transforms the first input tree into the second
input tree. Tree alignment, on the other hand, gives the optimal alignment of two trees with each other, as will be described below in more detail. These algorithms serve as a tutorial on how grammars on trees are to be formulated. We expand on earlier work by introducing several variants. While some are known from previous work, the terse and high level notation, ability to construct combined grammars, and automatic derivation of the corresponding outside grammar give a unique framework. Here, we also include some notations on multi-tape DP.

6. Tree Alignment

Our grammar for the case of tree alignment is based on the grammars formulated in [41, 42]. We expand those grammars with explicit tree concatenation operators as shown in the previous section. This allows our algorithms to parse both trees and forests as input structures.

Furthermore, we extended the grammars with the automatic derivation of an outside algorithm [24], described in Sec. 8. The combination of the inside and outside grammar allows for easy calculation of match probabilities. Since this is automatic, any user-defined grammar can be extended in this way as well. Though we point out that this still requires careful design of the inside grammar [7].

Representations of alignments (cf. Fig. 3) are typically in the form of a tree that includes are nodes from both trees, where nodes are matched up according to (i) and deleted nodes from both trees are inserted in an partial-order preserving fashion.

**Definition 11 (Tree alignment).** Consider a forest $G$ with vertex labels taken from $(\mathcal{A} \cup \{-\}) \times (\mathcal{A} \cup \{-\})$. Then we obtain restrictions $\pi_1(G)$ and $\pi_2(G)$ by considering only the
first or the second coordinate of the labels, respectively, and by then deleting all nodes that are labeled with the gap character $-$ instead of $\varnothing$, see Fig. 3. $G$ is an alignment of the two forests $F_1$ and $F_2$ if $F_1 = \pi_1(G)$ and $F_2 = \pi_2(G)$.

The cost of the alignment $G$ is the sum of the costs of the label pairs:

$$\gamma(G) = \sum_{(v_1, v_2) \in G} \gamma_{v_1 v_2}$$

(5)

Every alignment defines a unique mapping, but the converse is not true. The minimum cost alignment is in general more costly than the minimum cost edit script.

We will need a bit of notation. Let $F$ be an ordered forest. By $i : F$ we denote the subforest consisting of the first $i$ trees, while $F : j$ denotes the subforest starting with the $j + 1$-th tree. By $F^\downarrow$ we denote the forest consisting of the children-trees of the root $v = r_F$ of the first tree in $F$. $F^{-} = F : 1$ is the forest of the right sibling trees of $F$.

Now consider an alignment $A$ of two forests $F_1$ and $F_2$. Let $a = r_A$ be the root of its first tree. We have either:

1. $a = (v_1, v_2)$. Then $v_1 = r_{F_1}$ and $v_2 = r_{F_2}$; $A^\downarrow$ is an alignment of $F_1^\downarrow$ and $F_2^\downarrow$; $A^{-}$ is an alignment of $F_1^{-}$ and $F_2^{-}$.
2. $a = (v_1, -)$. Then $v_1 = r_{F_1}$; for some $k$, $A^\downarrow$ is an alignment of $F_1^\downarrow$ and $k : F_2$ and $A^{-}$ is an alignment of $F_1^{-}$ with $F_2 : k$.
3. $a = (-, v_2)$. Then $v_2 = r_{F_2}$; for some $k$, $A^\downarrow$ is an alignment of $k : F_1$ and $F_2^\downarrow$ and $A^{-}$ is an alignment of $F_1 : k$ with $F_2^{-}$.

These three cases imply the following dynamic programming recursion:

$$S(F_1, F_2) = \min \begin{cases} S(F_1^\downarrow, F_2^\downarrow) + \gamma_{v_1} \\ \min_k S(F_1^\downarrow, k : F_2) + S(F_1^{-}, F_2 : k) + \gamma_{\varnothing} \\ \min_k S(k : F_1, F_2^\downarrow) + S(F_1 : k, F_2^{-}) + \gamma_{\varnothing} \end{cases}$$

(6)

with initial condition $S(\emptyset, \emptyset) = 0$. The formal grammar underlying this recursion is

$$
(\ell_F) \rightarrow (\ell_F)\cdot(\ell_F) \mid \ell_F \mid (\ell_\varnothing)\cdot(\ell_F) \mid (\ell_\varnothing)\cdot(\ell_F) \mid (\ell_\varnothing)\cdot(\ell_F)
$$

(7)

It is worth noting that single tape projections of the form $T \rightarrow \Downarrow F$ make perfect sense. Since $\Downarrow$ is a parser that always matches and returns an empty string, which in turn is the neutral element of the concatenator $\cdot$, this formal production is equivalent to $T \rightarrow F$, i.e., it produces a forest $F$ that happens to consist just of a single tree $T$.

As depicted in Fig. 4 fixing the leftmost tree $T$ in a forest, there are two directions to traverse the forest: downwards towards the forests of the root’s children $F^\downarrow$ and sideways towards the remaining forest $F^{-}$. Regarding one single forest, the subforests $F^\downarrow$
and $F^\rightarrow$ are two disjoint entities, thus once split, they do not share any nodes in a further step of the decomposition algorithm. The grammar for tree alignment as described above is inefficient, however, because there is no explicit split between $F^\downarrow$ and $F^\rightarrow$ in the first step. The grammar shown in Eq. 8 explicitly splits the two forests in an earlier step to avoid redundancy.

An efficient variant that makes use of a number of facts that turn this problem into the equivalent of a linear grammar on trees has been described [43]. Trees are separated from their forests from left to right, and forests are always right-maximal. Given a local root node for the tree, and a ternary identifier ($\{T,F,E\}$), each forest, tree, and empty forest can be uniquely identified. Trees by the local root, forests by the local root of their left-most tree, and empty forests by the leaves “above” them. The asymptotic running time and space complexity for trees with $m$ and $n$ nodes respectively is then $O(mn)$.

If the nodes of the tree are labelled in pre-order fashion several operations on the forest can be done more efficiently. By splitting a forest into a tree and the remaining forest, we need to store the tree’s indices to know where it is located in the forest. In case of a pre-order indexing we can take the leftmost tree without storing additional indices as the left border of the tree is the smallest index (which is its root) of the original forest and the right border of the tree is just the predecessor of the leftmost root of the remaining forest. Thus, storing the roots’ indices in a forest will directly give us the rightmost leaves of the corresponding trees.

We finally, consider a variant of equ. (7) that distinguishes the match rule $(T \rightarrow (n)_F)$ with a unique non-terminal $(T)$ on the left-hand side. This rule, which corresponds to the matching of the roots of two subtrees, is critical for the calculation of match probabilities and will play a major role in Sec. 8. The non-terminals $(T)$ and $(F)$ designate insertion and deletion states, respectively. Thus, we can formulate the formal grammar...
for tree alignment as follows:

\[\begin{align*}
\text{iter} & \quad (\ell') \rightarrow (\ell)^{\circ} (\ell') \quad | \quad (\ell)^{\circ} (\ell') \quad | \quad (\ell)^{\circ} (\ell') \quad | \quad (\ell') \\
\text{align} & \quad (\ell) \rightarrow (\omega) : (\ell') \\
\text{deletion} & \quad (\ell) \rightarrow (\omega) : (\ell') \\
\text{insertion} & \quad (\ell) \rightarrow (\omega) : (\ell')
\end{align*}\]

This grammar explicitly splits \(F^i\) and \(F^r\) by applying \text{iter}. Hence, these parts recurse independently from each other. As shown in Fig. 4, aligning two trees can lead to aligning the downwards forest of one tree to the forest to the right of the other tree.

Given Eq. 6, the first case corresponds to our first rule in \text{iter} and the \text{align} rule. Thus replacing \((\ell')\) in \text{iter} we obtain \((\ell') \rightarrow ((\omega) : (\ell'))^{\circ} (\ell')\). Here, \((\omega)\) will give the score for aligning to nonterminals thus \(\gamma_{u,v}\). The tree concatenation operators \((\omega)\) and \((\ell')\) correspond to \(F^1\), \(F^1\), and \(F^r\), \(F^r\), respectively. Replacing \((\ell)\) and \((\ell')\) in the deletion and insertion rules, we get \((\ell') \rightarrow ((\omega) : (\ell'))^{\circ} (\ell')\) and \((\ell') \rightarrow ((\omega) : (\ell'))^{\circ} (\ell')\). This corresponds to case 2 and 3 in Eq. 6. As depicted in Fig. 4, given a deleted root in one tree, we recurse by aligning both subforests. In case one of the subforests is empty, it will be replaced by the first tree (the first \(k\) trees, respectively) of the remaining forest.

The following shows an example scoring algebra for the case of tree alignment with linear gap costs. No costs are added in the \text{iter} case but scores obtained from subsolutions are added, given by \(s_U\) and \(s_V\). Here, \(U\) and \(V\) stand for the nonterminal characters whereas \(u, v\) are terminals.

\[\begin{align*}
\text{iter}(s_U, s_V) & = s_U + s_V \\
\text{match}(s, (\omega)) & = s + \omega(u, v) \\
\text{deletion}(s, (\omega)) & = s + \omega(u, -) \\
\text{insertion}(s, (\omega)) & = s + \omega(-, v) \\
\text{empty}(s) & = 0 \\
\text{choice}(s_1, s_2, \ldots) & = \min(s_1, s_2, \ldots)
\end{align*}\]

with \(u, v \in \mathcal{A}\) and \(s, s_U, s_V \in S\)

6.1. Two and multi-tape Tree Alignment

**Definition 12 (multiple tree alignment).** Let \(T_1, T_2, \ldots, T_k\) be ordered trees following Defn. 7. As such in the partial order of nodes of each tree, parent nodes are ordered before their children and children are ordered from left to right. An alignment between trees is a triple of functions:

- (i) The set of matched nodes form a partial order-preserving bijection. That is, if \(a = f^i(a), b = f^j(b) \in T_i\) are matched with \(f^i(a), f^j(b) \in T_j\) the partial orders \(a < b\) and...
(i) The function \( f^j(a) < f^j(b) \) hold for all nodes \( a, b \) and pairs of trees \( T_i, T_j \). (ii) The set of deleted nodes form a simple surjection from a node \( a \in T_i \) onto a symbol (typically ‘\( - \)’) indicating no partner. (iii) For trees \( T_j \), (ii) holds analoguously.

The problem descriptions in Def. [11] and Def. [12] are equivalent. Every forest can be defined as a tree with the roots of the forest as children of a trees’ root. Given Def. [11], the mappings \( \pi_1 \) and \( \pi_2 \) can therefore be applied to a tree as well.

Given Def. [12] (i), the partial order is also preserved in case of forests, as the roots of the forests are partially ordered, too. Def. [12] (ii) and (iii) define the order of the deleted nodes that is preserved within the tree. This is also true for forests, and obtained by the mappings \( \pi_1 \) and \( \pi_2 \) defined in Def. [11].

A multiple alignment of forest structures can be constructed given the Definitions 2–8. We now show that, given these definitions, the alignment is well-defined and can always be constructed independent of the input structures. We do not show here that the resulting alignment is optimal in some sense, as this depends on the optimization function (or algebra) used.

Let \((F_1, \ldots, F_k)\) be an input of \( k \) forests. Successive, independent application of the operations specified in Definitions 2–8 to individual input trees yields the multi-tape version of the empty string \((\epsilon_1, \ldots, \epsilon_k)\), i.e., the input is reduced to empty structures. Not every order of application necessarily yields an alignment.

Just as in the case of sequence alignments [17] the allowed set of operations needs to be defined for multi-tape input. This amounts in particular to specify which operations act concurrently on multiple tapes. The above-given definitions can be subdivided into two sets. One set of operations takes a forest \( F \) and splits off the left- or rightmost tree \((F \rightarrow F' T \text{ or } F \rightarrow T F')\), where both \( F' \) and \( T \) are allowed to be empty. In case of \( T \) being empty then it holds that \( F' = F \). If \( F' \) is empty, then this yields a large decomposition of \( F \) only if \( T \equiv F \) (abusing notation somewhat). The other set of operations removes atomic elements from a tree and yields a tree of forest, depending on the rule used.

As in [17], all operations have to operate in “type lockstep” on all inputs simultaneously. That is, either all inputs separate off the left- or rightmost tree and yields the trees \((T_1, \ldots, T_k)\) and remaining forests \((F_1, \ldots, F_k)\), or all operations remove a terminal or atomic element, yielding the terminal elements \((n_1, \ldots, n_k)\) and remaining structure \((S_1, \ldots, S_k)\).

Furthermore, an operation \( T \rightarrow T \sim T \) is introduced. This operation does not further decompose \( T \) but provides a terminal symbol ‘\( - \)’ as element \( n_i \). This is analogous to the in/del case in string alignments.

**Theorem 13 (Alignment).** Every input \((F_1, \ldots, F_k)\) of forests can be deconstructed into \((\epsilon_1, \ldots, \epsilon_k)\) by successive application of the steps outlined above.

**Proof of Theorem.** At least one of the decompositions defined above for forests [2, 5] or trees [2, 5] can always be applied for each input. Due to lockstep handling, all inputs
simultaneously separate into a tree and remaining forest (yielding two structures), or apply a rule yielding a terminal element. As such, either all structures are already empty ($\epsilon$) or at least one structure can be further decomposed yielding a smaller structure.

7. The Affine Gap Cost Model for Alignments

The simple linear scoring of gaps in alignments as in the original formulation by [26] is often a poor model in computational biology. Instead, one typically uses affine gap cost with a large contribution for opening a gap and small contributions for extending the gaps. The sequence alignment problem with affine gap costs was solved by [44]. The corresponding formal grammar, in the version used by [17] reads

$$
M \rightarrow M'(u) \mid D'(u) \mid I'(u) \mid (\emptyset) \\
D \rightarrow M'(u) \mid D'(u) \mid I'(u) \\
I \rightarrow M'(v) \mid D'(v) \mid I'(v)
$$

(9)

where $u$ and $v$ are terminal symbols, ‘−’ denotes the opening of a gap, and ‘.’ denotes the extension of gap, typically scored differently. Considering only one tape or input dimension, a deletion is denoted by a leading ‘−’ followed by a number of ‘.’ characters, e.g. a sequence ‘−....’.

For trees, the situation is more complicated. A node in a tree may have siblings as well as children. Once a node has been aligned to an initial gap symbol (‘−’) both its siblings and its children are extending the initial gap. Compared to the three rules for matching, deletion and insertion, we now have to deal with seven different cases. In addition, we explicitly write each non-terminal in such a way as to show the state of each tape. In [41, 42] seven rules for affine gap costs in forests are formulated based on different modes of scoring: no-gap mode, parent-gap mode and sibling-gap mode. Parent and sibling mode indicate that the preceding node (either parent or sibling node) was considered a deletion. The formal grammar expressing the seven rules for tree alignment with affine costs is shown in the appendix 9.2.

In most applications, however, there is little reason to distinguish the parent and sibling mode gaps in the scoring function. Omitting also the explicit rules for gap extension, the grammar can be simplified considerably, see also [41, 42]. As the grammar is also just an in-between step to the final grammar for tree alignment with affine gap costs, we therefore refer to the appendix 9.2 as well. After removing the explicit rules for gap extensions, the grammar now includes rules for parent gap modes that call themselves recursively.

Following these observations, the grammar can be further simplified by summarizing several rules and making gap opening and gap extension costs implicit. To this end we write $F$ for the non-gap mode, $R$ for the parent-gap mode and $Q$ for sibling-gap mode. Instead of adding the gap costs in the explicit cases for gaps, e.g. $(f)$ and $(j)$, affine
gap costs are now added in the rules \((R \rightarrow T \circ R \mid Z \circ R)\) and \((Q \rightarrow T \circ Q \mid Z \circ Q)\) whereas gap opening costs are applied for \((F \rightarrow Z \circ F)\). No costs are added for the rule \((Q \rightarrow T \circ F)\). As the algorithm applies the scoring for each rule on both tapes at the same time, distinguishing between different tapes is not needed anymore as soon as the cases appear on both tapes symmetrically. Thus, the grammar can be simplified and scoring is applied implicitly such that we only distinguish between no-gap mode, gap-opening mode and gap extension:

\[
\begin{align*}
(F \rightarrow (T \circ (F)) \mid (T \circ (Q)) \mid (Z \circ (Q)) \mid (\emptyset)) \\
(Q \rightarrow (T \circ (F)) \mid (T \circ (Q)) \mid (Z \circ (Q)) \mid (\emptyset)) \\
(R \rightarrow (T \circ (R)) \mid (T \circ (Q)) \mid (Z \circ (Q)) \mid (\emptyset)) \\
(F \rightarrow (n) \downarrow (F)) \\
(Q \rightarrow (n) \downarrow (Q)) \\
(R \rightarrow (n) \downarrow (R))
\end{align*}
\]

8. Inside and Outside Grammars

An inside parser will readily produce two kinds of results. Of course a globally optimal solution, say the alignment distance between two trees, can be obtained. Alternatively, the partition function \(Z = \sum_{\omega} e^{s(\omega) / T}\) can be computed. Here, the sum runs over all configurations \(\omega\), \(s(\omega)\) is the score of \(\omega\) and \(T\) is a scaling temperature. For \(T \rightarrow 0\), \(Z\) just counts the number of optimal solutions, for \(T \rightarrow \infty\), all conformations are treated equally. The partition function \(Z\) thus provides access to a probabilistic model. This view plays a key role in practical applications.

While we typically cannot enumerate all possible states because of the exponentially large size of the search space, it is often possible to describe a polynomial number of subproblems that provide salient information about the solutions. For instance, consider the alignment of the two trees in Fig. 5. Here we can ask for the probability of two nodes being matched with each other over all possible alignments. To do this, we would need to know the partition function \(Z'\) of the alignment problem restricted to a single prescribed match; the desired probability is then simply \(Z' / Z\). The tool by which we can make such statements in a principled manner is the combination of the Inside grammar together with an Outside grammar. Conceptually Outside grammars describe decompositions, i.e., parses of the complements of Inside objects. Restricted problems thus are specified by fixed subdivision of the input into a part that is treated by the Inside and the complementary part that is treated by the Outside part.

Outside grammars are typically more complex, having both more rules and a more complex index space, which makes them a bit of a challenge to construct by hand. Because they operate on suitably defined complements of the inside objects, however, there is a completely generic construction of outside grammars [24]. It suffices to write down
Figure 5: Match probabilities for alignments of the trees $T_1$ and $T_2$. Results are compared for linear (l.h.s. column) and affine gaps costs (r.h.s. column), and for two different choices of the scaling temperature $T$. The size of the squares scales logarithmically with the probability as $1/(1 - \log p)$. The score were set to -2 for indels and mismatch, 2 for matches and -1 for the affine gap costs in the affine case. The trees refer to syntax trees for a sentence in german ($T_1$) and in english ($T_2$). For further details, see appendix.
the rules for index space transformation once for each symbol. The machinery then constructs the correct algorithm. The use of automatic construction has the added benefit that the evaluation algebra used for the inside case can be re-used for the outside case, as both grammars share the same signature. Hence they have not only completely isomorphic types, but also the same semantics of the functions is used for evaluation of each parse.

Below, we give the outside grammar (with start symbol \((F^* F^*)\) and outside “epsilon” symbols \(\sigma\) – given that \(\sigma\) in an outside grammar terminates with full input, not the empty input) for the simple linear-cost tree alignment problem (Eq. [11]) and combine inside and outside grammar to yield match probabilities.

\[
\begin{align*}
(F^*) & \rightarrow (T^*)^i (T^*)^j \mid (T^*)^i (F^*)^j \mid (T^*)^i \sigma (F^*)^j \mid (T^*)^i \sigma (F^*)^j \\
(T^*) & \rightarrow (F^*)^\sigma (F^* ) \\
(T^*) & \rightarrow (F^*)^\sigma (F^* ) \\
(F^*) & \rightarrow (F^*)^\sigma (F^* )
\end{align*}
\]  

(11)

The probability that nodes \(i\) in \(T_1\) and \(j\) in \(T_2\) are matched in any given alignment is now simply

\[
\frac{(T^i_j (T^*)^i (F^*)^j \sigma_{0,0})}{z_{0,0}}
\]

which again can be expressed as a grammar: \(P \rightarrow_p (T^i_j \sigma_{0,0})\) with evaluation function \(p = \lambda i.\lambda o \rightarrow io/z\), with \(z_{0,0} = \sigma_{0,0}\). This is why we separated out \((T^i_j)\) in the construction of the alignment grammar, as \((T^i_j)\) holds the total accumulated inside weight with a match \((i, j)\), while \((T^*)\) holds the total accumulated outside weight where the next match will be at \((i, j)\).

**Inside-Outside for Affine Gap Costs.** The combined Inside-Outside algorithm with an affine gap cost model can be implemented in complete analogy to the linear model. One designs the inside grammar and the outside grammar is constructed automatically. This yields an algorithm that computes the match probabilities using the affine gap cost model.

The example in Fig. 5 shows the effect of the scaling temperature \(T\).

Due to the small size of the inputs, alignment with linear and affine gap costs produces similar results. Depending on the temperature, the alignment with the single highest probability mass will dominate \((T \text{ small})\) or many sub-optimal solutions will show up with significant probability \((T \text{ large})\). For higher temperatures, the cost of opening a gap becomes less pronounced yielding a much less constrained probability space than for low temperatures – or the linear model at high temperatures.

**9. Tree Editing**

The string-to-string correction problem can be generalized to forests. To this end, edit operations need to be explained for trees. Again, we consider substitution, insertion,
and deletion:

\[
\begin{array}{cccc}
\text{relabel } y x & \text{relabel } x y & \text{insert } z & \text{delete } z
\end{array}
\]

and associate them with costs \( \gamma^r_{xy} , \gamma^s_{x} , \text{ or } \gamma^\emptyset_{x} \). As it turns out, the generalization of string-to-string correction to trees is not unique.

In this section, we explain our formal grammar for the string editing algorithm on trees instead of strings. Comparing and editing strings is a well-known algorithm in computer science applications, and hence, we expanded this application to trees as input described by a formal grammar.

**Definition 14 (Tree editing).** A mapping \( M \in V(F_1) \times V(F_2) \) between the vertex sets of the two forests such that for pairs \((x,y),(x',y')\) \( \in M \) holds

1. \( x = x' \) if and only if \( y = y' \). (one-to-one condition)
2. \( x \) is an ancestor of \( x' \) if and only if \( y \) is an ancestor of \( y' \). (ancestor condition)
3. \( x \) is to the left of \( x' \) if and only if \( y \) is to the left of \( y' \). (sibling condition)

The one-to-one condition implies that for each \( x \in F_1 \) there is a unique “partner” \( y \in F_2 \), i.e., \((x,y) \in M\), or \( x \) has no matching partner at all. With each mapping we can associate the cost

\[
\gamma(M) = \sum_{(x,y) \in M} \gamma^r_{xy} + \sum_{y,(x,y) \not\in M} \gamma^s_{y} + \sum_{x,(x,y) \not\in M} \gamma^\emptyset_{x}
\]

Individual edit operations correspond to “elementary maps”. Maps can be composed in a natural manner. Thus every edit script corresponds to a map. Conversely every map can be composed of elementary maps, and thus corresponds to an edit script. Furthermore, the cost of maps is subadditive under composition. As a consequence, minimum cost mappings are equivalent to the minimum cost edit scripts [19].

The problem of minimizing \( \gamma(M) \) has a rather obvious dynamic programming solution. For a given forest \( F \) we note by \( F - x \) the forest obtained by deleting \( x \) and \( F \setminus T(x) \) is forest obtained from \( F \) by deleting with \( x \) all descendants of \( x \). Note that \( T(x) - x \) is the forest consisting of all trees whose roots are the children of \( x \).

\[
D(F_1,F_2) = \min \begin{cases}
D(F_1 - v_1, F_2) + \gamma^r_{v_1} \\
D(F_1,F_2 - v_2) + \gamma^s_{v_2} \\
D(T(v_1) - v_1,T(v_2) - v_2) + \gamma^r_{v_1} \\
+D(F_1 - T(v_1), F_2 - T(v_2))
\end{cases}
\]
with $D(\emptyset, \emptyset) = 0$ for two empty forests. A key issue is to implement this algorithm in such a way that only certain classes of subforests need to be evaluated. The corresponding tree editing grammar $\mathcal{E}$ reads

$$
\begin{align*}
\ell F & \rightarrow (n) \cdot (\ell F) \\
(\ell F) & \rightarrow (\ell F) \cdot (\ell F) \\
(\ell F) & \rightarrow (\ell F) \cdot (\ell F) \\
\ell F & \rightarrow (n) \cdot (\ell F)
\end{align*}
$$

(15)

Note that the empty symbol “$-$” acts as neutral element for the concatenation operators, which we take to act component-wise. The grammar is based on the tree editing algorithm of [9], for which several more efficient implementations exist, see [40] for a detailed analysis of the Zhang-Shasha algorithm.

Given the formal grammar to the tree editing problem [15] we can model the individual cases of the recursion equation [14]. Replacing $(\ell F) \rightarrow (\ell F) \cdot (\ell F)$ in the iteration rule we get $(\ell F) \rightarrow (\ell F) \cdot (\ell F)$. This matches the third case of the recursion equation, as we add the score for a match and recurse on the forests of the children and the remaining forests. The other two cases delete a node from one of the current trees, add the score for a deletion or insertion and recurse on the remaining tree. However, the recursion equations describe are more general case by assuming to split a node from a forest. Our grammar first decomposes the forest into a tree and the remaining forest and deletes the node from the tree. As this is in conformation with the direction of traversing the tree based on its index structure, we obtain the same solution as following the recursion equation using the same index structure.

9.1. Outside Grammar

Analogously to the outside grammar for tree alignment (Eq. [11]), we give the outside grammar for tree editing with start symbol $(\ell F)$, and outside “epsilon” symbols $\sigma$ – given that $\sigma$ in an outside grammar terminates with full input, not the empty input.

$$
\begin{align*}
(\ell F) & \rightarrow (\ell F) \cdot (\ell F) \\
(\ell F) & \rightarrow (\ell F) \cdot (\ell F) \\
(\ell F) & \rightarrow (\ell F) \cdot (\ell F) \\
(\ell F) & \rightarrow (\ell F) \cdot (\ell F)
\end{align*}
$$

(16)

9.2. Inside-Outside and Affine Gap Costs

The following grammar describes tree editing with affine gap costs. Here, non-terminals $R$ and $Q$ describe the gap modes.

$$
\begin{align*}
(\ell F) & \rightarrow (\ell F) \cdot (\ell F) \\
(\ell F) & \rightarrow (\ell F) \cdot (\ell F) \\
(\ell F) & \rightarrow (\ell F) \cdot (\ell F) \\
(\ell F) & \rightarrow (\ell F) \cdot (\ell F)
\end{align*}
$$

(17)
Figure 6: Match probabilities for edits of the trees $T_1$ and $T_2$. Results are compared for linear (l.h.s. column) and affine gaps costs (r.h.s. column), and for two different choices of the scaling temperature $T$. The size of the squares scales logarithmically with the probability as $1/(1 - \log p)$. The score were set to -2 for indels and mismatch, 2 for matches and -1 for the affine gap costs in the affine case. The trees refer to syntax trees for a sentence in german ($T_1$) and in english ($T_2$). For further details, see appendix. In comparison to the tree alignment case, the labels of the trees’ nodes are in a slightly different order, as tree editing uses postorder while tree alignment is based on trees in preorder.
Analogous to Fig. 5 comparing alignment probabilities for linear and affine gap costs in a strict and a loose case, Fig. 6 shows the comparison in the case of tree editing. It can be seen, that tree editing generally returns higher probabilities for most of the cases, as tree editing is less strict than tree alignment. This is due to the fact, that deletion and insertion cases for tree editing can freely delete the rightmost leaf from the tree while tree alignment will only align root nodes to other root nodes or gap symbols.

**Conclusion**

Tree comparison has many applications and there exist several approaches to simplify and optimize tree editing [9] and tree alignment [41,42]. Based on existing optimizations, we formalized both algorithms such that they can be written as formal grammars as it has been done for string comparison [17]. Compared to strings, trees can be traversed in two directions, thus our data structure is 2-dimensional. Each grammar consists of terminals and nonterminals, whereas the terminals are single nodes and the nonterminals specify a tree and a forest.

In addition to the linear gap cost version of the algorithms, we developed grammars that provide variants with affine gap costs. Compared to simple, linear cost functions, additional non-terminals are required to distinguish between the initialization of a gap, or gap opening, and gap extension.

For each (inside) grammar, the corresponding outside grammar of the original algorithm [24] can be automatically calculated. Using inside and outside versions, we can specify match probabilities for each pair of subtrees. The inside and outside grammar together thus allow the calculation of ensemble properties such as the probability of each pair of local tree roots to be paired with each other.

For a concrete example from the area of linguistics, we give programs that calculate the pair and editing probabilities for sentences in german and english. While these programs are prototypical and not intended for serious analysis of sentence similarities between languages, they already provide information on conserved sentence structure. In addition they show how the inclusion of affine costs modifies the observed behaviour. This points to one of the major points of our framework: trying a variant, such as affine costs for tree alignment including Inside-Outside calculations can now be done with sufficient ease to allow for exploration of the “space of grammars” that describe a problem instance well.

Combining ADP with tree and forest structures as input, we are now able to apply single-tape and multi-tape DP algorithms on tree-like data. Together with the inclusion of Inside-Outside algorithms, the search space of DP algorithms on trees and forests can be explored broadly. As grammar and algebra can be changed easily, it is possible to compare results based on distinct grammars or cost functions.
One area we have touched only in passing are extensions of both, tree alignment and tree editing beyond two inputs. The basic definition [12] for tree alignments gives the minimum required structure of an alignment, both alignments and edit scripts for trees deserve further treatment outside of the scope of this work. For sequence-based algorithms, we have a formal framework [17, 27] that extends any desired number of sequences and an extension to tree-based algorithms should be investigated.

Further complications arise whenever non-ambiguity is desired. Here, we were mostly concerned with the theoretical basis for a formal language on tree inputs, but as shown by, e.g. [7], the design of a non-ambiguous algorithm on tree inputs is non-trivial. This topic deserves further treatment since proof of non-ambiguity would allow users of our framework to calculate ensemble properties (almost for free thanks to [24, 29]) with the knowledge that their underlying algorithm does not overcount certain structures.

References


[30] M. Riechert, C. Höner zu Siederdissen, P. F. Stadler, Algebraic dynamic programming for multiple context-free languages, TODO this one is accepted and published.


Appendix

Original grammar for affine gap costs

The following grammar expresses the seven rules for affine gap costs in forests [41, 42]. Here, different modes of scoring are applied: no-gap mode, parent-gap mode and sibling-gap mode. Parent and sibling mode indicate that the preceding node (either parent or sibling node) was considered a deletion. Correspondingly, the non-terminal symbol $F$ denotes a no-gap state, $P$ denotes a parent gap, and $G$ denotes a sibling gap. This means that in $P$ mode a gap was introduced in a node further toward the root, while in $G$ mode a gap was introduced in a sibling. In both modes, an unbroken chain of deletions then follows on that tape.

This grammar supports different scoring functions for parent and sibling gaps. Gap opening and gap extension can be distinguished explicitly by including the two additional rules given in equ. (19). They are useful in particular to produce a more expressive output in the backtracing step.

\[
\begin{align*}
(F_F) & \to (T_T)^O(F_F) \mid (T_Z)^O(F_F) \mid (Z_T)^O(F_F) \mid (S) \\
(P_F) & \to (T_T)^O(F_F) \mid (T_Z)^O(F_F) \mid (Z_T)^O(F_F) \mid (S) \\
(G_F) & \to (T_T)^O(F_F) \mid (T_Z)^O(F_F) \mid (Z_T)^O(F_F) \mid (S) \\
(F_P) & \to (T_T)^O(F_P) \mid (T_Z)^O(F_P) \mid (Z_T)^O(F_P) \\
(P_P) & \to (T_T)^O(F_P) \mid (T_Z)^O(F_P) \mid (Z_T)^O(F_P) \\
(G_P) & \to (T_T)^O(F_P) \mid (T_Z)^O(F_P) \mid (Z_T)^O(F_P) \\
(T_T) & \to (\hat{n})^I(F_F) \\
(T_Z) & \to (\hat{n})^I(F_P) \\
(Z_T) & \to (\hat{n})^I(F_P)
\end{align*}
\]
Improved version of grammar for affine gap costs

As, in most applications, there is little reason to distinguish the parent and sibling mode gaps in the scoring function. Omitting also the explicit rules for gap extension, the grammar can be simplified considerably, see also [41, 42]. Here, \((F_F)\) denotes the non-gap mode, whereas the gap-mode is represented by mixed terms. In particular, \((F_F)\) and \((T_Z)\) open gaps, while the remaining mixed terms refer to gap extensions. The rules for \((F_F)\), \((G_F)\), and \((F_G)\) produce the same cases on their right hand sides. The difference are the l.h.s. cases, which distinguish between no-gap mode and gap mode, thus between affine extension cost and gap opening cost. Additionally, the rules expressing parent gap modes \((P_F)\) and \((F_P)\) are recursively calling themselves.

\[
\begin{align*}
(F_F) & \rightarrow (F_F) \circ (F_F) \mid (F_F) \circ (T_Z) \mid (F_F) \circ (Z_T) \mid (F_F) \circ (G_F) \mid \$ \\
(G_F) & \rightarrow (T_T) \circ (F_F) \mid (T_T) \circ (F_G) \mid (T_T) \circ (T_Z) \\
(F_G) & \rightarrow (T_T) \circ (F_F) \mid (T_T) \circ (F_G) \mid (T_T) \circ (T_Z) \\
(P_F) & \rightarrow (n) \cdot (F_F) \\
(T_T) & \rightarrow (n) \cdot (F_F) \\
(Z_T) & \rightarrow (n) \cdot (F_F) \\
(G_F) & \rightarrow (n) \cdot (F_F)
\end{align*}
\]

Comparison of Syntax Trees for different languages

One application for tree comparisons is the comparison of syntax trees. Those syntax trees can be based on syntax for any language, programming language or formula. In this case, we focus on syntax trees for the same sentence in different languages. Fig. 7 shows the syntax trees whereas \(T_1'\) and \(T_2'\) show the real sentences whereas \(T_1\) and \(T_2\) were changed such that labels at the nodes are easily comparable. The inner nodes are labeled with \(S\) at the root for ‘sentence’, \(NP\) for noun production, \(PP\) for predicate production and \(VP\) for verb production.

As a further step in the development of tree comparison algorithms, the user will be able to specify the algorithm for comparing node labels such that the trees’ labels do not have to be rewritten and different comparison possibilities can be applied.

Sankoff’s and Fitch’s algorithms

Sankoff’s algorithm, and many others often used in particular in computational biology, proceeds using two decomposition steps, namely, the separation of the root from the forest of its child trees, and the stepwise decomposition of a forest into its component tree. The grammar underlying equ. (2) thus is simply

\[
T \rightarrow x : F \quad F \rightarrow T \circ F \mid \$
\]
Figure 7: Syntax trees for the same sentence in German (1) and English (2). $T'_1$ and $T'_2$ contain the original labels which were translated to letters to simplify the comparison of node labels ($T_1$ and $T_2$).
The same grammar can be used to describe Fitch’s small parsimony problem for binary trees, which was later generalized by Hartigan for arbitrary trees. Even though, both problems, the Sankoff as well as the Fitch/Hartigan algorithm use the same grammar, the evaluation algebras differ significantly. Each scoring algebra consists of functions corresponding to the rules in the grammar, thus in this case, a tree function, a forest function, a function for the empty case together with the choice function form the algebra. Each (sub)solution consists of a list of pairs \((s_T, V_T)\) consisting of the score \(s_T\) for the input tree \(T\) and a set \(V_T\), containing the nodes included in the solution. Thus, each instance of the problem will also receive the list of pairs \([(s_T, V_T)]\) as an input.

Let \(A\) be the set of possible labels in the tree and \(l\) the length of the input list. Then Fitch/Hartigan uses the scoring algebra depicted in Fig. [9] For the Sankoff version, only a label-dependent score \(w\) for the trees is given as input, with \(L_T\) as the set of labels in the current subtree \(T\), as it can be seen in Fig. [8] Thus, the second algebra considers the Sankoff version of the small parsimony problem.

\[
\begin{align*}
g_{\text{forest}}(s_i, F) & = (s_i + s_F, F') \\
g_{\text{tree}}(s_i, T) & = w((s_i, L_T)) \\
g_{\text{empty}}(s_i, \$) & = 0 \\
g_{\text{choice}}([s_1, \ldots, s_k]) & = s_m \text{ with } s_m = \max_i(s_i)
\end{align*}
\]

Figure 8: Scoring algebra for the Sankoff version of the small parsimony problem.

\[
\begin{align*}
g_{\text{forest}}([s_T, V_T], F) & = ([s_T, V_T]) + (s_T', V_T'), F') \\
g_{\text{tree}}([s_T, V_T], T) & = (s_T', V_T') \text{ whereas} \\
    k(a) & = |\{T' \mid a \in V_T'\} \forall a \in A, \\
    s & = \sum_{T'} s_T', \\
    k & = \max_a(k(a)), \\
    V_T' & = V_{s} \cup \{a \mid k(a) = k\}, \\
    s_T' & = s + l - k, \\
g_{\text{empty}}([s_T, V_T], \$) & = (0, 0) \\
g_{\text{choice}}([s_T, V_T], V_{s_T'}) & = (s_T', V_{s_T'}), \text{ with } s_T' = \max_{s_T}(s_T, V_{s_T}), V_{s_T'} = V_{s} \cup \{a \mid k(a) = k\}
\end{align*}
\]

Figure 9: Scoring algebra for the Fitch/Hartigan version of the small parsimony problem.