Algebraic Dynamic Programming

Robert Giegerich, Carsten Meyer

Faculty of Technology, Bielefeld University
33501 Bielefeld, Germany
{robert,cmeyer}@techfak.uni-bielefeld.de

Abstract. Dynamic programming is a classic programming technique, applicable in a wide variety of domains, like stochastic systems analysis, operations research, combinatorics of discrete structures, flow problems, parsing with ambiguous grammars, or biosequence analysis. Yet, no methodology is available for designing such algorithms. The matrix recurrences that typically describe a dynamic programming algorithm are difficult to construct, error-prone to implement, and almost impossible to debug.
This article introduces an algebraic style of dynamic programming over sequence data. We define the formal framework including a formalization of Bellman's principle, specify an executable specification language, and show how algorithm design decisions and tuning for efficiency can be described on a convenient level of abstraction.

AMAST Categories: programming methodology, specification languages, program transformation, functional programming

1 Introduction

The development of successful dynamic programming
recurrences is a matter of experience, talent, and luck.
An anonymous referee

1.1 Motivation

Dynamic programming (DP) is one of the classic programming paradigms, introduced even before the term Computer Science was firmly established. Bellman's "Principle of Optimality" [2] belongs to the core knowledge we expect from every computer science graduate. Significant work has gone into formally characterizing this principle [20, 22], formulating DP in different programming paradigms [7, 21] and studying its relation to other general programming methods such as greedy algorithms [3].

The analysis of molecular sequence data has fostered increased interest in DP. Protein homology search, RNA structure prediction, gene finding, and discovery of regulatory signals in RNA pose string processing problems in unprecedented

variety and data volume. A recent textbook in biosequence analysis [8] lists 11 applications of DP in its introductory chapter, and many more in the sequel.

Textbook examples are typically restricted to simple problems that can be solved without methodical guidance. Developing a DP algorithm for an optimization problem over a nontrivial domain has intrinsic difficulties. The choice of objective function and search space are interdependent, and closely tied up with questions of efficiency. Once completed, all DP algorithms are expressed via recurrence relations between tables holding intermediate results. These recurrences provide a very low level of abstraction, and subscript errors are a major nuisance even in published articles. The recurrences are difficult to explain, painful to implement, and almost impossible to debug: A subtle error gives rise to a suboptimal solution every now and then, which can hardly be detected by human inspection. In this situation it is remarkable that neither the literature cited above, nor computer science textbooks [1, 5, 6, 16, 18, 24] provide guidance in the development of DP algorithms.

1.2 Overview of our approach

The Algebraic Dynamic Programming approach (ADP) introduces a *conceptual* splitting of a DP algorithm into a recognition and an evaluation phase. The evaluation phase is specified by an *evaluation algebra*, the recognition phase by a *yield grammar*. Each grammar can be combined with a variety of algebras to solve different but related problems, for which heretofore DP recurrences had to be developed independently. Grammar and algebra together describe a DP algorithm on a high level of abstraction, supporting the development of ideas and the comparison of algorithms. A notation for yield grammars is provided that serves as a domain-specific language. Expressed in this language, the ADP algorithm can run as an executable prototype in a functional language, and can be translated into traditional DP recurrences implemented in an imperative language. Such implementation issues are treated in [14]. Here we concentrate on ADP as a programming method, and show how clever "tricks" found in the DP literature can be expressed transparently in this framework.

1.3 Related work

The ADP method has been developed recently in the application context of biosequence analysis. An informal presentation to the bioinformatics community has appeared in [10]. An old bioinformatics challenge, the folding of saturated RNA secondary structures [26], has been solved via ADP recently, but was published without reference to this method [9]. The aspect of ambiguity in dynamic programming (not covered here) is studied in [11].

2 Dynamic programming, traditional style

2.1 Palindromic patterns in strings

We consider strings over some alphabet. For $x = x_1 \dots x_m$ let $|x| = m$ and $x(i,j)$ denote the subword $x_{i+1} \dots x_j$. Note that $|x(i,j)| = j - i$. When x is clear from the context, we write (i,j) for $x(i,j)$. A string x is a palindrome, if $x = x^{-1}$. A separated palindrome [16] has the form $x = uvu^{-1}$, where we call v the "loop" of the palindrome, and u its "stem".

The string `panamacanal` is a separated palindrome with an empty stem, containing several non-trivial separated palindromes (e. g. `anamacana, acana`). Being a separated palindrome is not just a Yes-No question: The string `abccba` can be considered a separated palindrome in four different ways. Intuitively, $u =$ `abc`, $v = \varepsilon$ is more palindromic than (say) $u =$ `a`, $v =$ `bccb`. Let us introduce a scoring function $palScore(x)$, by means of which some palindromes score higher than others. Many scores are conceivable; for simplicity, we choose the length of the stem.

The two combinatorial optimization problems defined next will serve as running examples in this article.

Best separated palindrome (BSP):
Given x, find $palScore(x) = max\{|u| \mid uvu^{-1} = x\}$.

Local best separated palindrome (LBSP):
Given x, find $localpalScore(x) = max\{palScore(v) | uvw = x\}$.

$palScore(x)$ can be defined recursively, the basic fact being that awa is a palindrome of score $l + 1$ iff w is a palindrome of score l. In order to calculate $localpalScore(x)$, we must distinguish subwords that constitute a palindrome of certain score, and subwords that somewhere contain such a palindrome. We obtain recursive definitions for $palScore$ and $localpalScore$, using auxiliary functions p_x and q_x:

$$
\begin{array}{lll}
palScore(x) & = p_x(0,m) & \\
localpalScore(x) & = q_x(0,m) & \\
 & & \\
q_x(i,i) & = 0 & \text{for } i \in 0 \dots m \\
q_x(i,j) & = \max\{p_x(i,j), q_x(i+1,j), q_x(i,j-1)\} & \text{for } i < j \\
 & & \\
p_x(i,i) & = 0 & \text{for } i \in 0 \dots m \\
p_x(i,i+1) & = 0 & \text{for } i \in 0 \dots m-1 \\
p_x(i,j) & = \text{if } x_{i+1} = x_j \text{ then } 1 + p_x(i+1,j-1) \text{ else } 0 & \\
 & & \text{for } 0 \le i,j \le m \text{ and } i+2 \le j
\end{array}
$$

Implemented as recursive functions, p_x and q_x are inefficient, since most calls to $p_x(i,j)$ or $q_x(i,j)$ will be (re-)evaluated many times, leading to a runtime exponential in m. Dynamic programming is recursion plus tabulation. By a change of data type we achieve runtime $O(m^2)$. We re-interprete functions q_x and p_x as

two $m+1$ by $m+1$ matrices, whose entries are calculated (and re-used) via the above recurrences.

To judge the difficulty of developing this style of recurrences, the reader is invited to modify the recurrences as follows:

Exercise 1: Restrict analysis to palindromes with non-empty stem.

Exercise 2: Generalize the scoring scheme by introducing negative scores for non-stem parts of the palindrome.

2.2 Related problems

For reason of space, the simple problems[1] BSP and LBSP must suffice here for the exposition of our method. Further generalizations (allowed mismatches in the stem, recursively nested palindromes, and minimal free energy scoring [27]) leads to the problem of predicting RNA secondary structures.

Everything we develop in this paper pertains as well to the problem of pairwise string comparison, also called string edit distance or string alignment, where DP often is the method of choice. Although some definitions change in detail, the overall analogy is described by the following observation: An optimal alignment of x and y is equivalent to finding an optimal approximate separated palindrome with loop \$ for the string $x\$y^{-1}$, where \$ is a separator symbol not occuring elsewhere. We have not seen this observation exploited in the literature, maybe because the partial reversal of subscript ranges in forming $x\$y^{-1}$ makes the recurrences look quite different. In the ADP approach – where there are no subscripts – the correspondence is obvious. Pairwise sequence alignment is used for the exposition in [10].

3 Algebraic dynamic programming

3.1 Basic terminology

Alphabets. An *alphabet* $\mathcal{A}$ is a finite set of symbols. Sequences of symbols are called strings. ε denotes the empty string, $\mathcal{A}^1 = \mathcal{A}$, $\mathcal{A}^{n+1} = \{ax | a \in \mathcal{A}, x \in \mathcal{A}^n\}$, $\mathcal{A}^+ = \bigcup_{n \geq 1} \mathcal{A}^n$, $\mathcal{A}^* = \mathcal{A}^+ \cup \{\varepsilon\}$.

Signatures and algebras. A (single-sorted) signature Σ over some alphabet $\mathcal{A}$ consists of a sort symbol S together with a family of operators. Each operator o has a fixed arity $o : s_1 ... s_{k_o} \rightarrow S$, where each s_i is either S or $\mathcal{A}$. A Σ-algebra $\mathcal{I}$ over $\mathcal{A}$, also called an interpretation, is a set $\mathcal{S}_\mathcal{I}$ of values together with a function o_I for each operator o. Each o_I has type $o_I : (s_1)_I ... (s_{k_o})_I \rightarrow S_I$ where $\mathcal{A}_I = \mathcal{A}$.

A *term algebra* T_Σ arises by interpreting the operators in Σ as *constructors*, building bigger terms from smaller ones. When variables from a set V can take the place of arguments to constructors, we speak of a term algebra with variables,

[1] In fact, for the given simplistic score function, BSP and LBSP can be solved more efficiently using suffix trees than via dynamic programming.

$T_\Sigma(V)$, with $V \subset T_\Sigma(V)$. By convention, operator names are capitalized in the term algebra.

Tree grammars. Terms will be viewed as rooted, ordered, node-labeled trees in the obvious way. According to the special role of $\mathcal{A}$, only leaf nodes can carry symbols from $\mathcal{A}$. A term/tree with variables is called a *tree pattern.* A tree containing a designated occurrence of a subtree t is denoted $C[...t...]$.

A tree language over Σ is a subset of T_Σ. Tree languages are described by tree grammars, which can be defined in analogy to the Chomsky hierarchy of string grammars. Here we use regular tree grammars originally studied in [4]. In [13] they were redefined to specify term languages over some signature. Our further specialization so far lies solely in the distinguished role of $\mathcal{A}$.

Definition 1 (Tree grammar over Σ and $\mathcal{A}$.)
A (regular) tree grammar $\mathcal{G}$ over Σ and $\mathcal{A}$ is given by

- a set V of nonterminal symbols
- a designated nonterminal symbol Ax called the axiom
- a set P of productions of the form $v \to t$, where $v \in V$ and $t \in T_\Sigma(V)$.

The derivation relation for tree grammars is $\to^*$, with $C[...v...] \to C[...t...]$ if $v \to t \in P$. The language of $v \in V$ is $\mathcal{L}(v) = \{t \in T_\Sigma | v \to^* t\}$, the language of $\mathcal{G}$ is $\mathcal{L}(\mathcal{G}) = \mathcal{L}(Ax)$.□

For brevity, we add a lexical level to the grammar concept, allowing strings from $\mathcal{A}^*$ in place of single symbols. By convention, `char` denotes an arbitrary character, `string` an arbitrary string. Also for brevity, we allow syntactic conditions associated with righthand sides. See Figure 1.

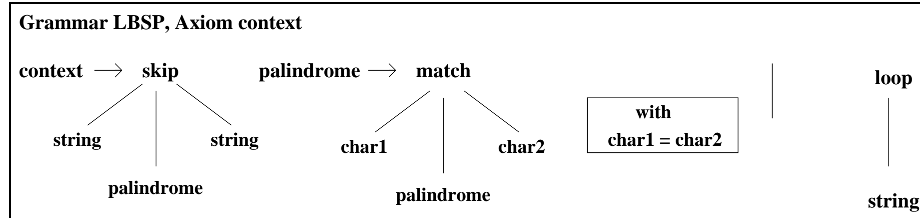

Fig. 1. The tree grammar LBSP for local separated palindromes (see Section 2.1).

The yield of a tree is normally defined as its sequence of leaf symbols. Here we are only interested in the symbols from $\mathcal{A}^*$; nullary constructors are not considered part of the yield. Hence the yield function y on $T_\Sigma(V)$ is defined by $y(t) = w$, where $w \in (\mathcal{A} \cup V)^*$ is the string of leaf symbols in left to right order.

3.2 Representing the search space by a term algebra

Any dynamic programming algorithm implicitly constructs a search space from its input. The elements of this search space have been given different names:

policy in [2], *solution* in [22], *subject under evaluation* in [10]. Since the former two terms have been used ambiguously, and the latter is rather technical, we shall use the term *candidate* for elements of the search space. Each candidate will be evaluated, yielding a *final state*, a *cost*, or a *score*, depending whether you follow [2], [22] or [8]. We shall use the term *answer* for the result of evaluating a candidate.

Typically, there is an ordering defined on the answer data type. The DP algorithm returns a maximal or minimal answer, and if so desired, it also reports one or all the candidates that evaluate(s) to this answer. Often, the optimal answer is determined first, and a candidate that led to it is reconstructed by backtracking. The candidates themselves do not have an explicit representation during the DP computation. This is the point that we will change.

Imagine that during computing the answer, we did not actually call those functions that perform the evaluation. Instead, we would apply them symbolically, building up a formula that – once evaluated – would yield this answer value. We shall make this formula an explicit representation of the candidate. For LBSP this is done in Figure 2.

These are the key ideas of algebraic dynamic programming:

Phase separation: We conceptually distinguish a recognition and an evaluation phase.

Term representation: Individual candidates are represented as elements of a term algebra T_Σ; the set of all candidates is described by a tree grammar.

Recognition: The recognition phase constructs the set of candidates arising from a given input string, using a device called tabulating yield parser.

Evaluation: The evaluation phase interprets these candidates in a concrete Σ-algebra, and applies the objective function to the resulting answers.

Phase amalgamation: To retain efficiency, both phases are amalgamated in a fashion transparent to the programmer.

The virtue of this approach is that the conglomeration of issues bemoaned above – the recurrences deal with search space construction, evaluation and efficiency concerns in a non-separable way – is resolved by algorithm development on the more abstract level of grammars and algebras. In Section 5 we explain the benefits from the viewpoint of programming methodology, while [14] shows how an ADP algorithm, in spite of its abstractness, can be implemented without loss of efficiency.

3.3 Evaluation algebras

Definition 2 (Evaluation algebra.) Let Σ be a signature over $\mathcal{A}$ with sort symbol Ans. A Σ-evaluation algebra is a Σ-algebra augmented with an objective function $h : [Ans] \to [Ans]$, where $[Ans]$ denotes lists over Ans. □

In most DP applications, the objective function minimizes or maximizes over all answers. We take a slightly more general view here. The objective may be to calculate a sample of answers, or all answers within a certain threshold of

optimality. It could even be a complete enumeration of answers. We may compute the size of the search space or evaluate it in some statistical fashion, say by averaging over all answers. This is why in general, the objective function will return a list of answers. If maximization was the objective, this list would hold the maximum as its only element.

We formulate a signature Π for the palindrome example[2]:

match:	$\mathcal{A}$	$\times$	Pal $\times$	$\mathcal{A}$	->	Pal	skipleft:	$\mathcal{A}^*$	$\times$	Pal	->	Pal
loop:	$\mathcal{A}^*$				->	Pal	skipright:	Pal	$\times$	$\mathcal{A}^*$	->	Pal
skip:	$\mathcal{A}^*$	$\times$	Pal $\times$	$\mathcal{A}^*$	->	Pal	skipl:	$\mathcal{A}$	$\times$	Pal	->	Pal
							skipr:	Pal	$\times$	$\mathcal{A}$	->	Pal

```
skip("p",match('a',match('n',match('a',loop("mac"),'a'),'n'),'a'),"l")
```

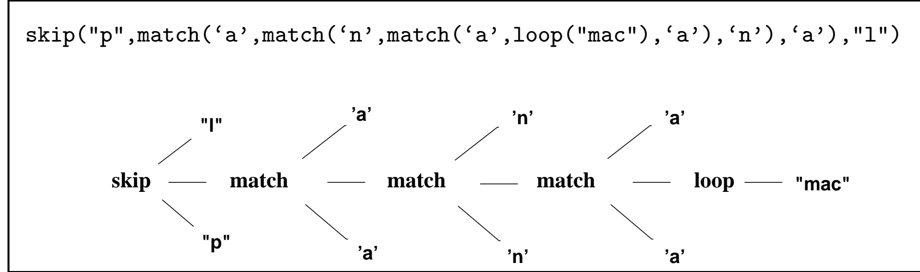

Fig. 2. The term representation of a LBSP candidate c for the input sequence `panamacanal`, and the tree representation of this term (lying on its side).

We specify two Π-algebras (see Figure 3). The evaluation algebra UNIT uses unit score whereas the algebra WEIGHT uses a weight function $(w(a, b))$ to score a match depending on the corresponding characters. While UNIT ignores the length of the separating loop and the skipped characters, WEIGHT scores skipped prefix and suffix according to their length and some (negative) parameters α and β, and the loop length with a specific function `lscore`. UNIT corresponds to the recurrences in Section 2.1, while WEIGHT solves Exercise 2.

Ans_{UNIT}	= $\mathbb{N}$	Ans_{WEIGHT}	= $\mathbb{N}$
`match(a,s,b)`	= `s + 1`	`match(a,s,b)`	= `s + w(a,b)`
`loop(s)`	= 0	`loop(s)`	= `lscore(length(s))`
`skip(xs,s,ys)`	= `s`	`skip(xs,s,ys)`	= `s +` α `* lenght(xs)` β `* length(ys)`
`h([])`	= `[]`	`h([])`	= `[]`
`h (l)`	= `[maximum(l)]`	`h (l)`	= `[maximum(l)]`

Fig. 3. Algebras UNIT (left) and WEIGHT (right)

For candidate c in Figure 2, we obtain

$$c_{UNIT} = 3$$
$$c_{WEIGHT} = 2w(`a`, `a`) + w(`n`, `n`) + lscore(3) + \alpha + \beta.$$

[2] The four operators on the right side are not used until later.

3.4 Yield grammars

We obtain an explicit and transparent definition of the search space of a given DP problem by a change of view on tree grammars and parsing:

Definition 3 (Yield grammars and yield languages.) Let $\mathcal{G}$ be a tree grammar over Σ and $\mathcal{A}$, and y the yield function. The pair $(\mathcal{G}, y)$ is called a yield grammar. It defines the yield language $\mathcal{L}(\mathcal{G}, y) = y(\mathcal{L}(\mathcal{G}))$. □

Definition 4 (Yield parsing.) Given a yield grammar $(\mathcal{G}, y)$ over $\mathcal{A}$ and $w \in \mathcal{A}^*$, the yield parsing problem is to find $P_\mathcal{G}(w) := \{t \in \mathcal{L}(\mathcal{G}) | y(t) = w\}$. □

The search space spawned by input w is $P_\mathcal{G}(w)$. Yield parsing is the computational engine underlying ADP. Its efficient implementation is explained in [14]. For the present development, we assume the availability of a correct and efficient yield parser.

3.5 Algebraic dynamic programming and Bellman's principle

Given that yield parsing traverses the search space, all that is left to do is evaluate candidates in some algebra and apply the objective function.

Definition 5 (Algebraic dynamic programming.)

- An ADP problem is specified by a signature Σ over $\mathcal{A}$, a yield grammar $(\mathcal{G}, y)$ over Σ, and a Σ-evaluation algebra I with objective function h_I.
- An ADP problem instance is posed by a string $w \in \mathcal{A}^*$. The search space it spawns is the set of all its parses, $P_\mathcal{G}(w)$.
- Solving an ADP problem is computing

$$h_\mathcal{I}\{t_\mathcal{I} \mid t \in P_\mathcal{G}(w)\}$$

  in polynomial time and space.

□

There is one essential ingredient missing: efficiency. Since the size of the search space may be exponential in terms of input size, an ADP problem can be solved in polynomial time and space only under a condition known as Bellman's principle of optimality. In his own words:

> "An optimal policy has the property that whatever the initial state and initial decision are, the remaining decisions must constitute an optimal policy with regard to the state resulting from the first decision." [2]

We formalize this principle:

Definition 6 (Algebraic version of Bellman's principle.) For each k-ary operator f in Σ, and all answer lists $z_1, \dots, z_k$, the objective function h satisfies

$$\begin{aligned} & h(\ [\ f(x_1, \dots, x_k) \mid x_1 \leftarrow z_1, \dots, x_k \leftarrow z_k\ ]\ ) \\ = & h(\ [\ f(x_1, \dots, x_k) \mid x_1 \leftarrow h(z_1), \dots, x_k \leftarrow h(z_k)\ ]\ ) \end{aligned}$$

Furthermore, the same property holds for the concatenation of answer lists:

$$h(\ z_1 +\!\!+\ z_2\ ) = h(\ h(z_1) +\!\!+\ h(z_2)\ )$$

□

The practical meaning of the optimality principle is that we may push the application of the objective function inside the computation of subproblems, thus preventing combinatorial explosion. Phase amalgamation is achieved by a yield parser that uses $\mathcal{I}$ in place of Σ (computing answers instead of trees), and applies $h_{\mathcal{I}}$ to intermediate answer lists. For the sake of efficiency, we shall annotate the tree grammar to indicate the cases where h is to be applied.

Summarizing, we state that an ADP algorithm is completely specified by the annotated yield grammar and the concrete evaluation algebra. Compared to the classic description of DP algorithms via matrix recurrences we have achieved the following:

– An ADP specification is *more abstract* than the traditional recurrences. Separation between search space construction and evaluation is perfect. Tree grammars and evaluation algebras can be combined in a modular way, and the relationships between problem variants can be explained clearly.
– The ADP specification is also *more complete*: DP algorithms in the literature often claim to be parametric with respect to the scoring function, while the initialisation equations are considered part of the search algorithm [8]. In ADP, it becomes clear that initialisation semantically is the evaluation of empty candidates, and it is specified within the algebra.
– Our formalization of Bellman's principle is *more general* than commonly seen. Objectives like complete enumeration or statistical evaluation of the search space now fall under the framework. If maximization is the objective, our criterion implies Morin's formalization (strict monotonicity) [22] as a special case.
– The ADP specification is *more reliable*. The absence of subscripts excludes a large class of errors that are traditionally hard to find.

4 The ADP programming system

We have implemented the ADP approach as a practical program development system. Here, we only sketch a part of the ADP language. Its implementation, described in [14], provides an executable prototype in Haskell, and a translation to efficient C-code. We also plan to provide the translation of an ADP specification into the traditional recurrences[3], formatted in LaTeX.

[3] A concession to traditionalistic attitudes in the bioinformatics community.

4.1 An ASCII notation for annotated yield grammars

Alike EBNF, productions in yield grammars are written as equations. The operator `<<<` is used to denote the application of a tree constructor to its arguments, which are chained via the `~~~`-operator. Operator `|||` separates multiple right-hand sides of a nonterminal symbol. The operator `...` indicates application of the objective function. Operator priority in decreasing order is {`~~~` , `<<<`, `|||` , ...}[4]. Parentheses are used as required for larger trees. The axiom symbol is indicated by the keyword `axiom`, and conditions are attached to productions via the keyword `with`. Using this notation, we write grammar LBSP0:

```
grammarLBSP0 = axiom context
context      = skip  <<< string ~~~ palindrome ~~~ string            ... h
palindrome   = match <<< char   ~~~ palindrome ~~~ char    with equal |||
               loop  <<< string                                      ... h
```

The annotation `"... h"` declares that the objective function is to be applied to answer lists resulting from the annotated productions. Some further annotation, not shown here, indicates which intermediate results are to be tabulated.

`grammarLBSP0` can be combined with both evaluation algebras of Section 3.3 to solve LBSP under either scoring scheme. But the grammar `grammarLBSP0` allows for multiple representations of the trivial palindrome with empty stem. Such ambiguity is avoided by the following refined grammar LBSP1:

```
grammarLBSP1 = axiom start
start        = loop  <<< string ||| context                          ... h
context      = skip  <<< string ~~~ palindrome ~~~ string            ... h
palindrome   = match <<< char   ~~~ pal_inner  ~~~ char  with equal
pal_inner    = match <<< char   ~~~ pal_inner  ~~~ char  with equal |||
               loop  <<< string                                      ... h
```

4.2 Efficiency of ADP programs

From the viewpoint of programming methodology, it is important that asymptotic efficiency can be analyzed and controlled on the abstract level. This property is a major virtue of ADP – it allows to formulate efficiency tuning as grammar and algebra transformations.

If not reduced by the objective function `h`, the number of answers is exponential in terms of input size, and dominates efficiency. Typically however, the objective function is applied to reduce the number of answers wherever necessary. In this case, efficiency is determined by the yield grammar:

Definition 7 (Width of productions and grammar.) Let t be a tree pattern, and let k be the number of nonterminal or lexical symbols in t whose yield size is not bounded by a constant. We define $width(t) = k - 1$. Let π be a production $v \rightarrow t_1 | \dots | t_r$. $width(\pi) = max\{width(t_1, \dots, t_r)\}$, and $width(\mathcal{G}) = max\{width(\pi) \mid \pi \text{ production in } \mathcal{G}\}$. □

[4] The implementation takes a different view that need not concern us here.

Theorem 8 Assuming the number of answers is bounded by a constant, the execution time of an ADP algorithm described by tree grammar $\mathcal{G}$ on input w of length n is $O(n^{2+width(\mathcal{G})})$.

Proof: See [14] □

5 Programming methodology

We demonstrate the benefits of the algebraic approach to dynamic programming by discussing issues of algorithm design, tuning and testing. Several "tricks" known from the DP literature can now be formulated as general techniques, i.e., in a problem independent fashion.

5.1 Problem variation, systematic testing and re-use: multiple algebras

The fact that a given grammar can be combined with different evaluation algebras has a variety of uses. Beside the two scoring algebras presented in Section 3.3 we formulate an algebra for the enumeration of all answers, an algebra to count the total number of answers, and an algebra to compute the expected number of answers $E_m(n, c)$ in a string of given length n and character composition c (see Figure 4). The function `p` gives the probability of two characters in the input string to be the same; it is given by a background distribution, or calculated using the character composition of the input string.

Ans_{ENUM}	$= T_\Pi$	Ans_{COUNT}	$= \mathbb{N}$	Ans_{EXP}	$= \mathbb{R}$
match	= Match	match(a,s,b)	= s	match(a,s,b)	= s * p(a,b)
loop	= Loop	loop(s)	= 1	loop(s)	= 1
skip	= Skip	skip(xs,s,ys)	= s	skip(xs,s,ys)	= s
h	= id	h([])	= []	h([])	= []
		h([x_1,...,x_r]) = [x_1+...+x_r]		h([x_1,...,x_r]) = [x_1+...+x_r]	

Fig. 4. Enumeration algebra (left), counting (middle) and expectation algebra (right)

Techniques:

Testing: The enumeration algebra allows to inspect the search space of our algorithm, as a means to review design decisions.
Search space combinatorics: The counting algebra is a flexible approach to the combinatorics of structures, supporting mathematical search space analysis.
Significance: The expectation algebra is an elegant way to obtain significance values in string pattern matching.

A deeper study of expectation algebras and their use for significance analysis is found in [19].

5.2 Connecting related problems: from global to local pattern matching

Grammar LBSP describes (local) separated palindromes embedded somewhere in a string. Dropping the first production and making `palindrome` the axiom, we obtain a grammar BSP describing strings that themselves are separated palindromes. This observation allows to formulate a general way to move from global to local pattern matching (or vice versa):

> *Technique: Global-local transformation*
> Let $\mathcal{G}$ be a grammar with axiom A_x, describing a pattern language. Adding a new axiom A'_x and the production
> A'_x `= skip <<< string ~~~` A_x `~~~ string`
> yields a grammar for the corresponding local patterns.

For readers familiar with the algorithms used in biosequence analysis: This is, in general terms and for arbitrary sequence analysis problems, the transition from the Needleman-Wunsch [23] to the Smith-Waterman algorithm [25]; however, for an input of form $x\$y^{-1}$ it must be applied at both ends of x and y.

5.3 Controlling efficiency: width reduction

The asymptotic efficiency of an ADP algorithm is $O(n^{2+width(\mathcal{G})})$. Two techniques are available to improve asymptotic efficiency. Both can be described as grammar transformations; they require that a complementary transformation of the evaluation algebra is possible.

Splitting productions Note that $width(LBSP) = 2$ and hence the runtime is $O(n^4)$ due to the first production. We modify the LBSP grammar such that the recognition of non-matched characters is split in one part for the non-matched prefix (`left_context`) and another for the suffix (`right_context`). In UNIT, we add the definitions `skipleft(xs,s) = s` and `skipright(s,ys) = s`.

This leads to a grammar with width 1 and execution time $O(n^3)$.

```
grammarLBSP2  = axiom start
start         = loop  <<< string ||| left_context                  ... h
left_context  = skipleft  <<< string     ~~~ right_context         ... h
right_context = skipright <<< palindrome ~~~ string                ... h
palindrome    = match <<< char ~~~ pal_inner ~~~ char  with equal
pal_inner     = match <<< char ~~~ pal_inner ~~~ char  with equal |||
                loop  <<< string                                   ... h
```

Technique: Production splitting

The transformation of

```
u = f <<< a ~~~ b ~~~ c ... h     to    u = f¹ <<< a ~~~ u1 ... h
                                       u1 = f² <<< b ~~~ c ... h
```

is correct iff:

(i) $f_I(a,b,c) = {f^1}_I(a, {f^2}_I(b,c))$

(ii) ${f^1}_I$ and ${f^2}_I$ satisfy Bellman's principle (Def. 6) with respect to h_I.

Splitting words Production splitting reduces the grammar to productions containing at most two symbols with unbounded yield. Further improvement can be achieved by replacing terminal symbols with unbounded yield size by terminal symbols with bounded yield size and an extra recursion. In the palindrome example, we modify the tree productions that deal with the skipped prefix and suffix, such that each production contains just one nonterminal of unbounded yield size. Extending UNIT, the operators `skipl` and `skipr` included in the signature Π (Section 3.3) are defined as `skipl(a,s) = s` and `skipr(s,a) = s`.

This gives us an ADP algorithm with $width(\mathcal{G}) = 0$ and hence quadratic execution time.

```
grammarLBSP3 = axiom start
start        = loop  <<< string ||| context                           ... h
context      = skipl <<< char ~~~ context |||
               skipr <<< context ~~~ char ||| palindrome               ... h
palindrome   = match <<< char ~~~ pal_inner ~~~ char  with equal
pal_inner    = match <<< char ~~~ pal_inner ~~~ char  with equal |||
               loop  <<< string                                        ... h
```

Combining grammar LBSP3 with scoring algebra UNIT, this ADP algorithm is equivalent to the recurrences given for *localpalScore* in Section 2.1. Making `pal_inner` the axiom of grammar LBSP3, it is equivalent to the recurrences given for *palScore*.

Technique: Word splitting

The transformation of

u = f <<< $a_1 ... a_r$ ~~~ v ... h $\qquad a_1 ... a_r \in \mathcal{A}^*$

into

u = f^1 <<< a ~~~ u ||| f^2 <<< v ... h $\qquad a \in A$

is applicable iff

(i) $f_I(a_1...a_r, v) = f^1_I(a_1, ..., f^1_I(a_r, f^2_I(v))...)$

(ii) ${f^1}_I$ and ${f^2}_I$ satisfy Bellman's principle (Def. 6) with respect to h_I.

A famous application of this technique is the use of affine gap scores in [15]. The scoring function $f(a_1...a_r, u) = r \times \alpha + \beta + u$ is decomposed into separate functions for gap opening $f^2(u) = \beta + u$ and extension $f^1(\alpha, u) = \alpha + u$.

5.4 The taming of the near-optimal solution space

In many applications, one is interested not only in an optimal answer, but also in near-optimal answers. The difficulty is that their number grows exponentially even in the vicinity of the optimum. We discuss two techniques to deal with this situation. Without loss of generality we assume in this section that the objective function is minimization.

k-best answers An obvious way to compute the k best answers is to use an objective function $\texttt{min}_k$, which keeps the k best answers for each subproblem. This only affects the evaluation algebra: replace the definition `h = min` by `h =` $\texttt{min}_k$.

Sorted lazy enumeration A much more elegant approach is the following: Assume answers can be computed in the form of sorted lazy lists (streams). Then, we can enumerate a stream of answers in order of optimality, with negligible overhead for the best answers. This solution has been hinted at occasionally [7, 17]. Since we have not seen it implemented, we sketch here how this can be achieved.

The key idea is to compute separate lists, i.e. `[[f x y | y <-ys] | x <-xs]` instead of `[f x y | y <- ys , x <- xs]`. The resulting lists are sorted because Bellman's principle implies strict monotonicity of all functions in the evaluation algebra. The sorted lists are combined using the lazy merge function.

For shortness we give the exact definition in Haskell notation. Let `p` and `q` be yield parsers, returning sorted lists of answers when applied to a subword (i, j) of the input. Answers may be not just scalar values, but also curried functions. We define their combination as follows:

```
(p ||| q) (i,j) = merge (p(i,j)) (q(i,j))
(p ~~~ q) (i,j) = foldr merge [] [foldr merge [] [[x y | y <- q (k,j)]
                                                  | x <- p (i,k)]
                                                  | k <-[i..j]]
```

Techniques:

k-best enumeration: In the evaluation algebra, use `h =` $\texttt{min}_k$.
sorted lazy enumeration: In the evaluation algebra, use `h = id` and use lazy merge yield parsing implementation.

6 Conclusion

What have we achieved in terms of putting dynamic programming over sequence data on a firm mathematical basis? The algebraic nature of our approach is essential, even where there is no deep algebraic reasoning behind it. (However, counting and statistical evaluation algebras become more intricate when the

modelled search space is not isomorphically embedded in T_Σ.) Casting the candidates of the search space into a representation as an algebraic data type opens the path to separate the description of the search space from its evaluation. This idea is quite general, and it appears that ADP is applicable to all DP problems over strings, including pairwise comparison, and possibly other domains. (We are developing DP algorithms for tree comparison, but this work has not yet been reformulated in ADP style [12].) Correctness arguments can now be given on a convenient level of abstraction – yield languages and evaluation algebras satisfying our new formulation of Bellman's principle. By virtue of Theorem 8, abstractness does not come at the price of loosing efficiency control.

We can give a clear methodical guidance for the development of successful DP recurrences. (Note that Steps 4 – 7 are actually independent.)

1. Design the signature Σ, representing explicitly all cases that might influence evaluation.
2. Design evaluation algebras, at least the enumeration, counting and one scoring algebra, describing the objective of the application.
3. Design the grammar defining the search space.
4. Improve efficiency by grammar and algebra transformation techniques.
5. Experiment with algorithmic ideas using the functional prototype.
6. Test the design via search space enumeration, and plausibility checking via counting algebra.
7. Generate C implementation and validate against the functional prototype.

Returning to our initial quotation of an anonymous referee from the bioinformatics community, we feel that we no longer need luck to find and implement correct DP recurrences. We can express and communicate our experience (and that of others) on an adequate level of abstraction, and devote our talent to tackling ever more ambitious DP problems.

References

1. A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *Data Structures and Algorithms.* Addison-Wesley, Reading, MA, USA, 1983.
2. R. Bellman. *Dynamic Programming.* Princeton University Press, 1957.
3. R. S. Bird and O. de Moor. From dynamic programming to greedy algorithms. In B. Moeller, editor, *State-of-the-Art Seminar on Formal Program Development.* Springer LNCS 755, 1993.
4. W. S. Brainerd. Tree generating regular systems. *Information and Control*, 14:217–231, 1969.
5. G. Brassard and P. Bratley. *Algorithmics: Theory and Practice.* Prentice-Hall, 1988.
6. T.H. Cormen, C.E. Leiserson, and R.L. Rivest. *Introduction to Algorithms.* MIT Press, Cambridge, MA, 1990.
7. S. Curtis. Dynamic programming: A different perspective. In R. Bird and L. Meertens, editors, *Algorithmic Languages and Calculi*, pages 1–23. Chapman & Hall, London, U.K., 1997.

8. R. Durbin, S. Eddy, A. Krogh, and G. Mitchison. *Biological Sequence Analysis*. Cambridge University Press, 1998.
9. D. Evers and R. Giegerich. Reducing the conformation space in RNA structure prediction. In *German Conference on Bioinformatics*, 2001.
10. R. Giegerich. A Systematic Approach to Dynamic Programming in Bioinformatics. *Bioinformatics*, 16:665–677, 2000.
11. R. Giegerich. Explaining and controlling ambiguity in dynamic programming. In *Proc. Combinatorial Pattern Matching*, pages 46–59. Springer Verlag, 2000.
12. R. Giegerich, M. Höchsmann, and S. Kurtz. Local Similarity Problems on Trees: A Uniform Model and its Implementation. 2002. (submitted).
13. R. Giegerich and K. Schmal. Code selection techniques: Pattern matching, tree parsing and inversion of derivors. In *Proc. European Symposium on Programming 1988*, pages 247–268. Springer LNCS 300, 1988.
14. R. Giegerich and P. Steffen. Implementing algebraic dynamic programming in the functional and the imperative paradigm. In E.A. Boiten and B. Möller, editors, *Mathematics of Program Construction*, pages 1–20. Springer LNCS 2386, 2002.
15. O. Gotoh. An improved algorithm for matching biological sequences. *J. Mol. Biol.*, 162:705–708, 1982.
16. D. Gusfield. *Algorithms on Strings, Trees, and Sequences*. Cambridge University Press, 1997.
17. S. Kurtz. *Fundamental Algorithms for a Declarative Pattern Matching System*. Dissertation, Technische Fakultät der Universität Bielefeld, 1995.
18. K. Mehlhorn. *Data structures and algorithms*. Springer Verlag, 1984.
19. C. Meyer and R. Giegerich. Matching and Significance Evaluation of Combined Sequence-Structure Motifs in RNA. *Z.Phys.Chem.*, 216:193–216, 2002.
20. L. Mitten. Composition principles for the synthesis of optimal multi-stage processes. *Operations Research*, 12:610–619, 1964.
21. O. de Moor. Dynamic Programming as a Software Component. In M. Mastorakis, editor, *Proceedings of CSCC, July 4-8, Athens*. WSES Press, 1999.
22. T. L. Morin. Monotonicity and the principle of optimality. *Journal of Mathematical Analysis and Applications*, 86:665–674, 1982.
23. S. B. Needleman and C. D. Wunsch. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *J. Mol. Biol.*, 48:443–453, 1970.
24. R. Sedgewick. *Algorithms*. Addison-Wesley, 2nd edition, 1989.
25. T. F. Smith and M. S. Waterman. The identification of common molecular subsequences. *J. Mol. Biol.*, 147:195–197, 1981.
26. M. Zuker and S. Sankoff. RNA secondary structures and their prediction. *Bull. Math. Biol.*, 46:591–621, 1984.
27. M. Zuker and P. Stiegler. Optimal computer folding of large RNA sequences using thermodynamics and auxiliary information. *Nucleic Acids Res.*, 9(1):133–148, 1981.