Bellman’s GAP - A Declarative Language for Dynamic Programming

Georg Sauthoff  Stefan Janssen  Robert Giegerich
Universität Bielefeld
Technische Fakultät, 33501 Bielefeld, Germany
{gsauthof, sjanssen, robert}@techfak.uni-bielefeld.de

Abstract
Dynamic programming is a well-established technique to solve combinatorial optimization problems. In several areas of applied computer science, such as operations research, natural language processing, or biosequence analysis, dynamic programming problems arise in many variations and with a considerable degree of sophistication. The simple way dynamic programming problems are normally presented in computer science textbooks – as a set of tabular recurrences – scales poorly for real world problems, where the search space is deeply structured and the scoring model is elaborate. Coming up with pages of correct recurrences is difficult, implementation is error-prone, and debugging is tedious. Algebraic Dynamic Programming (ADP) is a language-independent, declarative approach which alleviates these problems for a relevant class of dynamic programming problems over sequence data.

Bellman’s GAP implements ADP by providing a declarative language (GAP-L) with a Java-reminiscent syntax, and a compiler (GAP-C) translating declarative programs into C++ code, which is competitive to handwritten code, and arguably more reliable. This article introduces the GAP-L language, demonstrates the benefits of developing dynamic programming algorithms in a declarative framework by educational example, and reports on the practice of programming bioinformatics applications with Bellman’s GAP.

Categories and Subject Descriptors D.3 [PROGRAMMING LANGUAGES]: Language Classifications—Specialized application languages

General Terms Languages, Algorithms

Keywords Declarative Programming, Dynamic Programming, Regular Tree Grammars, Algebras, RNA Structure Prediction

1. Introduction
Difficulties with dynamic programming Dynamic programming is a widely used technique to solve combinatorial optimization problems. Often, it allows to evaluate a search space of exponential size in polynomial time. Variations of the basic algorithm return not only an optimal solution, but may also report co- or near-optimal solutions, or compute synoptic properties of the search space as a whole, such as its size or sum of all scores. They may sample the search space stochastically, or partition it into certain classes of interest and perform either one of the above analyses class-wise, and so on.

In several areas of applied computer science, such as operations research, natural language processing, or biosequence analysis, dynamic programming problems arise in many variations and with a considerable degree of sophistication. The simple way in which dynamic programming problems are normally presented in computer science textbooks – as a handful of tabular recurrences – scales poorly for real world problems, where the search space is deeply structured, the scoring model elaborate, and multiple objective functions may be used in combination. Coming up with several pages of correct recurrences is difficult, their implementation is error-prone, and debugging is tedious. For efficiency reasons, optimization is a separate first phase, followed by a backtracing stage to retrieve the solution associated with the optimal score. More comprehensive analyses, such as complete backtracing for the p percent near-optimal solutions, or optimization in a class-wise fashion, considerably add to the implementation effort.

Separating concerns Algebraic Dynamic Programming (ADP) [9] is a language-independent, declarative approach that alleviates these problems for a relevant class of dynamic programming algorithms, namely those over sequence data. The remedy is a perfect separation of four concerns that are traditionally expressed in the recurrences in an intermingled fashion: search space definition, candidate scoring, optimization objective, and tabulation issues (the last determines runtime and space efficiency). The central idea behind the algebraic approach can be introduced by a simple example: consider two strings $x = x_1 \ldots x_m$ and $y = y_1 \ldots y_n$. Their edit distance can be computed via the recurrences

\begin{align*}
\text{dist}(0, 0) &= 0 \quad (1) \\
\text{dist}(i, 0) &= \text{dist}(i - 1, 0) + \text{del}(x_i), 1 \leq i \leq m \quad (2) \\
\text{dist}(0, j) &= \text{dist}(0, j - 1) + \text{ins}(y_j), 1 \leq j \leq n \quad (3) \\
\text{dist}(i, j) &= \min \left\{ \begin{array}{ll}
\text{match}(x_i, y_j) & + \text{dist}(i - 1, j - 1) \\
\text{del}(x_i) & + \text{dist}(i - 1, j) \\
\text{ins}(y_j) & + \text{dist}(i, j - 1)
\end{array} \right. \quad (4)
\end{align*}

Here, $\text{del}$ and $\text{ins}$ are the scoring functions for deletions and insertions, and $\text{match}$ scores character replacements. The boundary conditions, Equations 1-3, indicate structural recursion over the two input sequences, but where does the three-fold case distinction in Equation 4 come from? It reflects the structure of a solution candidate, i.e. an alignment that edits $x$ into $y$. It recurs on an invisible data structure, which is evaluated without being constructed. To
make this view explicit, the last equation can be rewritten into

\[
\text{dist}(\eta) = \min \left[ \text{dist}(\eta_1), \text{dist}(\eta_2), \text{dist}(\eta_3) \right] = \\
\min \left\{ \text{match}(x_1, y_1) + \text{dist}(\zeta_1), \text{del}(x_i) + \text{dist}(\zeta_2), \text{ins}(y_j) + \text{dist}(\zeta_3) \right\}
\]

where \( \eta_1 = \text{Match}(x_1, \zeta_1, y_1) \)
\( \eta_2 = \text{Del}(x_i, \zeta_2) \)
\( \eta_3 = \text{Ins}(\zeta_3, y_j) \)

Here, \( \eta \) denotes the invisible candidate(s) for a subword, and \( \zeta \) the invisible candidate on which a right-hand side call to \( \text{dist} \) recurs. Note that the use of subscripts with \( \eta \) and \( \zeta \) becomes implicit, since, if \( \eta \) is derived from subproblem \( (x_1, \ldots, y_j) \), it is clear in each case which characters \( x_i \) or \( y_j \) are consumed by the local case analysis, and that \( \zeta \) is derived from the remaining subproblem.

The ADP approach makes the invisible candidate structure explicit. Candidates are modeled as trees. The function symbols used at inner tree nodes reflect the designer’s case analysis. They are seen as tree constructors on the specification level, and will be interpreted (called) as scoring functions at runtime. They are collected in a signature which serves as the interface between two (otherwise) independent specification components: grammar and algebras.

The search space is defined by a tree grammar. Candidate scoring is done by an evaluation algebra, implementing the score functions and the optimization objective. Tabulation issues are hidden from the programmer, thus there are no subscripts any more, and hence no subscript errors. Different tree grammars may share evaluation algebras, or different evaluation algebras may be used with one grammar. In particular, products of evaluation algebras give rise to new evaluation algebras, a feature providing re-use of components and a great convenience in practice. An implementation of this approach must take responsibility to generate efficient dynamic programming code, given these declarative constituents.

The Bellman’s GAP system  
Bellman’s GAP implements ADP. It provides a declarative language (GAP-L) with a Java-reminiscent syntax, in which the programmer specifies tree grammars and evaluation algebras. The language provides three types of evaluation algebra products, allowing to derive more sophisticated analyses from tested components without the need for any reprogramming. Furthermore, it includes a number of pragmatic extensions to the ADP approach, such as generic evaluation algebras and multi-track input. The Bellman’s GAP compiler (GAP-C) translates declarative programs into C++ code, which is competitive to handwritten code, and arguably more reliable. It performs extensive optimizations to achieve optimal asymptotic space and time efficiency, with reasonable constant factors.

In this article

- we recall (in a compact form) the principles of ADP, and report on their extension by new product operations,
- we introduce the GAP-L language, which bridges the gap between the abstract definitions and the practice of ADP,
- we report on the experiences with ex-bedding ADP from Haskell,
- we demonstrate the benefits of program development in Bellman’s GAP by educational example,
- we report from the practice of programming bioinformatics software in GAP-L,
- we conclude with a short list of theoretical and practical research topics which emerge from our experience with Bellman’s GAP, and
- the optimizations of the GAP-C compiler are described elsewhere [10].

Relation to previous work  
During the early development of ADP, the approach was implemented as a combinator language in Haskell [7, 8]. Efficiency concerns with applications emerging around 2004 [12, 18, 19] motivated the implementation of a compiler, which translated the Haskell-embedded notation into more efficient C code [11]. However, the Haskell syntax turned out to be an obstacle to the wider acceptance of the method within the bioinformatics community, in spite of the evident increase in programmer productivity. Also, the fragile borders of an embedded language encouraged algorithm designers to incorporate non-ADP features from the host language, resulting in an unfortunate mixture of automated translation and subsequent hand-patching. Hence, Bellman’s GAP was designed to free ADP from its Haskell embedding, and at the same time to incorporate new features that had evolved after publication of [9].

Outside the realm of the algebraic approach, related work is for example the Dynamite system [3] for biosequence analysis, or, in the natural language processing community, the DYNA language [5]. DYNA is based on a general, Prolog-style backtracking scheme, allowing the programmer to concentrate on the logic of the parsing algorithm, rather than its implementation. Neither of these approaches achieves a separation of search space construction and evaluation. In both cases, unfortunately, it must be said that these approaches have worked well only in the hands of their creators. Since the gain in abstractness is not large enough, they have not found more widespread use. This holds also, albeit for different reasons as described above, for our early efforts with Haskell-embedded ADP. With Bellman’s GAP, we hope to break this barrier.

2. GAP-L semantics

Signatures, tree grammars and evaluation algebras  
This paragraph recalls, in a very terse format, the basic definitions of ADP, following the literature [9]. Let \( \mathcal{A} \) be an alphabet and \( \mathcal{A}^* \) be the set of strings or sequences over \( \mathcal{A} \). A signature \( \Sigma \) over \( \mathcal{A} \) is a set of function symbols and a datatype place holder (sort) \( S \). The return type of an \( f \in \Sigma \) is \( S \), each argument is of type \( S \) or \( \mathcal{A} \). \( T_\Sigma \) denotes the term language described by the signature \( \Sigma \) and \( T_\Sigma(V) \) is the term language with variables from the set \( V \). A \( \Sigma \)-algebra or interpretation \( \mathcal{E} \) is a mathematical structure given by a carrier set \( S_\Sigma \) for \( S \) and functions operating on this set for each \( f \in \Sigma \), consistent with their specified type. Interpreting a term \( t \in T_\Sigma \) by \( \mathcal{E} \) is denoted \( \mathcal{E}(t) \) and yields a value in \( S_\Sigma \).

A regular tree grammar \( \mathcal{G} \) over a signature \( \Sigma \) is defined as a tuple \((V, \mathcal{A}, Z, P)\), where \( V \) is the set of non-terminals, \( \mathcal{A} \) is an alphabet, \( Z \in V \) is the axiom, and \( P \) is the set of productions. Each production is of form

\[
v \to t \text{ with } v \in V, t \in T_\Sigma(V)
\]

The language generated by a regular tree grammar \( \mathcal{G} \) is the set of trees

\[
\mathcal{L}(\mathcal{G}) = \{ t \in T_\Sigma | Z \Rightarrow^* t \}
\]

where \( \Rightarrow^* \) is the reflexive transitive closure of \( \Rightarrow \). By construction, \( \mathcal{L}(\mathcal{G}) \subseteq T_\Sigma \). Its elements are seen as trees when it comes to constructing them, and as formulas when it comes to their evaluation.

Symbols from \( \mathcal{A} \) reside on the leaves of these trees. The symbol \( y \) denotes the yield function and is of type \( T_\Sigma \to \mathcal{A}^* \). It is defined as \( y(a) = a \), where \( a \in \mathcal{A} \) and \( y(f(x_1, \ldots, x_n)) = y(x_1) \ldots y(x_n) \), for \( f \in \Sigma \) and \( n \geq 0 \). The yield language \( \mathcal{L}_y(\mathcal{G}) \) of a tree grammar \( \mathcal{G} \) is defined as

\[
\mathcal{L}_y(\mathcal{G}) = \{ y(t) | t \in \mathcal{L}(\mathcal{G}) \}
\]
Regular tree grammars as we use here pose a special type of parsing problem: given \( x \in A^* \), we construct the search space \( \{ t \in \mathcal{L}(G), y(t) = x \} \). This process - computing the inverse of \( y - \) is called yield parsing. A Bellman’s GAP programmer does not need to care about how yield parsing works.

The candidate trees constitute the “invisible” data structure mentioned in our introductory remarks. They allow us to effectively separate search space construction from search space evaluation. Note that the candidate trees, the elements of \( y^{-1}(x) \), are not parse trees. Each candidate tree has a parse tree by \( G \), but it is a terminal tree in \( \mathcal{L}(G) \). Basing candidate evaluation on parse trees directly might be feasible, but would create interdependence between search space construction and evaluation.

An evaluation algebra is a \( \Sigma \)-algebra augmented with an objective function \( h : [S] \rightarrow [S] \), where the square brackets denote multisets (in theory, and lists in practice).

An ADP problem instance is specified by a regular tree grammar \( G \), evaluation algebra \( E \) and input sequence \( x \in A^* \). Its solution is defined by

\[
G(E,x) = hE[E(t)|t \in \mathcal{L}(G), y(t) = x] \tag{8}
\]

The square brackets in Equation 8 denote a multiset. This is required because in practice, we often ask for all co-optimal solutions, or all solutions within a percentage of optimality. The notation \( G(E,x) \) suggests the use of the regular tree grammar \( G \) (more precisely, its yield parser) as a function called with evaluation algebra \( E \) and input \( x \) as parameters. Internally, however, Equation 8 is not executed literally. Rather, the application of the objective function is amalgamated with the evaluation of the candidate trees, which are not constructed explicitly. (In functional language terminology, this is a case of deforestation.) Also, tabulation of intermediates is used where appropriate, to avoid exponential blow-up. The prerequisite for correct and efficient computation of the solution in this way is Bellman’s Principle of Optimality [2], which is defined by Equations 9 and 10 in the ADP framework

\[
hE[ fE(x_1, \ldots, x_k) | x_1 \leftarrow X_1, \ldots, x_k \leftarrow X_k ] = \\
hE[ fE(x_1, \ldots, x_k) | x_1 \leftarrow hE(X_1) , \ldots, x_k \leftarrow hE(X_k) ] \tag{9}
\]

\[
hE( X_1 \cup X_2 ) = hE( hE(X_1) \cup hE(X_2) ) \text{ and } hE[] = [], \tag{10}
\]

where \( X_i \) denote multisets. Note that these equations imply less general criteria which are found in the literature. When \( h \) is minimization or maximization, it implies (strict) monotonicity of each \( f \in \Sigma [15] \). When the evaluation algebra holds only a single, binary and commutative function \( f \), \((h,f)\) forms a semiring where \( h \) distributes over \( f \) [17].

When compiling an ADP algorithm coded in GAP-L, it is always assumed that the evaluation algebra used satisfies Bellman’s Principle. This is the developer’s responsibility. However, in some cases, the compiler can notice that this principle is violated, and issues a warning.

**Example** We present an ADP version of the classic optimal binary search tree algorithm [4] to demonstrate the ADP concepts from the previous paragraph. Given a set of keys and their access probabilities, the algorithm computes the binary tree with minimal mean access time. Why is this a sequence analysis problem at all? Because in a search tree, the order of leaves is fixed. The yield string of any search tree must hold the keys in sorted order.

With a dynamic programming approach, the minimal expected access time results from the optimization phase, the underlying optimal tree structure is derived via backtracing. With the example input sequence

\[
s = [(1.05), (2.05), (3.040), (4.010), (5.020), (6.001), (7.019)]
\]

the two binary trees \( t_1 \) and \( t_2 \) have a mean access time of 2.18 and 2.98, where the access time is the number of key comparisons in a lookup operation, which corresponds to the depth of the accessed node.

\[
\begin{array}{c}
t_1 : \\
3 \quad t_2 : \\
7 \quad 7 \\
2 \quad 2 \quad 4 \quad 4 \\
1 \quad 1 \quad 5 \quad 5 \\
\end{array}
\]

In this example, the candidates of the search space are all possible binary search trees. The signature requires a branching symbol \( br \), a leaf symbol \( lf \) and a symbol that represents an empty subtree \( nil \).

\[
\begin{array}{c}
br : T \times A \times T \\
lf : A \\
nil : \\
\end{array} \rightarrow T
\]

Using these symbols, the following candidate trees represent \( t_1 \) and \( t_2 \):

\[
t'_1 : \\
\quad br \\
\quad (\frac{3}{0.4}) \\
\quad br \\
\quad (\frac{2}{0.05}) \quad nil \\
\quad (\frac{0.05}) \\
\quad nil \\
\quad (\frac{0.1}) \\
\quad br \\
\quad (\frac{1}{0.2}) \quad lf \\
\quad (\frac{6}{0.01}) \\
\quad nil \\
\quad (\frac{0.2}) \quad lf \\
\quad (\frac{0.01})
\]

\[
t'_2 : \\
\quad br \\
\quad (\frac{7}{0.19}) \\
\quad nil \\
\quad (\frac{0.19}) \\
\quad br \\
\quad (\frac{4}{0.1}) \\
\quad br \\
\quad (\frac{2}{0.05}) \quad if \quad nil \\
\quad (\frac{5}{0.2}) \quad if \quad nil \\
\quad (\frac{6}{0.01}) \\
\]

The following tree grammar \( btrees \) (axiom is \( btree \)) generates all possible binary search trees using the function symbols from the
signature:

\[
\text{btree} \rightarrow \quad \text{br} \mid \text{if} \mid \text{nil}
\]

where \( a \in A \). The alphabet \( A \) is \( K \times P \), where \( K \) is the set of keys and \( P = [0, 1] \).

Computing the mean access time is done via the evaluation algebra \( \text{mean} \). The place-holder (sort) \( T \) is mapped to \( \mathbb{R} \times \mathbb{R} \), where the first component of a result is the mean access time of the candidate and the second component is sum of the probabilities of the yield of the candidate tree. The evaluation functions are implemented by

\[
\begin{align*}
\text{br}(l, (k, p), r) &= (l_1 + r_1 + l_2 + r_2 + p, l_2 + r_2 + p) \\
\text{lf}((k, p)) &= (p, p) \\
\text{nil} &= (0, 0) \\
\text{h} &= \min
\end{align*}
\]

The objective function \( h \) is minimization, to choose the binary tree with the minimal access time. Thus, a call to \( \text{btrees} (\text{mean}, s) \) computes the minimal mean access time for the example sequence \( s \):

\[
\text{btrees} (\text{mean}, s) = [(1.96, 1)]
\]

i.e. the minimal mean access time for the input sequence \( s \) is 1.96.

The non-optimizing \( \text{printing} \) algebra, by convention named \( \text{print} \), transforms a candidate tree into a textual notation, i.e. \( T \) is mapped to the domain of character sequences.

\[
\begin{align*}
\text{br}(l, (k, p), r) &= "(c" + l + "y" + \text{str}(k) + "c" + r + "y") \\
\text{lf}((k, p)) &= \text{str}(k) \\
\text{nil} &= "\" \\
\text{h} &= \text{id}
\end{align*}
\]

where \( \text{str} \) converts its argument to a string representation and \( + \) means string concatenation. Note that we choose to represent only the key, not its probability.

Evaluating \( \text{btrees} (\text{print}, s) \) enumerates the complete search space of 2,128 candidates, including strings for both example candidates:

\[
\begin{align*}
t_1 &\rightarrow (((1)2)(3)((4)(5)(6))(7)(l)) \\
t_2 &\rightarrow (((1)2)(3)4((5)(6))7(1))
\end{align*}
\]

With a slightly more elaborate \( \text{print} \) algebra, we generate the \( \LaTeX \) code for tree graphics as depicted above.

Computing both, minimal mean access time and the structure of the candidate which achieves it, is done via a product algebra: \( \text{btrees} (\text{mean} \times \text{print}, s) \) returns a list of optimal structures with minimal mean access time, e.g.

\[
[(1.96, 1), "((1(2))(3)((4)(5)(6))(7)(1)))"]
\]

This example also demonstrates how the “separation of concerns” mentioned above is achieved in practice: The signature defines a universe of candidates, algebras define different ways to score them, objective functions capture the intended goal of the analysis, and tree grammars define the candidate space arising from given input. Efficiency concerns related to tabulation have been eliminated completely, as this issue is automated by the compiler. An immediate benefit of this separation of concerns is that we can build more sophisticated analyses from simpler ones by operations on evaluation algebras.

Products of algebras

If combinatorial optimization is to be performed under multiple objectives, one could design algebras which operate on tuples of scores and apply the multiple objectives. A much faster and safer way to achieve this is often possible via product algebras. They allow to describe different objectives independently, and to afterwards combine them in various fashions, avoiding a lot of redundant and error-prone coding. With algebras \( A \) and \( B \) already available, one simply calls \( \mathcal{G}(A \times B, x) \), with no extra programming or debugging effort.

We present three kinds of products here. The “lexicographic” product has been introduced and studied in [22], the other two are new.

We need to introduce two properties of algebras, which are prerequisites for well-defined products. An algebra is called unitary, if its evaluation functions return lists of results holding at most one element. A generic algebra \( A(k) \) has a parameter \( k > 0 \) such that it returns the \( k \) best solutions under its objective function. Naturally, \( A(1) \) is unitary.

Let \( A \) and \( B \) be unitary evaluation algebras over \( \Sigma \). The cartesian product \( A \times B \) is an evaluation algebra over \( \Sigma \). Its functions take arguments of the form \((a, b) \in S_A \times S_B \) or \( z \in A \). For simplicity, we define the functions of \( A \times B \) for the case of a binary function \( f \) with one argument of either kind, and leave the general case to the reader.

\[
f_{A \times B}((a, b), z) = (f_A(a, z), f_B(b, z))
\]

The objective function of \( A \times B \) is

\[
h_{A \times B}[(a_1, b_1), \ldots, (a_m, b_m)] = [(l, r) | l \leftarrow h_A[a_1, \ldots, a_m], \ r \leftarrow h_B[b_1, \ldots, b_m]]
\]

The algebras are required to be unitary, because otherwise, Bellman’s Principle would be violated and the answer sets would blow up exponentially, without adding information. The cartesian product has little use by itself, because the results returned from \( A \) and \( B \) are independent – according to Eq. 12 they are drawn from different candidates in the search space. Thus, instead of \( \mathcal{G}(A \times B, x) \) one might call \( \mathcal{G}(A, x) \) and \( \mathcal{G}(B, x) \) separately, which gives the same result, albeit more slowly. However, in combinations with other products, we have found the cartesian product useful, and we will see examples of this later.

The lexicographic product \( A \times B \) of an evaluation algebra over \( \Sigma \) and has the functions

\[
f_{A \times B} = f_{A \times B}
\]

for each \( f \in \Sigma \), and the objective function

\[
h_{A \times B}[(a_1, b_1), \ldots, (a_m, b_m)] = [(l, r) | l \leftarrow \text{set}(h_A[a_1, \ldots, a_m]), \ r \leftarrow h_B[r'], (l', r') \leftarrow [(a_1, b_1), \ldots, (a_m, b_m)], l' = l \]

Here, \( \text{set}(X) \) reduces the multiset \( X \) to a set. This product gets its name from the fact that if both algebras optimize, then it implements the lexicographic ordering of the two independent criteria as its objective. However, this product is not restricted to the case of two optimizing algebras, and exhibits a surprising versatility of use [22]. For example, a product \( A \times P \), where \( P \) is an algebra that produces an external “print” representation of the candidate, specifies a backtracing phase after optimization with \( A \).

Our third product combines the objective functions in a more sophisticated way. Assume we have the generic \( \text{MinPrice}(k) \) algebra which returns the \( k \) cheapest pizzas in our search space, and algebra \( \text{Kind} \) which evaluates pizzas as “vegetarian”, “meat”, “seafood”, or “other”, but does not make any choices. Then, the
Our first signature is and then proceed (in any order) to define algebras and grammars.Candidate terms in the search space, i.e. we define the signature, implements beginners, and powerful to use for ADP experts. To lower the entry issues will be discussed in our description of GAP-L.

Given string \( x \), we seek a palindromic structure for it, which is optimal under some scoring scheme. This is a toy example, which we will gradually extend to show the building-block style of program development in GAP-L. Exact palindromes are strings which read the same forward and backward, i.e. \( x = x^{-1} \).

We use several variants of the basic problem, such as approximate palindromes (allowing errors), nested palindromes, or finding the sub-string with the best palindrome score under a given scoring model.

We first decide which function symbols we need to model the candidate terms in the search space, i.e. we define the signature, and then proceed (in any order) to define algebras and grammars. Our first signature is

\[
\text{signature } \text{palis}(\text{alphabet}, \text{answer}) \{ \\
\text{answer match(alphabet, answer, alphabet)}; \\
\text{answer nil(void)}; \\
\text{answer turn(alphabet)}; \\
\text{choice [answer] h([answer])}; \\
\}
\]

where answer denotes the sort symbol, and function symbols are \( \{\text{match, nil, turn}\} \). They cover the cases of two matching characters, and the mid-point of a palindrome of even or odd length. The signature declaration resembles a Java interface declaration and the \( [] \) brackets denote a list type. The choice keyword marks the function symbol \( h \) as the objective function.

We describe the search space of all palindrome candidate terms with Grammar Pali1, axiom is \( S \)

\[
\begin{align*}
S & \rightarrow \text{match} | \text{turn} | \text{nil} \\
& \rightarrow \text{max}[\text{string}(\text{match}, S, \text{CHAR}) | \text{turn}(\text{CHAR}) | \text{nil}(\text{EMPTY}) \# h ; ]
\end{align*}
\]

where the first two rules are shorthands for \(|A|\) rules each, one for each \( a \in A \).

In GAP-L syntax the tree patterns are written like function applications:

\[
\text{grammar Pali uses palis(axiom = S) } \\
S = \text{match}_1(\text{CHAR}, S, \text{CHAR}) \text{ with equal | turn(CHAR) | nil(EMPTY) # h ; }
\]

The underlying signature and the axiom are specified in the header of the grammar definition. The \# operator specifies where the objective function should be applied to the rules’ alternatives (\# binds more weakly than \( | \) ). While it could be applied by default with every production, there are situations which require exceptions from this default, and we decided to give the programmer full control by an explicit \# operator. Terminal symbol/parser names are written in upper case, such as CHAR for a single character from the alphabet, and STRING for a string.

The first alternative of non-terminal \( S \) uses syntactic filtering, i.e. the left hand side of the with keyword is only parsed if the equal filter returns true for the parsed sub-word. The equal filter tests if the first and last character of the sub-word are equal. This filtering reduces the search space, rather than penalizing non-palindromes with a large cost. As a consequence, the match algebra function does not need to check the characters. Without syntactic filtering, we would need to duplicate the match rule for every character of the alphabet.

Next, we define a simple scoring scheme:

\[
\text{algebra score } \\
\text{implements palis(alphabet = char, answer = int) } \\
\text{int match(char a, int b, char c)} \\
\text{[ return b + 3; ]} \\
\text{int turn(char l)} \text{[ return 0; ]} \\
\text{choice [int]} \text{h([int] x)} \\
\text{[ return list(maximum(x)); ]}
\]

The syntax of the algebra definition resembles the definition of a Java class. The header contains the mapping between alphabet and sort symbol to concrete data types.

Perfect palindromes do not pose an optimization problem. The search space is empty if \( x \) is not a palindrome, and otherwise, it is \([l]\) with \( score(t) = |x| \cdot \text{div} 2 \cdot 3 \). No dynamic programming is needed – a simple loop is sufficient that checks for character matches in \( x \) outside to inside. In fact, the optimizations within GAP-C recognize such a special case and generate a simple loop without allocating any tables.

To make the example more interesting, we want to search the input for local palindromes. They may hold a (non-palindromic)}
turn of any length, and may be embedded somewhere in a longer string. We extend the signature for this:

signature paliS(alphabet, answer) {
    answer match(alphabet, answer, alphabet);
    answer turn(int);
    answer sl(char, answer);
    answer sr(answer, char);
    choice [ answer ] h([ answer ]); }

sl and sr allow for leading or trailing characters around the palindrome and turn allows for a region of several unmatched characters.

We extend the grammar towards local palindromes:
grammare Pali2, axiom is skipl

skipl → sl | skipr skipr → sr | S

char skipl | skipr S

S → match | turn

char S char string

Here string is a terminal symbol denoting an arbitrary string over \( \mathcal{A} \), and char denotes a character.

We write the Pali2 grammar in GAP-L as:

grammar Pali2 uses paliS(axiom = skipl) {
    skipl = skipr |
        sl(CHAR, skipl) # h ;
    skipr = skipr(sr, CHAR) |
        S # h ;
    S = match CHAR, S, CHAR) with equal |
        turn SEQ0) # h :}

The SEQ0 terminal parser accepts an arbitrary sub-word of the input and returns its length.

In our simple scoring scheme, skipping characters in the beginning or end or in the middle turn is for free:

algebra localscore
    implements paliS(alphabet = char,
        answer = int) {
    int match(char a, int b, char c) {
        return b + 3 ;
    } int turn(int l) { return 0 ;
    int sl(char c, int x) { return x ;
    int sr(int x, char c) { return x ;
    choice [ int ] h([ int ] x)
    [ return list(maximum(x)); ] }

Our scoring ignores the length of the "turn", but since SEQ0 passes the length of the subword to the function turn, we could have penalized long turns with a score of, say, −0.2 · l.

With grammar Pali2, the palindromic structure of a string is no longer unique. If we depict alternative structures using parenthesis for matched characters, 'a' for characters constituting the turn, and 'x' for leading/trailing characters, three candidate solutions (out of many, resulting from applying Pali2 to the source text of this article) can be depicted as

partofatwotrack
p1: -<((((+++))))--

wherethereader
p2: --(((+++)))----

grammar
p3: -<((())

As elements of \( \mathcal{L}(Pali2) \), they are represented as trees

p1 : sl sl p3 : sl
    \| \| \| \|
p   sr w sl g match
    \| \| \| \|
sr k h sr r match r
    \| \| \| \|
m match c sr r a match a
    \| \| \| \|
 match a sr e m turn m
    \| \| \| \|
r match r sr d
    \| \| \| \|
t match t match a
    \| \| \| \|
o turn o e match e
    \| \| \| \|
faw r match r
    \| \| \|
et turn e
    \| th

where the reader may verify (using Eq. 8) that Pali1(score, "grammar") = [], and Pali2(localscore, "grammar") = [9], where this optimal score is derived from the candidate p3.

In many situations, the size of the search space for given input is of interest. It can be determined by a counting algebra, which scores every candidate by 1 and the objective function sums over all candidates. Care must be taken to write the algebra in a way such that it satisfies Bellman’s Principle. Since there is a schematic way to achieve this, a counting algebra can be automatically derived from the signature. The syntax to request a counting algebra in GAP-L is:

algebra howMany auto count ;

The compiler generates an algebra named howMany which is equivalent to following explicit algebra definition:

algebra count
    implements paliS(alphabet = char,
        answer = int) {
    int match(char a, int b, char c) {
        return b + 3 ;
    } int turn(int l) { return 1 ;
    int sl(char c, int x) { return x ;
    int sr(int x, char c) { return x ;
    choice [ int ] h([ int ] x)
    [ return list(sum(x)); ] }

This is a particularly simple example, but the automatic generation works for any signature.

A print algebra does not optimize an objective nor does it do syntopic analysis of the search space, but it specifies how a candidate structure is mapped to a string representation. The following algebra marks gaps with - characters, matches with parentheses and an unmatched position with a + character, such as -(((+++)))-. The objective function is the identity.
algebra print
  implements paliS(alphabet = char, answer = string) {
    string match(char a, string x, char b) {
      string r; append(r, '('); append(r, x); append(r, ')');
      return r;
    }
    string turn(int l) {
      string r; r = append(r, '+', l);
      return r;
    }
    string sl(char b, string x) {
      string r; append(r, '-'); append(r, x); return r;
    }
    string sr(string x, char b) {
      string r; append(r, '-'); return r;
    }
    choice [string] h([string] x) {
      return x;
    }
  }

Bellman’s GAP requires the programmer to specify in advance which combinations of grammars and algebras will be called. This is because the compiler can perform extensive optimizations and produce specialized code when the instances are known.

As part of the GAP-L program we add two instance definitions for the products we want to use:

```
instance scorecnt = pali2( localscore * count ) ;
instance scoreprt = pali2( localscore * print ) ;
```

This example displays the versatility of the lexicographic product. According to Eq. 14, the first instance computes the number of co-optimally scoring palindromes in the search space. The second instance computes the optimal palindrome score and the print string of (all) their associated candidate structures. If you ever have programmed a backtracking phase to retrieve the candidate behind the optimal score, you will appreciate getting it without programming effort in this way.

The product operators \( \times \) and \( \otimes \) are written as \( \% \) and / characters in GAP-L.

**Multi-track input** GAP-L supports dynamic programming on multiple sequences. An example of a multi-track DP algorithm is the pairwise sequence alignment algorithm, i.e. the optimal sequence of edit operations to transform one sequence into the other.

We write a sequence alignment grammar to exemplify the GAP-L multi-track syntax. It implements our introductory edit-distance example.

First, an input declaration specifies a two track program:

```
input < raw , raw >
```

The raw keyword means that no preprocessing of the input sequence is done. Otherwise, some character transformations may be specified.

In the grammar, the \( <> \) brackets start input track bifurcations.

**Grammar**

```
grammar Align uses alignS(axiom = ali) {
  ali = match( <CHAR, CHAR> , ali) |
  ins( <EMPTY, CHAR> , ali) |
  del( <CHAR, EMPTY> , ali) |
  nil( <EMPTY, EMPTY> ) # h ;
}
```

The non-terminal `ali` is a two-track non-terminal, i.e. it is part of a two-track context. The \( <> \) parentheses enclose as many components as the number of tracks in the current context. Each component is in a one-track context. In this example, the one-track contexts only contains terminal parser calls, but it is also possible to call one-track non-terminals from multi-track bifurcations. A use case for this is the minisatellite alignment problem [1], but the model is too complex to be included here.

The \( <> \) parentheses are also used in the signature declaration and algebra definition to access the different tracks.

```
signature alignS(alphabet, answer) {
  answer match(<alphabet, alphabet>) ;
}
```

**Filters, parameters, and more** We have already seen syntactic filtering, using the \texttt{with} keyword followed by filtering function defined by the programmer. This boolean function is applied to an input sub-word before it is parsed, and no parse is made if the filter is false. This implements a syntactic pruning of the search space. Semantic filtering is also possible using the keyword \texttt{suchthat}. It tests on values derived for a candidate by the given algebra. In this case, the candidate is parsed and scored, but may be eliminated from further consideration, for any reason we specify via the semantic filter function.

Sometimes, a grammar requires many rules with renamed non-terminal symbols, but isomorphic in structure. In this case, GAP-L allows the use of nonterminal symbols that pass parameters from the left-hand to the right-hand side.

Although subscripts are completely banned in ADP programming, sometimes scoring requires access to positions in the input, for example, when a match of two characters near the ends is to be scored more highly than near the middle. To this end, GAP-L provides a parser \texttt{LOC} which recognizes an empty subword and returns its position in the input. Positions so obtained become regular arguments to the algebra functions.

Finally, for programming in the large, GAP-L provides a module concept and allows to mix several algebras, signatures and grammars in one program.

### 4. Ex-bedding Experience

As described in the introduction, ADP was first implemented as domain specific language (DSL) embedded in Haskell. With Bellman’s GAP we have chosen to ex-bed ADP from Haskell into the GAP-L language and have created the optimizing GAP-C, which translates GAP-L into efficient C++ code. [10] The reasons for the ex-bedding are threefold.

First, like with other DSLs embedded into a host language, error reporting and diagnostics are problematic in the ADP Haskell embedding. Small errors, like a missing parameter or an accidental indenting (offside rule) may lead to several screen pages of type inference errors. Effectively, to program in the embedded language one has to know how to program in Haskell and implementation details of the client language. Haskell-literacy in bioinformatics is marginal. We designed GAP-L and GAP-C with this in mind. The syntax is Java-like and algebra functions are written as imperative code. GAP-C includes some efforts for user friendly reporting of warnings and diagnostics regarding syntax and semantic errors.

Second, the performance of the Haskell embedding is limited. The runtime is several magnitudes greater in comparison with handwritten code. For several algorithms, the memory usage is
very high, such that they can only be used for short sequences [10]. GAP-C includes several optimizations such that the generated imperative code is competitive with handwritten code. Since the compiler is specialized to the domain of dynamic programming, memory management is not a problem in the generated code.

Third, new language features of GAP-L, e.g. multitrack DP and filters are problematic to implement in the embedding, because of the danger of bulky parser combinators and inter-parser interactions.

However, we still maintain the Haskell embedding, because it proves to be useful in several respects. We use it as reference implementation of the ADP core. The test suite of GAP-C includes several parallel test cases against it. Also, it is advantageous rapid prototyping of new language ideas in some cases, e.g. new product variants. Rewriting large parts of the compiler for a language extension, where it is unclear if it pays off, is not feasible.

5. Program development in Bellman’s GAP

We further extend our palindrome example to demonstrate the ease of developing dynamic programming algorithms in GAP-L. See Table 1 for an overview. We rely on the reader’s intuition to evaluate the extend to which such convenience carries over from our toy to real-world applications.

Four variations on palindromes Proceeding from Pali2 to Pali3, we merely drop the syntactic filter, and thus leave it to the scoring functions to score two equal, or similar, characters positive, the others negative:

\[
\text{localscore} = \begin{cases}
+ & \text{match (CHAR, CHAR)} \\
- & \text{match (CHAR, CHAR)}
\end{cases}
\]

Next, we change the grammar to allow for multiple successive palindromes in the input. For this, we need to add a new non-terminal to the grammar which also becomes the axiom, and we need to add another alternative rule to \( S \) to append only non-empty palindromes:

\[
\text{grammar Pali4 uses paliS(axiom = A) \{} \\
A = \text{skipl} \ |
app(\text{skipl}, S) \ # \ h ; \\
\ldots \\
S = \text{match(CHAR, S, CHAR)} \ |
\text{match(CHAR, turn(SEQ0), CHAR)} \ # \ h ; \ \}
\]

Adding one more alternative rule for \( S \), we can allow for nested palindromes, i.e. the “turn” of a palindrome can recursively contain palindromes:

\[
\text{grammar Pali5 uses paliS(axiom = A) \{} \\
A = \text{skipl} \ | \ app(\text{skipl}, S) \ # \ h ; \\
\text{skipr} = \text{sr(skipr, CHAR)} \ # \ h ; \\
S = \text{match(CHAR, S, CHAR)} \ |
\text{match(CHAR, sl(CHAR, SEQ0), CHAR)} \ # \ h ; \ \}
\]

At this point, the grammar already resembles a simplified RNA secondary structure prediction grammar. We will build this in the next section on “real world” examples.

In Table 1 we give an overview of different results.

<table>
<thead>
<tr>
<th>program</th>
<th>features</th>
<th>opt. score</th>
<th>answer</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{Pali1(score * print, x)} )</td>
<td>exact palindrome</td>
<td>-</td>
<td>( (+++++++) )</td>
</tr>
<tr>
<td>( \text{Pali2(score * print, x)} )</td>
<td>longer turn, free ends</td>
<td>6</td>
<td>( +-----((+------)) )</td>
</tr>
<tr>
<td>( \text{Pali3(score * print, x)} )</td>
<td>unmatched characters anywhere</td>
<td>7</td>
<td>( +-----((+------)) )</td>
</tr>
<tr>
<td>( \text{Pali4(score * print, x)} )</td>
<td>several adjacent palindromes</td>
<td>15</td>
<td>( (+)---(+)()((++)) )</td>
</tr>
<tr>
<td>( \text{Pali5(score * print, x)} )</td>
<td>nested palindromes</td>
<td>18</td>
<td>( ((+))((+++)) )</td>
</tr>
</tbody>
</table>

| Table 1. program variations for different palindrome problems. |

<table>
<thead>
<tr>
<th>grammar</th>
<th>Pali5(( trns * count, x ))</th>
<th>Pali5(( trns * score + print, x ))</th>
</tr>
</thead>
<tbody>
<tr>
<td>input</td>
<td>ababdcdaaadacda</td>
<td>ababdcdaaadacda</td>
</tr>
<tr>
<td>results</td>
<td>(1, 26632)</td>
<td>(1, 15)</td>
</tr>
<tr>
<td></td>
<td>(2, 60684)</td>
<td>(2, 15)</td>
</tr>
<tr>
<td></td>
<td>(3, 16896)</td>
<td>(3, 15)</td>
</tr>
<tr>
<td></td>
<td>(4, 200)</td>
<td>(4, 10)</td>
</tr>
</tbody>
</table>

| Table 2. the results of the Pali5 grammar and two classifying example products. |

splitting the search space into classes

Let us assume we want to understand our search space more deeply, and compute several near-optimal structures, which are subject to the condition that they hold a different number of local palindromes. We leave it to
the reader to design the evaluation algebra \(\text{trns}\) which counts the number of “turns” in a candidate structure. (Hint: use \(\text{app}(x, y) = x + y, \text{match}(a, x, b) = x\).) Using this algebra \(\text{trns}\) in the two products shown in Table 2, we obtain the count of structures for each number of turns, as well as the optimal score/structure for each number:

This simple example demonstrates a general method: any analysis \(G(B, x)\) can be refined with respect to a classification attribute that can be computed by yet another algebra \(A\), simply by calling \(G(A*B, x)\).

An \(O(n^6)\) time, \(O(n^3)\) space algorithm To give an example for an algorithm of higher complexity, let us seek optimal joint palindrome structures for two sequences, as exemplified by

\[
\begin{align*}
\text{MI-SISSISSIPPI} \\
+((+))() \\
\text{PAPPALAPAPP-}
\end{align*}
\]

We extend the grammar \(\text{Align}\) by a production which derives a matching character pair in either sequence, where the two pairs need not be the same. This case is scored by the new function \(\text{pmatch}\). (The extended signature \(\text{align2}\) is not shown).

**Grammar PaliJoint uses** \(\text{align2}(\text{axiom} = \text{pj})\) {\n  \(\text{pj} = \text{rep}(\text{<CHAR, CHAR>}, \text{pj})\) | \n  \(\text{ins}(\text{<EMPTY, CHAR>}, \text{pj})\) | \n  \(\text{del}(\text{<CHAR, EMPTY>}, \text{pj})\) | \n  \(\text{nil}(\text{<EMPTY, EMPTY>})\) | \n  \(\text{pmatch}(\text{<CHAR, CHAR>}, \text{pj}) . \) \n  \(<\text{CHAR, CHAR>}, \text{pj}) \# \text{h}:
}\]

**Algebra score implements** \(\text{align2}(\text{alphabet} = \text{char}, \text{answer} = \text{int})\) {\n  \(\text{int rep}(\text{<char a>, char b>, int x})\) \n  \(\text{if} (\text{a}==\text{b}) \text{return} \text{x + 1}; \) \n  \(\text{else return} \text{x - 1};\) \n}\n
\[
\begin{align*}
\text{int ins}(\text{<void, char b>, int x})\) \n  \(\text{return} \text{x - 1};\) \n\end{align*}
\]

\[
\begin{align*}
\text{int del}(\text{<char a, void>, int x})\) \n  \(\text{return} \text{x - 1};\) \n\end{align*}
\]

\[
\begin{align*}
\text{int nil}(\text{<void, void>})\) \n  \(\text{return} 0;\) \n\end{align*}
\]

\[
\begin{align*}
\text{int pmatch}(\text{<char a1, char b1>, int x,} \\
\text{<char a2, char b2>, int y})\) \n  \(\text{if} ((\text{a1} == \text{a2}) && (\text{b1} == \text{b2})) \) \n  \(\text{if} (\text{a1} == \text{b1}) \text{return} \text{x + y + 3;} \) \n  \(\text{else return} \text{x + y + 2;}\) \n  \(\text{else} \{ \text{return} -1000;\} \)
\end{align*}
\]

\[
\begin{align*}
\text{choice [int] h([int] x}) \) \n  \(\text{return} \text{list}((\text{maximum}(x)); \)
\end{align*}
\]

Here is one of the two optimal candidates for input strings “MIS-SISSISSIPPI” and “PAPPALAPAPP” with score 5.

\[c_1 : \]

\[
\begin{align*}
\text{rep} \quad \text{pmatch} \\
(\text{M}) \quad (\text{P}) \\
\text{ins} \quad \text{pmatch} \\
(\text{L}) \quad (\text{P}) \quad \text{nil} \quad (\text{\_}) \quad \text{del} \\
(\text{\_}) \quad \text{pmatch} \quad (\text{\_}) \quad \text{nil} \quad (\text{\_}) \quad \text{nil} \\
(\text{\_}) \quad \text{rep} \quad (\text{\_}) \quad \text{nil} \quad (\text{\_}) \quad \text{nil} \\
(\text{\_}) \quad \text{nil} \\
\end{align*}
\]

The new rule in Grammar PaliJoint has the abstract form \(\text{pj} \rightarrow (\text{'} \text{pj} \text{'}) \text{pj}\). Due to insertions and deletions, the position of the split between the two instances of \(\text{pj}\) is independent in the two input sequences. This leads to an algorithm which requires a four-dimensional table to tabulate non-terminal \(\text{pj}\) and has asymptotics of \(O(n^6)\) time and \(O(n^3)\) space. This concludes our expository example series, and we turn to the real world.

### 6. Practice of declarative programming with Bellman’s GAP

Although the Bellman’s GAP system has been released just recently [10], there is already a substantial amount of practical experience. Earlier applications developed with the ADP approach were transcribed into GAP-L. Firmly based on ADP theory, re-coded grammars and algebras must produce identical results in both implementations. This has been a great testbed for GAP-C. The re-coded tools now enjoy the improved efficiency achieved by GAP-C, and have smoothly replaced their earlier versions available for interactive and webservice use at http://bibiserv.cebitec.uni-bielefeld.de⁠¹.

In this section, we report how grammars, algebras and products interplay in practice to solve bioinformatics problems in the area of RNA structure analysis.

**RNA folding** RNA is a chain molecule composed from nucleotides holding the bases A (Alanine), C (Cytosine), G (Guanine), and U (Uracil). Folding back onto itself, base pairs (A-U), (C-G), and (G-U) making hydrogen bonds create secondary structure. An RNA sequence \(x\) has a large set \(F(x)\) of possible foldings, its folding space. Individual structures are encoded by string representations as in

\[
x = \text{AUGCUCCAAUUGCGUCCAACGCCUUAAUG} \\
\ldots((.((.))))....((.))....)
\]

where pairing base positions are indicated by matched parentheses, and dots denote unpaired bases. “Stacks” of successive base pairs give rigidity to the folding. This class of structures resembles our palindromes described by grammar Pali5, now taking base pairs for matching characters. A more refined grammar is required, though,

¹ The administration pages of the server collect information about the actual usage of these tools in the bioinformatics community.
because it must accommodate a sophisticated thermodynamic scoring scheme. Based on this thermodynamic model, a structure of minimal free energy (MFE) is computed via dynamic programming, and returned as the predicted structure by established bioinformatics tools such as Mfold and RNAfold [14, 24].

**RNAshapes**

RNAshapes [12] provides a grammar \( G_{\text{MicroStates}} \) (describing \( F(x) \)) as well as algebras \( A_{\text{MFE}} \) (implementing the thermodynamic energy model), and \( A_{\text{dot-backet}} \) (mapping candidates to their string representations). Hence,

\[ G_{\text{MicroStates}}(A_{\text{MFE}} * A_{\text{dot-backet}}; x) \]

performs “classical” structure prediction by free energy minimization, akin to the aforementioned tools. The unique feature of the program RNAshapes is shape abstraction: candidate structures in \( F(x) \) are mapped to equivalence classes called shapes, characterized by their arrangement of stacks and unpaired regions, irrespective of their length. The abstract shape of the above example structure would be represented as:

\[ [[[] []]] \]

Shape abstraction is implemented by an algebra \( A_{\text{shape}} \), mapping candidates to their shape. The *interleaved product* is used to compute the \( k \) best structures of different shape by a call to

\[ G_{\text{MicroStates}}(A_{\text{shape}} \odot A_{\text{MFE}}(k)) * A_{\text{dot-backet}}; x) \]

This is called *simple shape analysis*.

A second function of RNAshapes is to re-scale folding energies into Boltzmann probabilities, which are accumulated over all structures within the same shape. This calls for a simple transformation from the \( A_{\text{MFE}} \) algebra to \( A_{\text{BWE}} \), which computes Boltzmann-weighted energies. It requires a much more refined grammar \( G_{\text{MicroStates}} \) in order not to overcount candidates in the folding space. \( G_{\text{MicroStates}} \) has 26 non-terminal symbols and 67 productions. Trees on the right hand side of some productions have a height \( > 1 \) (which is allowed by Eq. 5 but has not occurred in our simpler examples). A call to

\[ G_{\text{MicroStates}}(A_{\text{shape}} \odot (A_{\text{MFE}} \times A_{\text{BWE}})) * A_{\text{dot-backet}}; x) \]

making use of a cartesian as well as two lexicographic products, computes the probabilities of all shapes which \( x \) can fold into, together with the minimal-free-energy structure within each shape.

(For backtracking the \( A_{\text{BWE}} \) values are actually disregarded by GAP-C.) This is called *complete probabilistic shape analysis*; it provides more comprehensive information than simple shape analysis, but is more expensive to compute.

**pknotsRG and pkiss**

So-called pseudoknots in RNA are structures which break the nested (palindromic) pattern of base pairing. Two common classes of pseudoknots, called “H-type” and “kissing hairpin” are exemplified here using different types of matching parenthesis:

- **H-type**
  
  \[ ...[[[...\{[...\}]]]...]] \]

- **kissing HP**
  
  \[ ...[[[...\{[...\}]]]...]\{[...\}]]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... ]... }
As defined by Equations 6 – 10, algebraic dynamic programming is restricted to sequence input, and to a recursive pattern which follows the subword structure of the input. There are other types of recursion in dynamic programming: information flow may go from the context of a subword to the inside. Recursion may be based not on the input data structure, but on the score values achieved by candidates, e.g. with knapsack-type problems. Furthermore, the input data structure may be trees rather than strings, as occurs in RNA structure comparison. Such extensions of the algebraic approach are yet to be studied.

On the practical side, we have started to recreate the Infernal tool [16] in Bellman’s GAP. This software creates structure-based covariance models for families of RNA sequences, which are collected in the Rfam data base [6] and widely used in RNA gene finding. Making use of product algebras, we plan to explore different, novel semantics [13] that can be associated with stochastic grammars.

Finally, as algebraic dynamic programming has started to enter (bio)informatics curricula at other universities, we feel obliged to enhance our suite of educational materials2, migrating our collection of educational examples to GAP-L. These examples include typical problems such as the edit distance problem on strings, in many variants, optimal matrix chain multiplication, El Mamun’s caravan, and satisfiability, aside from classical bioinformatics problems. We hope to solicit further examples from the community.

References


