BIOINFORMATICS REVIEW
Vol. 16 no. 8 2000
Pages 665–677

A systematic approach to dynamic programming in bioinformatics

Robert Giegerich*

Faculty of Technology, Bielefeld University, 33615 Bielefeld, Germany

Received on September 10, 1999; revised on February 18, 2000; accepted on April 5, 2000

Abstract

Motivation: *Dynamic programming is probably the most popular programming method in bioinformatics. Sequence comparison, gene recognition, RNA structure prediction and hundreds of other problems are solved by ever new variants of dynamic programming. Currently, the development of a successful dynamic programming algorithm is a matter of experience, talent and luck. The typical matrix recurrence relations that make up a dynamic programming algorithm are intricate to construct, and difficult to implement reliably. No general problem independent guidance is available.*

Results: *This article introduces a systematic method for constructing dynamic programming solutions to problems in biosequence analysis. By a conceptual splitting of the algorithm into a recognition and an evaluation phase, algorithm development is simplified considerably, and correct recurrences can be derived systematically. Without additional effort, the method produces an early, executable prototype expressed in a functional programming language. The method is quite generally applicable, and, while programming effort decreases, no overhead in terms of ultimate program efficiency is incurred.*

Contact: *robert@techfak.uni-bielefeld.de*

Motivation and overview

The role of dynamic programming in bioinformatics

Dynamic programming (DP) is a fundamental programming technique, applicable to great advantage where the input to a problem spawns an exponential search space in a structurally recursive fashion. If subproblems are shared and the principle of subproblem optimality holds, DP can evaluate such a search space in polynomial time. Classical application examples of DP are optimal matrix chain multiplication or the triangulation of a convex polygon (Cormen *et al.*, 1989).

For very good reason, dynamic programming is the most popular paradigm in computational molecular biology. Sequence data—nucleic acids and proteins—are determined on an industrial scale today. The desire to give a meaning to these molecular data gives rise to an ever increasing number of sequence analysis tasks, which naturally fall into the class of problems outlined above. Dynamic programming is used for assembling DNA sequence data from the fragments that are delivered by automated sequencing machines (Anson and Myers, 1997), and to determine the intron/exon structure of eucaroytic genes (Gelfand and Roytberg, 1993). It is used to infer function of proteins by homology to other proteins with known function (Needleman and Wunsch, 1970; Smith and Waterman, 1981), and it is used to predict the secondary structure of functional RNA genes or regulatory elements (Zuker, 1989). A recent textbook (Durbin *et al.*, 1998) presents a dozen variants of DP algorithms in its introductory chapter on sequence comparison. In some areas of computational biology, dynamic programming problems arise in such variety that a specific code generation system for implementing such algorithms has been developed (Birney and Durbin, 1997).

Some of the intrinsic difficulties of dynamic programming algorithms

The development of a DP algorithm is a challenging task. The core of the algorithm is given by a set of recurrences, defining the entries in one or more matrices. Neither the bioinformatics literature (Waterman, 1995; Gusfield, 1997; Durbin *et al.*, 1998), nor the computer science text books such as (Cormen *et al.*, 1989) give advice on how to systematically choose these matrices and construct the DP recurrences for a problem at hand. In all but the simplest cases, these recurrences are difficult to obtain and to validate. Even when given, they can be hard to understand, as all bioinformatics teachers may witness. Their implementation is error prone and time consuming to debug, since a subtle subscript error may not make the program crash, but instead quietly lead to a suboptimal answer every now and then.

The intrinsic difficulty of DP recurrences can be explained by the following observation: for the sake of achieving polynomial efficiency, DP algorithms perform two subtasks in an interleaved fashion: the 'first' phase is

*Work partially supported by a grant from the Australian National University.

the construction of the search space, or, in other words, the set of all possible answers. The 'second' phase evaluates these answers and makes choices according to some optimality criterion. The interleaving of search space construction and evaluation is essential to prevent combinatorial explosion. On the other hand, describing both phases separately is a great intellectual relief. It is often used in instruction. To explain sequence alignment, we may say 'first' we form all possible alignments, 'then' we score them all and choose the best, but this strategy lacks the amalgamation of both phases and suggests an algorithm of exponential runtime.

A resolution via an algebraic approach to dynamic programming

This article shows a way to reconcile the simplicity of separate phase descriptions with runtime efficiency. With a suitably chosen interface between the two phases, phase amalgamation comes for free, and the complexity introduced by it need not bother us any further. The intuitively simple concept of an algebraic data type is used to organize phase separation as well as phase amalgamation, hence our approach has been named *algebraic dynamic programming*.

Goals and structure of this article

We introduce the method of *algebraic dynamic programming* (ADP) for the systematic development of dynamic programming algorithms. The main virtue of ADP is its high degree of abstractness, which supports intuitive reasoning as well as formal validation, leading to more reliable implementations and modular (re-usable) algorithm components. A significant increase of programming productivity results from this. More complex problems can be approached with better chance of success, and there is no loss of efficiency compared to *ad hoc* approaches. With some experience in ADP, you will gain a strong intuitive judgement whether a new problem at hand is likely to have a DP solution, and what its efficiency will be.

Our introduction to ADP here will be detailed but semi-formal. It is written towards the bioinformatics community, although the method pertains to DP in general. Underpinnings in formal language theory, scope of application, detailed efficiency analysis, further classical and new applications are only touched upon. They are treated in more detail in (Giegerich, 1998; Evers *et al.*, 1999; Giegerich *et al.*, 1999; Giegerich, 1999). The discussion of related work in bioinformatics is postponed to Section **Related work**

The first part of this article introduces the method, using as a running example pairwise sequence alignment under the general and the affine cost model. We try to carefully elucidate all central aspects of the method to the novice as well as to the expert in DP.

The second part of this article sketches an application of our method—the development of a new algorithm for aligning recombinant DNA sequences in such a way that signals of recombination are respected. We shall document how this method is able to find more meaningful gap positions in pairwise sequence alignment, but the application of this algorithm to sequence data of various origins will be reported elsewhere. Instead, we will discuss software engineering issues arising in the practical use of ADP.

The method: algebraic dynamic programming

Throughout this article, x and y denote the two sequences to be aligned, with lengths m and n, respectively.

Recognition and evaluation phase

DP algorithms typically solve optimization problems, which are defined by evaluation rules over some recursive search space. The elements of this search space may be alignments, RNA secondary structures, gene structures, polygon triangulations, and so on. Because of this variety, let us introduce the generic name of a *subject under evaluation* (SUE) for such data. Their evaluation is defined by some scoring scheme, together with some rule of preference such as maximum similarity or minimum free energy.

The key observation on which our method is based is the following: DP recurrences arise as the amalgamated composition of two algorithmic subtasks (conceptually: phases) called recognition and evaluation. The recognition phase takes care of *which* SUEs to analyse, for instance 'all alignments of x and y that have no gaps in a certain region', or 'all RNA structures that do not have isolated base pairs and fold into a cloverleaf pattern'. The evaluation phase scores these SUEs and applies the rule of preference. Both phases can be described and implemented *separately* in a very transparent way.

Algebras define formula vocabularies

The algebraic view of data and grammars is essential to structuring our ideas. Fortunately, we do not need to dive deeply into the theory of data types. A (concrete) algebra is simply some set and a family of operations defined on it, such as integer numbers with integer arithmetic. An *abstract algebra* is merely a supply of symbols, used to construct formulas (terms), while a set of values is left (yet) unspecified. From the symbols $\{0, 1, +, *\}$ we build terms such as $1 + 1$ or $(0 * 1) + 0$. These may be interpreted in the Boolean algebra, or in an algebra of numbers, where they evaluate to a Boolean or a numeric value, respectively. We may also choose to interpret them in an algebra of symbols (the term algebra), where $+$ merely denotes the function that, applied to two terms x and y, creates the term $x + y$.

Formulas describe SUEs

We are used to thinking of sequence alignments as strings padded in lines of a matrix, or as paths on a grid (Waterman, 1995). We may represent RNA structures as 'squiggle' drawings, circular graphs, base pair lists or 'mountain structures' (Hofacker *et al.*, 1999). All these are useful representations. In a slightly more abstract and more uniform view, our SUEs can be represented as terms constructed over a suitable algebra. For example, an alignment of the two sequences `actac` and `agggtc` is written traditionally

```
a c - - t a c
a g g g t - c
```

Let the operators `R, I, D` denote the standard edit operations replacement (including a match), insertion and deletion. Let `E` denote the empty alignment. The alignment may now be written as the formula

```
R(a,R(c,I(R(t,D(a,R(c,E,c)),t),gg),g),a)
```

Formally speaking, `R` is a ternary operator that applies to two individual bases, and the remaining alignment. `I` and `D` are binary operators, that apply to an inserted or deleted sequence of bases and the remaining alignment. Formulas can always be depicted as trees; such a treelike representation is shown in Figure 1. Note that the formula and the tree are more specific than the traditional representation, as they clearly state that the alignment has one (two-base) insertion rather than two (single-base) insertions.

There is always a choice. We may choose to represent insertions and deletions base by base, distinguishing the gap opening (operators `I, D`) from gap extension (operators `Ix, Dx`). Our alignment is now represented by the formula

```
R(a,R(c,I(Ix(R(t,D(a,R(c,E,c)),t),g),g),g),a)
```

Both the algebras {`R, I, D, E`} and {`R, I, Ix, D, Dx, E`} will be used in the sequel. Let us call them the *general* and the *affine alignment algebra*.

Algebras evaluate SUEs

Once alignments are written as formulas, it is easy to define scoring schemes. We only need to supply a suitable algebra. Let `w(a,b)` be an integer score associated with replacing character `a` by character `b`, and let `gap(u)` be an arbitrary function that assigns a cost to a gap, i. e. a sequence `u` of inserted or deleted bases. The *general cost algebra* is defined as follows:

```
general_cost_algebra = (R(a,r,b) = w(a,b) + r,
                        I(r,u)   = gap(u) + r,
                        D(u,r)   = gap(u) + r,
                        E        = 0,
                        h        = minimum   )
```

Alternatively, let `open` and `extend` be constant costs associated with opening a gap, and extending a gap by a single base. The *affine cost algebra* is defined as follows:

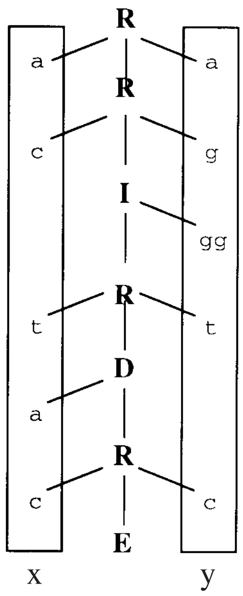

Fig. 1. An alignment in tree form, and its yield indicated in boxes `x` and `y`.

```
affine_cost_algebra = (R(a,r,b) = w(a,b) + r,
                       I (r,b)  = open   + r,
                       Ix(r,b)  = extend + r,
                       D (a,r)  = open   + r,
                       Dx(a,r)  = extend + r,
                       E        = 0,
                       h        = minimum  )
```

Algebras that evaluate SUEs are called *evaluation algebras*. Aside from the functions required to interpret SUE formulas, we always specify a choice function `h` with each algebra. It indicates our rule of preference, to be used in optimization.

An alignment is scored simply by interpreting and evaluating the formula that represents it according to the chosen evaluation algebra. Assuming that `w(x,x) = 0`, the above alignment example evaluates to

```
R(a,R(c,I(Ix(R(t,D(a,R(c,E,c)),t),g),g),g),a)  =
w(a,a)+w(c,g)+open+extend+w(t,t)+open+w(c,c)+0 =
w(c,g)+2*open+extend
```

in the affine cost algebra, while in the general cost algebra it evaluates to

```
R(a,R(c,I(R(t,D(a,R(c,E,c)),t),gg),g),a)       =
w(a,a)+w(c,g)+gap(gg)+w(t,t)+gap(a)+w(c,c)+0 =
w(c,g)+gap(gg)+gap(a)
```

This is the discipline we impose: the data type $\mathcal{SUE}$ representing the SUEs must be chosen such that the intended evaluation function can be specified as an $\mathcal{SUE}$-algebra. We have not yet encountered an application where such a choice was impossible or even difficult.

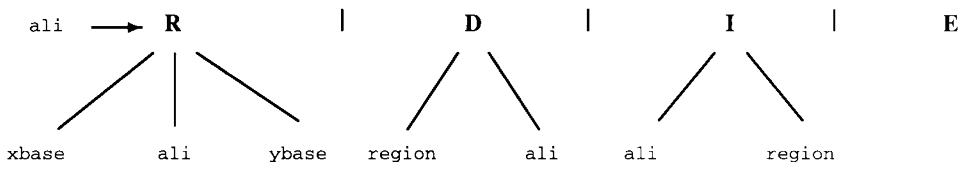

Fig. 2. The general alignment grammar in graphical form. The only nonterminal symbol is `ali`. `xbase` and `ybase` denote arbitrary bases in `x` or `y`, respectively. `region` denotes a nonempty sequence of bases.

Grammars describe specific classes of SUEs

An algebra always defines a language of formulas. In all but the most trivial cases, the SUEs we are interested in form a proper subset of this language. We may want to exclude alignments where a gap is artificially split in two; we may want to normalize alignments by the rule that if a deletion is adjacent to an insertion, the deletion always goes first. In some application context, we may forbid gaps in a particular region altogether, and so on.

We introduce *yield grammars*[†] to specify exactly which SUEs we are interested in. A yield grammar is given by a tree grammar and a yield function. A *tree grammar*, in turn, is like a context-free grammar, with terminal symbols, nonterminal symbols, a designated axiom symbol and productions where the right-hand sides are trees (formulas) taken from some underlying term algebra. Nonterminal symbols take the place of variables at the leaves of these trees. See Figure 2.

The *yield function* maps the terminal symbols at the leaves of a tree into a sequence of symbols, or in the case of alignments, into two such sequences, reproducing `x` and `y`. See Figure 1.

The *parsing problem for yield grammars* is: given two sequences `x` and `y`, find all alignments `w` such that `yield(w) = (x,y)`. Yield parsers will implement the recognition phase.

We now introduce a notation for tree grammars that is more suitable for computer input than the graphical form in Figure 2. In this notation, `|||` separates alternative rules for the same nonterminal symbol, and `<<<` denotes the application of some operator to its arguments, which are separated by `~~~`. This notation is also called the blackboard notation. Here is the general alignment grammar:

```
ga_grammar = axiom ali where
   ali =  R <<< xbase  ~~~ ali ~~~ ybase   |||
          D <<< region ~~~ ali             |||
          I <<<            ali ~~~ region  |||
          E <<< empty
```

This grammar actually does not restrict the language of alignment formulas at all. In the case of the affine alignment algebra, we shall be more specific. While formulas representing alignments can be built from `R, I, Ix, D, Dx, E` in arbitrary ways, our intention is that a gap always starts with an `I` or `D` operator, followed by zero or more `Ix` or `Dx` operators, respectively. Such well-formedness conditions are expressed via the following affine alignment grammar:

```
aa_grammar = axiom ali where
   ali   = R  <<< xbase ~~~ ali   ~~~ ybase |||
           D  <<< xbase ~~~ exDel           |||
           I  <<<           exIns ~~~ ybase |||
           E  <<< empty

   exDel = Dx <<< xbase ~~~ exDel           ||| ali
   exIns = Ix <<<           exIns ~~~ ybase ||| ali
```

Note that both grammars allow a deletion immediately adjacent to another deletion. We may consider such alignments redundant, since the adjacent gaps should be merged. It is an easy exercise to modify the grammars such that adjacent deletions or insertions are excluded. A further refinement leads to the canonization used by (Kurtz and Myers, 1997) in order to evaluate statistical significance of alignments.

A grammar in blackboard notation is the first and creative step in ADP. All further steps are systematic refinements of this first version.

Parsers implement the recognition phase

We now solve the yield parsing problem in a rather elegant way. This section turns the grammar into the corresponding recognizer, with only a minor refinement of notation. The technique we use is combinator parsing (Hutton, 1992), adapted to our yield grammars.

We need a little notation and conventions. Let `x` and `y` be two input sequences, of length `m` and `n`, respectively. The `i`-th character of `x` is denoted `x!i`. Subwords of `x` or `y` will be denoted by their bounding positions. The subword `(i,j)` of `x` holds the characters `x!(i+1)...x!(j)`. So the length of subword `(i,j)` is `j-i`, and if `x = "justice"`, then subword `(0,4) = "just"`, subword `(4,7) = "ice"`. The convenience of this convention is with splitting subwords: `(i,j) = (i,k) ++ (k,j)` for

[†] Computationally, yield grammars are quite similar to context-free grammars; however, the tree grammar by itself is a regular grammar, and yield parsing as defined here is different from tree parsing, see (Brainerd, 1969) or (Giegerich, 1990).

all `k` between `i` and `j`. The empty list is denoted `[]`, `++` denotes list concatenation and the list comprehension `[f(x) | x <- xs, pred(x)]` denotes the list of all values `f(x)` such that `x` is from the list `xs` and satisfies predicate `pred`.

Precedence for the operators in our grammar notation was chosen such that grammars can be written mostly without parentheses. `<<<` takes precedence over `~~~`, which takes precedence over `|||`. Furthermore, `~~~` associates to the left. Hence, a production such as

```
u = f <<< u ~~~ v ~~~ w ||| z
```

is implicitly parenthesized as

```
u = (((f <<< u) ~~~ v) ~~~ w) ||| z
```

Parsers operate on the two input sequences `x` and `y`. A parser is a function that can be applied to a suffix of `x` and a suffix of `y`, yielding a list of all the alignments of the two suffixes. There is a parser for each nonterminal and terminal symbol of the grammar. Parsing its input, it returns a list of all SUEs of the appropriate type that can be constructed from the input. If no such SUEs exist, a parser *fails* by returning an empty list.

Terminal parsers apply to one of the input sequences, and in their case, a position pair `(i,j)` denotes a subword of this input sequence.

These are the parsers recognizing terminal symbols:

```
xbase(i,j)  = [x!j | i+1 == j]
ybase(i,j)  = [y!j | i+1 == j]
region(i,j) = [(i,j)| i < j)]
empty(i,j)  = [() | i == m, j == n]
```

If successful, parsers `xbase` and `ybase` return a result list holding a single base, `region` returns a non-empty subword. `empty` returns a void result (denoted here by an empty tuple). All result lists are empty if the parser fails.

The operators `|||` etc. are defined as higher-order functions that compose parsers from simpler ones. This is why they are colloquially called parser combinators. Their definition is given in Figure 3. One subtlety needs extra explanation: the definition of `f <<< p` seems to assume a unary function `f`. But note that `f` may as well be a binary (ternary etc.) function, that is given only its first argument. In that case, `f(x)` denotes the function derived from `f` by fixing the first argument to `x`. So, a parser such as `(R <<< xbase)` actually returns a list of functions. The grammar ensures that eventually all functions are supplied with all their arguments. Therefore, the equivalence

```
(R <<< xbase -~~ ali ~~- ybase)(i,j) =
[R(x!(i+1),r,y!(j+1)) | r <- ali(i+1,j+1)]
```

holds for `i<m` and `j<n` under the above definitions.

Finally, the `axiom` clause is interpreted as the initial call to the parser for the axiom symbol:

```
axiom p = p(0,0)
```

With these refinements in mind, take a fresh look at grammars. The general alignment grammar now reads

```
ga_grammar = axiom ali where
   ali =  R <<< xbase  -~~ ali ~~- ybase   |||
          D <<< region +~~ ali             |||
          I <<<            ali ~~+ region  |||
          E ><< empty
```

The productions have now turned into mutually recursive definitions for the parsers for nonterminal symbols. The general alignment grammar defines a single parser `ali`, whereas the affine alignment grammar defines parsers `ali`, `exIns` and `exDel`. For each grammar and sequences `x` and `y`, the axiom clause means a call to `ali(0,0)`, which evaluates to the list of all well-formed alignments of `x` and `y`.

Parser combinators are certainly the quickest way to obtain a correct, recursive parser from a grammar. They are likely to yield the least efficient parsers the world has seen.

Dynamic programming = recursion + tabulation

As the result of the previous section, a yield grammar in our notation can be interpreted as a top-down, nondeterministic, recursive parser. Calling this parser on two sequences `x` and `y`, it will enumerate all their alignments (as specified by the grammar). Doing so, it will consume exponential time and space, for two reasons. The first reason is that it recomputes parses of the same subsequences many times. The second reason is the sheer size of the answer—there are exponentially many alignments of two sequences. This and the next section will eliminate both reasons for exponential explosion.

DP achieves polynomial efficiency over a recursively defined, exponential search space by tabulating and reusing results for shared subproblems. Mathematically, this is no big deal—a table is nothing but a function mapping a (finite) subscript domain to the values of the corresponding table entries. When `f` is a function of a pair of integers, let us denote by `f!` an `m` by `n` table such that `f!(i,j) = f(i,j)`.

This convention allows us to introduce some efficiency annotation in the grammar, turning parsers into tabulated functions:

```
ali! =   R <<< xbase    -~~ ali! ~~- ybase   |||
         D <<< region   +~~ ali!             |||
         I <<<              ali! ~~+ region  |||
         E ><< empty
```

Here it is, our first DP algorithm. As annotated, the general alignment grammar specifies a recursively defined, `m` by `n` table, where `ali!(i,j)` holds all alignments of suffixes `i,j`.

The *tabulating parser* so obtained is much more efficient than the recursive one, as all intermediate parses are

```
(p ||| q)(i,j) = p(i,j) ++ q(i,j)                                  --  alternative
(f <<< p)(i,j) = [f(u)|u <- p(i,j)]                                --  function application
(f ><< p)(i,j) = [f   |u <- p(i,j)]                                --  nullary function appl.
(p +~~ q)(i,j) = [z(v)|k <- [i+1..m], z <- p(i,k), v <- q(k,j)] --  juxtaposition
(p ~~+ q)(i,j) = [z(v)|k <- [j+1..n], z <- p(i,k), v <- q(j,k)]
(p -~~ q)(i,j) = [z(v)|i < m, z <- p(i,i+1), v <- q(i+1,j)]     --  bounded juxtaposition
(p ~~- q)(i,j) = [z(v)|j < n, z <- p(i,j+1), v <- q(j,j+1)]
```

Fig. 3. The operators of the blackboard notation are redefined as parser combinators. The parser `(p ||| q)` yields the concatenated results of parsers `p` and `q`. The parser `(f <<< p)` yields the results of parser `p` with the function `f` applied to each result. `(f ><< p)` defines the case for a nullary function `f`. The `~~` operator comes in four variants. In `(p +~~ q)`, `p` reads from input `x`, while `q` reads `x` and `y`. Conversely, in `(p ~~+ q)`, `p` reads both inputs, while `q` reads from `y`. The combinators `-~~` and `~~-` define the special case of `+~~` and `~~+` for reading a single character from `x` and `y`, respectively.

calculated only once, stored in and re-used from the table. This is what we know as DP. Each table entry holds a list of answers, i.e. alignments. If only there were not an exponential number of these alignments, the table could be computed in polynomial time.

From parsers to multi-purpose evaluators

Who wants to see an answer of exponential size, anyway? In the previous sections, we have implicitly assumed that the parsers were using the term algebra of alignments, as if they were explicitly constructing those SUEs representing alignments. By a simple abstraction, parsers turn into general purpose *evaluators*: we interpret the symbols `R, I, D, E` etc. as the functions of a suitable evaluation algebra, given to the parser as an extra parameter. If called with a scoring algebra, `ali!(0,0)` evaluates to the list of scores of all possible alignments. All we need to add is the application of the choice function already provided within each algebra. We use the operator `(...)` to associate the choice function with a production. It takes lowest priority among all grammar operators. Its formal definition is

```
    (p ... h)(i,j) = if z == [] then [] else h(z)
                     where z  = p(i,j)
```

Augmented by the choice function, the affine alignment evaluator now reads

```
af_eval alg         = axiom (ali!) where
  (R,D,Dx,I,Ix,E,h) = alg

  ali!   = R  <<< xbase  -~~ ali!   ~~- ybase |||
           D  <<< xbase  -~~ exDel!           |||
           I  <<<            exIns! ~~- ybase |||
           E  ><< empty                       ... h
  exDel! = Dx <<< base   -~~ exDel! ||| ali!  ... h
  exIns! = Ix <<< exIns! ~~- base   ||| ali!  ... h
```

This is the amalgamation of recognition and evaluation phase! Instead of constructing the recognized structures, the parser evaluates them incrementally. The choice function applied at intermediate steps controls the number of results. The function call `af_eval affine_cost_algebra x y` evaluates to the score of an optimal alignment, in time and space $O(m * n)$. Our DP algorithm is complete.

What have we achieved? Is this anything other than a long way to arrive at a simple DP algorithm? Well—note that we have more than a single algorithm. The above grammar specifies the recurrences independently of the evaluation algebra. If we change the way we score the alignments, we merely plug in alternative definitions for the functions of the algebra. For example, we may specify a unit cost algebra, where deletions, insertions and mismatches count 1. It is just as easy to introduce position-dependent scoring. We may also use the grammar to do statistics: as an exercise, the reader may devise a *counting algebra*, such that `af_eval counting_algebra x y` computes the number of possible alignments of `x` and `y`. Such statistics is extensively used in RNA secondary structure research in (Hofacker *et al.*, 1999).

Furthermore, note that our DP algorithm specified in this form is free of subscript errors, plainly because there are no subscripts. All the subscript handling is done by the combinators, and once these have been set up correctly for a given area of application, we may write more complicated grammars for more refined analyses, without worrying about subscripting at all. Some quite sophisticated grammars for restricted classes of RNA secondary structures can be found in (Giegerich, 1998).

Deriving first-order DP recurrences

A DP algorithm is completely specified by an annotated tree grammar and an evaluation algebra. This actually is a much more convenient level to reason about DP algorithms than the traditional matrix recurrences. But still, we need these recurrences explicitly, in order to implement the algorithm efficiently in an imperative programming language such as FORTRAN or C. However, we have already done all the algorithm design work; the rest is merely straightforward formula manipulation.

Gotoh first specified the recurrences for computing the optimal global alignment under the affine cost model in quadratic time (Gotoh, 1982; Waterman, 1995). Readers familiar with this work will rightfully expect that our tabulated parsers `ali!`, `exIns!` and `exDel!` correspond to the three tables D, F and E used in (Waterman, 1995). Substituting the definition of the affine cost algebra and the parser combinators, we can systematically eliminate all higher order functions in the grammar, leaving us with three tables defined via mutual recursion. Let us consider `exDel!` first. From the definitions we obtain in a first step

```
exDel!(i,j) = minimum([extend + exDel!(i+1,j)|i<m]
                     ++[ali!(i,j)])
```

All table entries are singleton lists, so we identify `[v]` and `v`. Splitting up on `i` we obtain the start and the recursive case:

```
exDel!(m,j) = ali!(m,j)
exDel!(i,j) = minimum[extend + exDel!(i+1,j),
                       ali!(i,j)] for i<m
```

Similarly for `exIns!` we obtain

```
exIns!(i,n) = ali!(i,n)
exIns!(i,j) = minimum [extend + exIns!(i,j+1),
                        ali!(i,j)] for j<n
```

Finally we substitute the definitions for `ali!`, yielding

```
ali!(i,j) = minimum
    ([w(x!(i+1),y!(j+1)) + ali!(i+1,j+1)|i<m,j<n] ++
     [open + exDel!(i+1,j) | i<m] ++
     [open + exIns!(i,j+1) | j<n] ++
     [0 | i == m, j == n])
```

Again splitting up the cases we arrive at

```
ali!(m,n) = 0
ali!(i,n) = open + exDel!(i+1,n)   for i<m
ali!(m,j) = open + exIns!(m,j+1)   for j<n
ali!(i,j) = minimum[w(x!(i+1),y!(j+1))+ali!(i+1,j+1),
                    open + exDel!(i+1,j),
                    open + exIns!(i,j+1)],otherwise
```

These recurrences are equivalent to those in (Waterman, 1995), but not identical. Since the recursion runs in the opposite direction, the gap opening contribution occurs in the recurrences for `ali!` rather than with those for `exDel!` and `exIns!` as in (Waterman, 1995).
recurrences

Method summary

Having run through the development of a classical DP algorithm with our method, let us reconsider what was genuine to the method, and what results from this particular application.

The introduction of the algebraic data type $\mathcal{SUE}$ representing SUEs, and the separate specification of grammar and evaluation algebra, is the core of our method. Different areas of application will use different algebras to describe SUEs. Tree grammars will always be used in the given notation, but the yield function and correspondingly the operational definition of the combinators may need to be adjusted. For example, RNA folding works on a single input sequence, which is reflected in the way the combinators are defined.

Once the SUE algebra is chosen, grammars and evaluation algebras may vary independently, leading to independent refinements of SUEs and of scoring functions. Each combination of grammar and algebra leads to a specific set of recurrences.

The efficiency of the specified algorithm is easily read from the grammar. The recognition phase always requires $O(n^{t+2})$ steps, where t is the maximal number of `+~~` and `~~+` operators in the right-hand side of a production[‡]. The overall efficiency critically depends on the properties of the choice function.

An application: development of an algorithm for aligning recombinant DNA sequences

In this section we report on an application[§] of the ADP approach. We only give a short biological justification, and mainly concentrate on the software engineering aspects of using ADP. The novelties in this section (aside from the algorithm itself) are the use of an executable specification, and some slight extensions of the method.

The problem: searching for the signals of recombination

DNA recombination is an important mechanism in molecular evolution. Genes that have evolved independently in different strains of a virus, for example, may recombine in a new strain. This adds the power of parallel processing to Darwinian evolution, which is otherwise based on error and trial.

In the presence of recombinant DNA, most commonly used analysis programs go wrong. There is no longer a tree-like phylogeny, as different parts of a sequence stem from different ancestors. In such a case, the best we can hope for from a tree reconstruction program is to tell us that there is support for several contradictory trees in the sequence data.

However, there is a difference between data that are just noisy, and data that carry a clear signal about recombination events in their evolutionary history. There are different conceivable ways to explicitly search for recombination signals. The PhylPro program (Weiller, 1998) does so by monitoring patterns of changes in the mutual similarities in a multiple sequence alignment. In

[‡] In fact, recognition time can always be reduced to $O(n^3)$ by a grammar transformation shown in (Giegerich, 1998).

[§] The application of a programming method is the development of a new algorithm. The application of this algorithm to study sequence data will be reported elsewhere.

this section, we take a direct approach, making pairwise sequence alignment sensitive to signals of recombination.

Many molecular mechanisms of recombination leave traces in the form of target site duplications of varying length. Our goal is to generalize the edit-distance model of sequence comparison such that

- insertions and deletions are recognized as recombination signals in the presence of target site duplications and
- arbitrary and independent score functions may be assigned with recombination sites and ordinary gaps.

Extending the edit-distance model

So far we have worked with the classical string edit model, using replacements, insertions and deletions. As new edit operations, we introduce recombinant deletions (L) and insertions (S). Let t be a (typically short) sequence of nucleotides that occurs in x and y. $L(tut, t-)$ denotes a *recombinant deletion* from x: following the target site t, present in both x and y, a sequence u of nucleotides is deleted from x. This requires a second copy of t to follow u in x. In y, a gap of the combined length of u and t is introduced.

$S(t-, tut)$ denotes a *recombinant insertion* in y: following the target site t, present in both x and y, a sequence u of nucleotides is inserted into y, followed by a new copy of t in y. In x, a gap of the combined length of u and t is introduced.

A recombinant alignment example

The improvements to be expected from a recombinant alignment algorithm over pairwise alignment insensitive to recombination signals are demonstrated in Figure 4. We applied the algorithm developed below to chicken immunoglobin sequences extracted from a multiple alignment, and de-gapped. These are two typical findings:

- In the left part of the recombinant alignment, a gap of length 12 (present in the multiple alignment) is re-discovered in the correct position. Additionally, it is marked as a direct repeat, as it may result from a recombinant insertion with an empty insert. Further experiments reveal that a pairwise alignment insensitive to recombination, but with the same scoring otherwise, has an insertion in approximately the same position, but does not correctly exhibit the repeat due to the local ambiguity of the alignment.
- The right part of the original alignment is poor with 8 mismatches (marked by the symbol *) within a region of 23 bases (between the delimiters > and <). The recombinant alignment offers an alternative explanation. It exhibits both a recombinant deletion and an insertion, with significant target sites, reducing the mismatch count to 1 over the same region.

Rapid prototyping via a functional programming language

The ADP method relies on algebraic data types and higher order functions. Both concepts are provided by modern functional programming languages, based on Church's λ-calculus Barendregt (1984). In particular, the comfortable syntax of the functional language *Haskell* Bird (1998) allows to embed our algorithm development in *Haskell*, thus providing executable specifications at early stages of the development. The price to be paid amounts to some minor changes in notation: the most obvious change is that we write function application in the form `f x y` or `(f x y)` if necessary, rather than `f(x,y)`. The expression `f(x,y)` is now seen as a call to a function with a single argument, which is a value pair. Prefix function symbols always take precedence over infix operators like `|||`.

This section imports the basic definitions from the previous ones; these are not repeated here. Otherwise, the lines marked > constitute a complete and executable program[¶]. Given today's compiler technology, lazy functional languages come with a certain overhead in computation time and (more dramatically) memory space. A functional language implementation of the algorithm will often not achieve the efficiency that is eventually required in practice. But still, having an executable version around is an invaluable advantage during program development; we will return to this in Section **Experiences in program development**.

Step 1: The recombinant alignment algebra

Our first step is to design the algebra representing recombinant alignments. The general alignment algebra introduced earlier is now extended by new operators `L` and `S` to represent recombinant deletions and insertions, respectively. We call it the recombinant alignment algebra.

```
>  data Ali= R Base Ali Base                    |
>            D Region Ali                       |
>            I Ali Region                       |
>            L Region Region Region Ali Region  |
>            S Region Ali Region Region Region  |
>            E
```

Step 2: Evaluation algebras

The scoring algebra. Our approach allows a general cost function; however, we will use affine costs, mimicking the defaults of ClustalW (Thompson *et al.*, 1994).

[¶] In fact, the LaTeX source of this section is an executable *Haskell* script.

```
                                         >*    * ** * **     *   <
...tac------------tatggctggtaccag...ctccggttccctatccggctccacaggcacat...
...tactatggctggtactatggctggtaccag...ctccggctccccaggcagaaccacaagcacat...

                                         >                *                  <
...tactatggctggtac------------cag...ctccggttccctatccggctccacaggca------------cat...
...tactatggctggtactatggctggtaccag...ctccgg------------ctccccaggcagaaccacaagcacat...
...RRRSSSSSSSSSSSSTTTTTTTTTTTTRRR...RLLLLLUUUUUUUTTTTTRRRRRRRRSSSUUUUUUUUUTTTRRR...
```

Fig. 4. Original alignment (top) and recombinant alignment (bottom).

```
>  match a b = if a == b then 0 else case (a,b) of
>     ('a','g') -> 1
>     ('a','c') -> 3
>     ('a','t') -> 3
>     ('g','a') -> 1
>     ('g','c') -> 3 -- and so on

>  clust_alg = (fE, fR, fD, fI, fL, fS, h) where
>   fE         = 0
>   fR a x b   = x + match a b
>   fD (i,j) x = x + open + fromInt(j-i)*extend
>   fI x (i,j) = x + open + fromInt(j-i)*extend
>   fL (i,j) (u,u') _ x _ =
>       x + ropen (i,j) + fromInt(u'-u)*rextend
>   fS (i,j) x _ (u,u') _ =
>       x + ropen (i,j) + fromInt(u'-u)*rextend
>   h x        = [minimum x]

>   open = 5.0
>   extend = 0.2
>   ropen (i,j) = open/fromInt(j-i)
>   rextend = extend
```

Scores are floatingpoint values (and `fromInt` denotes type conversion). This cost algebra gives a strong preference to recombinant insertions and deletions with a long target site duplication by dividing the gap opening cost by the length of the target site. Also, note that the duplication of the target site is not scored at all, i.e. it does not contribute to the length-dependent score of the insert.

Step 3: A grammar for well-formed recombinant alignments

The data type `Ali` is not specific enough to describe exactly all meaningful alignments. For example, it allows to represent two subsequent insertions, which should rather be merged into a single, longer insertion.

We therefore introduce a grammar generating exactly the well-formed alignments. The blackboard version is shown in Figure 5.

The three occurrences of `region` in the productions associated with `S` and `L` must all derive the same nucleotide sequence. Furthermore, target sites must extend maximally to the right. These two properties are checked by the predicates `tsdup` and `maximal`.

Step 4: Implementing recognition

For implementing the recognizer, the operators in the grammar need to be refined as explained in Section **Parsers implement the recognition phase**. Besides this, we need a parser for `uregion`, and a definition of `suchthat`.

```
>  uregion (i,j)  = [(i,j) | i <= j]
>  suchthat q f (i,j) = [s | s <- q (i,j), f s]
```

The code for the predicates `tsdup` and `maximal` is straightforward and not shown.

Step 5: Tabulation and abstraction

For indicating tabulation, there is a change in notation, since *Haskell* does not allow the convenience of using the array indexing symbol `!` when specifying a tabulated function. We therefore define a function `tabulated` that tabulates a function, such that `p = tabulated q` can be written instead of the earlier `p! = q`. Tabulation is introduced for the nonterminal symbols `ali`, `noDel`, `noIns`, `recombIns`, and `recomDel`. Tabulation is not required for the nonterminal `match`, as this parser only performs a constant amount of work on each call.

Finally, we make the evaluation algebra a parameter of the grammar, obtaining the first version of a DP algorithm that calculates recombinant alignments under arbitrary evaluation algebras. See Figure 6.

Step 5a: Optimization to yield an $\boldsymbol{O(n^3)}$ *implementation*

The evaluator so obtained runs in $O(n^2)$ space and in $O(n^6)$ time, due to the four occurrences of `+~~` and `~~+` in the productions associated with recombinant insertions and deletions. This problem requires some algorithmic imagination, not covered by the ADP method. The result, however, is easily expressed via a transformed grammar, such that only one `+~~` operator remains in

```
recomb_grammar  = axiom ali where
   ali   = match  |||  D <<< region ~~~ noDel  |||  I <<< noIns  ~~~ region
   noDel = match  |||  I <<< match  ~~~ region
   noIns = match  |||  D <<< region ~~~ match
   match = E <<< empty
     |||   R <<< xbase  ~~~ ali     ~~~ ybase
     ||| ((S <<< region ~~~ noIns   ~~~ region ~~~ uregion ~~~ region)
                  `suchthat` tsdup) `suchthat` maximal
     ||| ((L <<< region ~~~ uregion ~~~ region ~~~ noDel   ~~~ region)
                  `suchthat` tsdup) `suchthat` maximal
```

Fig. 5. The grammar for recombinant alignments in blackboard notation. The new terminal symbol `uregion` represents a possibly empty region (the insert). A new `suchthat` clause is used to associate syntactic restrictions with regions representing target sites and their duplications.

```
>  dp_alignments alg = axiom (p ali) where
>    (fE, fR, fD, fI, fL, fS, h) = alg
>    ali   = tabulated( match ||| fD <<< region +~~ (noDel!)  |||
>            fI <<< (noIns!)  ~~+ region ... h)
>    noDel = tabulated( match ||| fI <<< match  ~~+ region ... h)
>    noIns = tabulated( match ||| fD <<< region +~~ match  ... h)
>    match =  fE ><< empty    ||| fR <<< xbase -~~ (ali!) ~~- ybase |||
                 (recombIns!) ||| (recombDel!) ... h
>    recombIns  = tabulated(
>                 ((fS <<< region +~~ (noIns!) ~~+ region ~~+ uregion ~~+ region)
>                  `suchthat` tsdup) `suchthat` maximal ... h)
>    recombDel  = tabulated(
>                 ((fL <<< region +~~ uregion +~~ region +~~ (noDel!) ~~+ region)
>                  `suchthat` tsdup) `suchthat` maximal ... h)
```

Fig. 6. The executable first prototype of a dynamic programming algorithm for recombinant sequence alignment.

each production. Finally, repeat precomputation in $O(n^2)$ is used instead of checking afterwards for target site duplication and maximality. For lack of space, these refinements can not be shown here; for details see (Giegerich *et al.*, 1999). As a result, the evaluator runs in $O(n^3)$ time and $O(n^2)$ space, using arbitrary scoring functions.

Step 6: Deriving DP recurrences

Dynamic programming recurrences were derived from the improved abstract evaluator combined with the scoring algebra, as explained in Section **Deriving first-order DP recurrences**, and implemented in C.

Experiences in program development

Having worked with ADP before, and with the suitable combinators available, it was a matter of two hours to define the recombinant alignment data type, the grammar and the evaluation algebras, to arrive at (almost) the evaluator in Step 5. Although extremely slow, it could be run on small artificial examples, and was used to discuss the modelling with the cooperating biologist. This led to the observation that it was sufficient to consider *maximal* target site duplications, and this restriction was readily added. Coming up with the improved parsers required some new algorithmic ideas, not covered by the ADP method. The resulting *Haskell* program, as obtained after Step 5a, was able to analyse some small sets of real data, further fostering our confidence in the validity of the approach. The derivation of DP recurrences was done independently by a senior researcher and a graduate student, who cross-checked their results. Their implementation in C by the student required three days of work, including debugging. The functional program helped to spot errors in the C program that might otherwise have gone unnoticed. First measurements show that the C-program runs faster than the compiled *Haskell* program by a factor of 68, while using 2% of the space. Still—we feel that this route of development has saved several weeks of programming, testing and re-programming. Of course, this is subjective evaluation, but note that the maximality restriction was found only after running the prototype, and we see no way to achieve $O(n^3)$ without it. A direct approach to an efficient DP algorithm might have been a futile effort.

Discussion

Related work

The advantages of a *declarative approach to biosequence analysis* have been explicated most influentially by Searls (Searls, 1988, 1989; Searls and Murphy, 1995). In his recent review (Searls, 1997), Searls discusses different approaches to the gene prediction problem in the presence of introns. Although gene structure can be described by a context-free grammar and hence can be recovered by parsing methods, Searls feels that DP methods such as used in (Gelfand and Roytberg, 1993; Snyder and Stormo, 1993) may have an inherent efficiency advantage:

> ‘Although parsing of context-free languages can be performed with similar efficiency, the problem of examining all parses, where there may be an exponential number of them, may be intractable even so’ (Searls, 1997).

This fear of exponential explosion is appropriate as long as we keep the parsing and the examination phase separate, but ADP amalgamates the phases, such that the grammar based and the DP approach achieve the same asymptotic efficiency. As a yield grammar can be transformed into DP recurrences by straightforward formula manipulation, they should no longer be seen as competing methods. Rather, the grammar based approach must be recognized as the explanation of the DP approach on a higher level of abstraction.

Lefebvre has used parsing techniques and *attributed context-free grammars* for sequence alignment and RNA folding (Lefebvre, 1995, 1996). This has resulted in a folding program based on energy minimization and parser generation technology. It has been reported to achieve efficiency similar to DP algorithms. With sophisticated parsing techniques used in place of DP, the method is more amenable to be used by compiler writing specialists.

Stochastic context-free grammars have been used intensively in the area of RNA folding and modelling of RNA families (Eddy and Durbin, 1994; Sakakibara *et al.*, 1994). Calculating probabilities is just a particular way of scoring. In our framework, a stochastic grammar is expressed by a yield grammar, called with a stochastic evaluation algebra that multiplies probabilities along a derivation, and adds probabilities of alternative derivations. Stochastic context-free grammars are typically implemented via the Cocke-Younger–Kasami algorithm (Aho and Ullman, 1972), which is essentially a DP scheme formulated for context-free grammars in Chomsky normal form. Our parser combinators can be seen as an (albeit late) justification of the CYK algorithm by deriving its recurrences from a nondeterministic, recursive parser. Our combinator parsers do not require a normal form, but the width reduction transformation introduced in (Giegerich, 1998) can be used to achieve a similar form where desired for the sake of efficiency.

Stochastic regular grammars are equivalent to *hidden Markov models*, which come along with their own terminology, living ‘in happy ignorance of the Chomksy hierarchy’ (Durbin *et al.*, 1998). The regular case in ADP is a yield grammar that makes no use of the `+~~` and the `~~+` operator; beyond this, it requires no special treatment to reach $O(n^2)$ time efficiency.

The Dynamite system (Birney and Durbin, 1997) provides a *code generation language for DP* recurrences, while our approach is concerned with their development. Although the system can be of great benefit to the expert, it has not (yet) found the widespread use one might expect. One reason certainly is that the development of correct DP recurrences is at least as difficult as their faithful implementation in (say) C. Our approach and the Dynamite system are complementary—we may feed Dynamite with DP recurrences derived via ADP.

It can be quite instructive to re-derive the recurrences of a published DP algorithm via ADP. DP algorithms often are somewhat pragmatic about the SUEs actually evaluated. Some SUEs may be evaluated several times, and there may even be some malformed ones being scored, as long as the scoring functions guarantee that they never score optimally. In such cases, the recurrences cannot be used to calculate statistics about SUEs, and enumerating near-optimal solutions may produce a few surprises. By contrast, a carefully designed grammar leads to a few extra tables to be calculated—increasing space requirements by a constant factor, but with the advantage that recurrences will work correctly with any evaluation algebra, including counting and enumeration of near-optimal solutions.

Scope of the ADP method

This is the inevitable problem of advocating program *development* methods: to introduce a new method, examples should be simple. But are they convincing? Small examples can always be solved without much systematics, so why bother? We hope that the reader has caught on to the essence of the method, and can extrapolate the benefits gained in a context where numerous or more complicated problems have to be solved.

Let us summarize a few points that delineate the scope of the ADP approach. The line of thought goes from more theoretical to most pragmatic concerns.

From the *programming methodology* perspective, it is nice to have an explanation of DP as a compound of simpler techniques—parsing, tabulation and algebraic evaluation. This view seems to apply generally, even in cases where the parsing problem lies beyond context-free languages.

The *parser combinators are domain specific*: RNA folding or gene recognition takes a single input sequence, while (pairwise) alignment takes two. Parser combinators must be adjusted to this, which is a simple exercise. Once defined, they can be re-used for many different analyses in the same problem domain. The declarative understanding of the grammar framework suffices for developing new analyses.

Flexibility is another strength of the ADP method. Although it proposes a rather rigid way of approaching a DP problem, it easily adapts to problem peculiarities without compromising the overall transparency. The lookahead-based parser for recombinant deletions and insertions is a case in point.

Some *intellectual overhead* is always involved in adopting a formal method such as ADP, but any experienced 'DP hacker' will appreciate that, in the form of the annotated grammar and an evaluation algebra, we have an algorithm specification where our ideas are lucidly expressed.

Such an *executable specification* is invaluable in program development. It should be used whenever a functional programming language is available. The grammar is a functional program that tends to be correct from the start—no subscripts, no errors. If there is something wrong with the grammar, it is most likely an error in our modelling of the problem domain, and we are fortunate to catch it early.

The *ultimate derivation of DP recurrences* should be done not by the developer of the algorithm (a particularly nice aspect), but rather by two independent persons. Their results should be cross-checked before implementing them in an imperative programming language.

In *program testing*, the prototype is just as useful. Implementations in (say) C are most effectively validated by comparing their results (via another program, of course) to those of the *Haskell* program. Who else would you trust that the selected alignment is truly optimal?

Other applications and future work

For educational purposes, the ADP method has been applied to classical problems, such as global and local alignment, pattern alignment and a new way of evaluating alignment ambiguity. This is summarized in (Giegerich, 1999), where also some generalizations are discussed. The extension of the edit-distance model to accommodate recombinant alignments is discussed in more detail in (Giegerich *et al.*, 1999). A pattern language to describe motifs in RNA secondary structure is sketched in (Evers *et al.*, 1999), where motifs are essentially described via (sub-) grammars. RNA folding is studied in detail in (Giegerich, 1998), where the ADP method was originally developed. (However, the algebraic nature of the approach is worked out more succinctly in the present treatment.) The full energy minimization algebra for RNA folding has recently been implemented by D. Evers, and will be used to tackle the problem of folding saturated RNA structures, which has been suggested in 1984 (Zuker and Sankoff, 1984) and is still unsolved. To delineate the power of ADP, parser combinators using lookahead or other additional information should be studied systematically. Another goal of our work is a technique to directly translate annotated yield grammars into C, obviating the need to derive the explicit matrix recurrences at all. In this sense, some day we may be able to do DP *without* DP recurrences.

Acknowledgement

The original incentive of this work came from discussions with Gene Myers on yet another set of recurrences for RNA folding. Stefan Kurtz accompanied this work with valuable advice at all stages. Dirk Evers demonstrated that ADP scales up to large problems by implementing a suitable grammar and the full energy minimization algebra for RNA folding. Everything I know about recombination I learned from Georg Weiller. Matthias Höchsmann and Stefan Kurtz derived the DP recurrences for the recombinant alignment program, and M.Höchsmann did the C implementation. Then referees' comments have substantially helped to improve the presentation.

References

Aho,A.V. and Ullman,J.D. (1972) *The Theory of Parsing, Translation, and Compiling*. Prentice-Hall.

Anson,E.L. and Myers,G.W. (1997) Realigner: a program for refining DNA sequence multi-alignments. In *Proceedings of the 1st Conference on Computational Molecular Biology* pp. 9–16.

Barendregt,H.P. (1984) The lambda calculus: its syntax and semantics. *Studies in Logic and the Foundations of Mathematics*. 2nd edn., North-Holland.

Bird,R. (1998) *Introduction to Functional Programming using Haskell*. 2nd edn., Prentice-Hall.

Birney,E. and Durbin,R. (1997) Dynamite: a flexible code generating language for dynamic programming methods. In *Proceedings of the 5th Conference on Intelligent Systems for Molecular Biology* AAAI Press, Menlo Park, CA, USA, pp. 56–64.

Brainerd,W. (1969) Tree generating regular systems. *Information and Control*, **14**, 217–231.

Cormen,T.H., Leiserson,C.E. and Rivest,R.L. (1989) *Introduction to Algorithms*. MIT Press, Cambridge, MA, USA.

Durbin,R., Eddy,S., Krogh,A. and Mitchison,G. (1998) *Biological Sequence Analysis*. Cambridge University Press.

Eddy,S. and Durbin,R. (1994) RNA sequence analysis using covariance models. *Nucleic Acids Res.*, **22**(11), 2079–2088.

Evers,D., Giegerich,R. and Kurtz,S. (1999) A general pattern matching language for specific motifs in RNA secondary structure. In *Proceedings of the 4th European Conference on Theory and Mathematics in Biology and Medicine*.

Gelfand,M.S. and Roytberg,M.A. (1993) A dynamic programming approach for predicting the exon–intron structure. *Biosystems*, **30**, 173–182.

Giegerich,R. (1990) Code selection by inversion of order-sorted derivors. *Theor. Comput. Sci.*, **73**, 177–211.

Giegerich,R. (1998) A declarative approach to the development of dynamic programming algorithms, applied to RNA folding. *Report 98–02* Technische Fakultät, Universität Bielefeld.

Giegerich,R. (1999) Towards a discipline of dynamic programming in bioinformatics. Parts 1 and 2: Sequence comparison and RNA folding. *Report 99–05* Technische Fakultät, Universität Bielefeld,(Lecture Notes).

Giegerich,R., Kurtz,S. and Weiller,G.F. (1999) An algebraic dynamic programming approach to the analysis of recombinant DNA sequences. In *Proceedings of the First Workshop on Algorithmic Aspects of Advanced Programming Languages* pp. 77–88.

Gotoh,O. (1982) An improved algorithm for matching biological sequences. *J. Mol. Biol.*, **162**, 705–708.

Gusfield,D. (1997) Algorithms on strings, trees, and sequences. *Computer Science and Computational Biology*. Cambridge University Press.

Gusfield,D. (ed). (1994) In *Proceedings of the Fifth Annual Symposium on Combinatorial Pattern Matching, Asilomar, California, June 1994* Lecture Notes in Computer Science, **807**.

Hofacker,I.L., Schuster,P. and Stadler,P.F. (1999) Combinatorics of RNA secondary structures. *Discr. Appl. Math.*, **89**, 177–207.

Hutton,G. (1992) Higher order functions for parsing. *Journal of Functional Programming*, **3**(2), 323–343.

Kurtz,S. and Myers,G.W. (1997) Estimating the probability of approximate matches. In *Proceedings of Combinatorial Pattern Matching 1997* Springer Lecture Notes in Computer Science 1246.

Lefebvre,F. (1995) An optimized parsing algorithm well suited to RNA folding. In *Proceedings of the 3rd Conference on Intelligent Systems for Molecular Biology* AAAI Press, Menlo Park, CA, USA, pp. 222–230.

Lefebvre,F. (1996) A grammar-based unification of several alignment and folding algorithms. In *Proceedings of the 4th Conference on Intelligent Systems for Molecular Biology* AAAI Press, Menlo Park, CA, USA, pp. 143–154.

Needleman,S.B. and Wunsch,C.D. (1970) A general method applicable to the search for similarities in the amino acid sequence of two proteins. *J. Mol. Biol.*, **48**, 443–453.

Sakakibara,Y., Brown,M., Hughey,R., Mian,I.S., Sjölander,K., Underwood,R.C. and Haussler,D. (1994) *Recent methods for RNA modeling using stochastic context-free grammars*. (Gusfield, 1994), pp. 289–306.

Searls,D.B. (1988) Representing genetic information with formal grammars. In *Proceedings of the 1988 National Conference of the American Association for Artificial Intelligence* pp. 386–391.

Searls,D.B. (1989) Investigating the linguistics of DNA with definite clause grammars. In *Proceedings of the North American Conference onLogic Programming* MIT Press, pp. 189–208.

Searls,D.B. (1997) Linguistic approaches to biological sequences. **13(4)**, 333–344.

Searls,D.B. and Murphy,K.P. (1995) Automata-theoretic models of mutation and alignment. In *Proceedings of the Third Conference on Intelligent Systems for Molecular Biology, ISMB 95* AAAI Press, Menlo Park, CA, USA, pp. 341–349.

Smith,T.F. and Waterman,M.S. (1981) Comparison of biosequences. *Adv. Appl. Math.*, **2**, 482–489.

Snyder,E.E. and Stormo,G.D. (1993) Identification of coding regions in genomic DNA: an application of dynamic programming and neural networks. *Nucleic Acids Res.*, **21**, 607–613.

Thompson,J.D., Higgins,D.G. and Gibson,T.J. (1994) CLUSTAL W: Improving the sensitivity of progressive multiple sequence alignment through sequence weighting, position-specific gap penalties and weight matrix choice. *Nucleic Acids Res.*, **22**(22), 4673–4680.

Waterman,M.S. (1995) *Introduction to Computational Biology. Maps, Sequences and Genomes*. Chapman & Hall, London, UK.

Weiller,G.F. (1998) Phylogenetic profiles: a graphical method for detecting genetic recombinations in homologous sequences. *Mol. Biol. Evol.*, **15**(3), 326–335.

Zuker,M. (1989) On finding all suboptimal foldings of an RNA molecule. *Science*, **244**, 48–52.

Zuker,M. and Sankoff,S. (1984) RNA secondary structures and their prediction. *Bull. Math. Biol.*, **46**, 591–621.