Bellman’s GAP—a language and compiler for dynamic programming in sequence analysis

Georg Sauthoff¹, Mathias Möhl²,³, Stefan Janssen¹ and Robert Giegerich¹,*

¹Center of Biotechnology and Faculty of Technology, Bielefeld University, 33615 Bielefeld, Germany and ²Department of Computer Science and ³Centre for Biological Signalling Studies (BIOSS), Albert-Ludwigs-Universität 97110 Freiburg, Germany

Associate Editor: Ivo Hofacker

ABSTRACT

Motivation: Dynamic programming is ubiquitous in bioinformatics. Developing and implementing non-trivial dynamic programming algorithms is often error prone and tedious. Bellman’s GAP is a new programming system, designed to ease the development of bioinformatics tools based on the dynamic programming technique.

Results: In Bellman’s GAP, dynamic programming algorithms are described in a declarative style by tree grammars, evaluation algebras and products formed thereof. This bypasses the design of explicit dynamic programming recurrences and yields programs that are free of subscript errors, modular and easy to modify. The declarative modules are compiled into C++ code that is competitive to carefully hand-crafted implementations. This article introduces the Bellman’s GAP system and its language, GAP-L. It then demonstrates the ease of development and the degree of re-use by creating variants of two common bioinformatics algorithms. Finally, it evaluates Bellman’s GAP as an implementation platform of ‘real-world’ bioinformatics tools.

Availability: Bellman’s GAP is available under GPL license from http://bibiserv.cebitec.uni-bielefeld.de/bellmansgap. This Web site includes a repository of re-usable modules for RNA folding based on thermodynamics.

Contact: robert@techfak.uni-bielefeld.de

Supplementary information: Supplementary data are available at Bioinformatics online

Received on April 19, 2012; revised on January 9, 2013; accepted on January 10, 2013

1 MOTIVATION AND BACKGROUND

We review the difficulties experienced in the development of dynamic programming algorithms, and outline how Bellman’s GAP alleviates them.

1.1 Implementing dynamic programming algorithms

Bellman’s GAP is a programming system, which supports the development of dynamic programming (DP) algorithms over sequence data. In bioinformatics, such algorithms are ubiquitous, ranging from sequence alignments and RNA structure prediction to the predictions of RNA interactions or stochastic modeling based on hidden Markov models and stochastic context-free grammars (Durbin et al., 1998).

The development of a DP algorithm traditionally consists of two stages. In the first, creative stage, the recursions of the algorithm are developed, and a scoring scheme is defined. In the second stage, the algorithm is implemented in an imperative programming language such as C, C++ or Java. This stage requires to intermingle recursions and scoring scheme for the sake of runtime efficiency, to decide about space-efficient allocation of tables for intermediate results, to dedicate a substantial amount of work to implement simple, full or even stochastic backtracing and may be to provide an efficient sliding-window version to search through large sequence data. All this requires algorithmic techniques that are known in principle, but must be recreated and adjusted with each application, and which are error-prone to implement and tedious to debug. Later modifications or extensions of the original design tend to affect many lines of code, creating new chances of error in already trusted code.

Bellman’s GAP supports both stages of DP algorithm development. It consists of the language GAP-L and the compiler GAP-C. GAP-L allows to compactly describe the design of DP algorithms in an abstract way. It enforces modularity of components, and thus fosters re-use and extension. GAP-C automatically generates a reasonably efficient C++ implementation of any algorithm specified in GAP-L, contributing the aforementioned implementation techniques.

Automating the implementation stage has a profound backward influence on the design stage. Models can be developed gradually, with their compiled implementations allowing us to test our ideas in many variants as they evolve. For each variant, alternative implementations can be tested by changing some compiler options. The recursions can, for example, either be evaluated bottom-up or top-down, the choice of DP tables can be varied, and both single thread and multi-threaded parallel code under the OpenMP standard can be generated where the scoring scheme allows for this.

1.2 Related work

Bellman’s GAP supports a specific programming method—DP over sequence data—but not a particular application domain. In this sense, it is similar to a compiler for linear algebra problems described in Fabregat-Traver and Bientinesi (2012), which reads equations and properties of the operators involved (such as symmetry, positive definiteness) and then transforms the equations...
to choose an efficient and numerically stable implementation from a large reservoir of numerical algorithms. This has applications, for example, in genome-wide association studies (GWAS), but it is not a GWAS tool. Similarly, Bellman’s GAP helps to implement flowgram denoising or RNA folding algorithms, but by itself, it is not a tool for next-generation read analysis or an RNA-specific tool. Domain-specific knowledge is brought in by the user. Depending how central the DP is in an application scenario, and depending on re-usable modules already available from earlier work in that domain, Bellman’s GAP may decrease overall work more or less.

In the recent Tornado software (Rivas et al., 2012), a (context-free) grammar is at the heart of a declarative specification, whose implementation is automated.

Citing from Rivas et al. (2012), the TORNADO language is designed to provide compact grammar descriptions for a wide range of RNA structural features such as nearest-neighbor dependencies (e.g. stacking rules or mismatches), including higher-than-first-order dependencies, and parameterization of arbitrary loop length distributions.

In contrast, Bellman’s GAP seeks abstractness and modularity rather than compactness, and does not per se support a particular application domain. Input could be a biosequence as well as a sequence of matrix dimensions (matrix chain optimization), an arithmetic formula (El Mamun’s caravan) or a logical clause (3SAT-problem). Using explicit names for evaluation functions turns the grammar into a tree grammar, allowing to encapsulate domain-specific knowledge in evaluation algebras. For example, for RNA structure analysis, several grammars are available together with algebras for energy scoring, partition function, base pair counting, shape abstractions and structure output. To add in stochastic grammars, one needs to provide a stochastic evaluation algebra, together with support for parameter training. All algebras, as well as mechanisms for backtracking, sampling etc., can work with different grammars over the same signature, and owing to algebra products (see below), they can be used in combination.

Finally, let us remark that aside from Bellman’s GAP, two further implementation of the algebraic DP framework are currently under way in other groups.

1.3 Background: algebraic DP

We review, in an informal mode, the concepts of the algebraic discipline of DP (ADP), which circumscribes the scope of problems supported by Bellman’s GAP. See Giegerich et al. (2004) for the original definitions and Sauthoff et al. (2011) for their recent extensions, including the formal semantics of the GAP-L language.

DP algorithms evaluate a search space of solution candidates. A particular problem instance is given by an input sequence, or several thereof. A scoring scheme evaluates candidates in the search space and chooses from them the eventual solution(s), according to some objective function. Most often, this objective is score minimization or maximization, but counting candidates, enumeration and sampling from the search space are also common. Scoring scheme and objective function together constitute an evaluation algebra. The search space arising from concrete input is defined via a problem decomposition scheme, recursively splitting the input problem into smaller sub-problems. Tabulation is used to store and retrieve solutions from sub-problems, to avoid that they are evaluated multiple times. Finally, Bellman’s principle of optimality must hold. It ensures that the objective function distributes over scoring, and hence it suffices to consider only optimal solutions to sub-problems in search of the overall optimum. Other than this condition, there is no restriction on the objective functions that can be provided in an evaluation algebra.

In many applications, especially in biosequence analysis, the recursive problem decomposition naturally follows the sub-word structure of the input sequence(s). In this situation, any candidate can be described as a tree, composed of scoring functions, and bearing the input sequence at its leaves. Evaluating this tree, we get the candidate’s score.

Figure 1 shows two such candidate trees, one representing an alignment of two short amino acid sequences and one representing a secondary structure of an RNA molecule. Substituting concrete functions for the tree symbols rep, del, ins, nil, say by the unit scoring scheme, the alignment candidate evaluates to a score of 3. Substituting base pair counting functions for split, pair, open, nil, the RNA structure candidate evaluates to its number of base pairs, which is 2 in our example.

Which candidate trees need to be considered for a given input? This is conveniently described by a tree grammar, which encodes the logic of problem decomposition in a natural top-down manner.

Sequence alignment by the Needleman–Wunsch algorithm (Needleman and Wunsch, 1970) is described by the grammar Alignment:

\[
\text{Ali} \rightarrow \text{nil} | \text{rep} | \text{del} | \text{ins} \\
< a, b > \text{Ali} < \text{a}, > \text{Ali} < \epsilon, a > \text{Ali}
\]

where Ali is the axiom of the grammar and the only non-terminal symbol, and a, b denote arbitrary characters from the underlying sequence alphabet.

RNA folding can be described by the grammar RNAStruct that reflects the algorithm of Nussinov et al. (1978):

\[
N \rightarrow \text{nil} | \text{open} | \text{split} \\
< a, \text{N} > \text{pair} < a, \text{N} > \text{pair}
\]

with the three cases for empty structures, structures where the first base \(a\) is unpaired and those where the first base \(a\) is paired with a complementary base \(\hat{a}\).

Whatever our grammar is, for given input \(x\), all the trees generated by the grammar, which spell out \(x\) at their leaves, constitute the search space. Grammar RNAStruct accepts a single input string, while grammar Alignment is a two-track grammar and expects two strings, indicated by the use of \(<, >\).

A grammar and an evaluation algebra together precisely define the problem class to be solved. Expressed in GAP-L, all the technical ingredients of a DP algorithm—concrete recursions, tabulation decisions, backtracking and more—can be generated automatically.

Having designed a tree grammar \(\mathcal{G}\) and an evaluation algebra \(\mathcal{E}\), we simply compile. This turns the grammar into an evaluator of the search space defined by it, which by convention is also named \(\mathcal{G}\). On input \(x\), we call \(\mathcal{G}(\mathcal{E}, x)\) to obtain the desired solution.
2 THE BELLMAN’S GAP LANGUAGE AND COMPILER

2.1 GAP-L language features

The signature construct in GAP-L is similar to an interface declaration in Java. For the alignment example, we use

```java
signature Align(alphabet, answer) {
    answer rep(<alphabet, alphabet>, answer);
    answer del(<alphabet, void>, answer);
    answer ins(<void, alphabet>, answer);
    answer nil(<void, void>);
    choice [answer] h([answer]);
}
```

With the name of the signature, generic names for the input alphabet and the results of candidate evaluation are specified. The declarations of the function symbols follow. The use of `void` indicates that `del` reads a character form the first track only, and `ins` only from the second; `[and]` indicates list types. The objective function, marked by `choice`, always computes on lists.

An algebra definition assigns concrete data types to the generic names and provides implementations for the function symbols. Algebra `score` uses distance minimization, and scores mismatches with 3 and gaps with 4:

```java
algebra score implements
Align(alphabet = char, answer = int) {
    int rep(<char a, char b>, int m) {
        if (a == b) return m; else return m + 3;
    }
    int del(<char g, void>, int m) {
        return m + 4;
    }
    int ins(<void, char g>, int m) {
        return m + 4;
    }
    int nil(<void, void>) { return 0; }
    choice [int] h([int] l) { return list(minimum(l)); }
}
```

In the tree grammar, terminal symbols are, by convention, written in uppercase, and tree patterns are written like function declarations with underscores.

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

The operator `#` explicitly specifies that the objective function `h` is used to choose from alternative candidates.

Let `print` be another algebra that computes a visualization of a candidate as an alignment, i.e. the two input sequences padded with gaps. Used in a product such as

```
instance main = alignment(score * print);
```

we obtain the optimal score together with the candidates that achieve it. Each instance declaration implements a separate

---

Fig. 1. Left: tree representing an alignment of the amino acid sequences `DARLING' and `AIRLINE’. `rep(ace), ins(ert) and del(ete) denote the typical edit operations, and `nil` denotes an empty alignment. Right: tree representing a secondary structure assigned to an RNA sequence. `open` indicates a base pair enclosing a sub-structure, `split` a branching structure, `open` an unpaired base next to a sub-structure, and `nil` the empty sub-structure.
algorithm, for which GAP-C produces specialized code, depending on the algebras involved.

The signature for the RNA folding example is:

```plaintext
signature Nuss(alphabet, answer) {
  answer split(answer, answer);
  answer pair(alphabet, answer, alphabet);
  answer open(alphabet, answer);
  answer nil(void);
  choice [answer] h([answer]);
}
```

A simple algebra for base pair maximization is

```plaintext
algebra bpmax implements
  Nuss(alphabet = char, answer = int)
  int split(int x, int y) (return x+y);
  int pair(char a, int x, char b) (return x+l);
  int open(char a, int x) (return x);
  int nil(void) (return 0);
  choice [int] h([int l]) (return list(maximum(l)));
}
```

The RNAStruct grammar, which models the search space of all structures for the inout sequence, is a little bit more interesting than grammar alignment, as not all trees built from split, pair, open, nil are allowed:

```plaintext
grammar RNAStruct uses Nuss(axiom=N) {
  N = nil(EMPTY) |
  open(CHAR, N) |
  split(pair(CHAR, N, CHAR) with basep, N) # h;
}
```

This expresses that a candidate of the form

```
split(pair(CHAR, x, CHAR), pair(CHAR, y, CHAR))
```

is not welcome as part of the search space, as this structure already has a (unique) representation as

```
split(pair(CHAR, x, CHAR),
     split(pair(CHAR, y, CHAR), nil(EMPTY)))
```

In addition, the grammar further restricts the search space with the syntactic filter basep. This predicate guards the application of the tree pattern pair(CHAR, N, CHAR). It is only evaluated if the first and last characters are complementary to each other.

```plaintext
algebra cnt auto count
```

asks the compiler to generate an algebra cnt, which counts the number of candidates, and we leave it to the reader to write an algebra print that converts a candidate into a dot-bracket string (see also supplement). We declare two instances

```plaintext
instance cooptnum = RNAStruct(bpmax * cnt)
instance cooptimals = RNAStruct(bpmax * print)
```

where cooptnum determines the maximum number of base pairs, and the number of different structures that achieve it, and cooptimals computes and outputs all the co-optimal structures.

### 2.2 Products in GAP-L

Evaluation algebras have certain (non-exclusive) properties that are relevant when it comes to their use in combination. Algebra A is

- **unitary**, if it returns a single result (e.g. simple minimization, counting),
- **set valued**, if the computed multi-sets do not contain multiple entries (e.g. k-best minimization),
- **selective**, if the choice function always returns a sub-multi-set of a given multi-set (e.g. min(k) or enumeration),
- **enumerative**, if its choice function is \( h(X) = X \),
- **synoptic**, if \( h \) computes new values from candidate values, which are not associated with a particular candidate (e.g. counting, summation, average).

These properties are used by the compiler for certain optimizations, but more importantly, they govern the use of products. Generally, a product in ADP is an operation that takes two algebras \( A \) and \( B \) over the same signature and creates combined algebra, say \( A \times B \). To achieve such an effect in traditional DP recurrences, almost every line of the code would have to be modified. Product operators allow to achieve this combination by a single keystroke.

Bellman’s GAP offers five product operators (see Table 1). Products can be combined with a user-defined semantic filter, using a suchthat clause. The use of these algebra products will be demonstrated in the examples below.

### 2.3 Compiler features

The main task of the compiler GAP-C is to analyze the grammar and the algebra products used in instance declarations. From these, it derives recurrences (akin to a traditional formulation of a DP algorithm) for which imperative C++ code is generated.

As with the design of the language GAP-L, a goal of the compiler is to make ADP convenient to use. For this, the front-end of the compiler pedantically keeps track of source code locations to report parse errors to the column. In the middle-end, GAP-C includes semantic analyses for better diagnostics. For example, a type-checker pinpoints missing arguments or wrong types when using function symbols in algebras or the grammar. A product analysis checks whether a product is known to violate Bellman’s principle, and a runtime analysis phase computes the asymptotically optimal runtime and verifies whether the designer’s choice of table allocation (when explicitly given) actually achieves such a runtime.

We look at two of the optimizations that contribute to generating C++ target code that is competitive with handwritten one. For a deeper treatment, see Sauthoff et al. (2011).

GAP-C can generate code for different backtracing schemes. Backtracing is normally used to compute the structure of an optimal candidate (e.g. a string representation of an alignment or a secondary structure) after the optimal score is computed. Using backtracing is a performance optimization, as the candidate reconstruction is only done for those sub-solutions that are part of the optimal solutions. In the case of an alignment algorithm, this saves constant runtime factors and space; for RNA
secondary structure prediction, the forward computation finding the maximum number of base pairs is in \(O(n^3)\), while the backtracing for the optimal candidate(s) is in \(O(n^2)\).

For our first GAP-L code example, the product \texttt{score*print} computes the optimal alignment score and the alignment itself. With a compiler switch, GAP-C automatically generates backtracing code for the algebra print. Thus, only the algebra \texttt{score} is computed in the forward computation. By default, the backtracing code automatically outputs all co-optimal candidates. For the flowgram alignment variant below (Section 3.1), this is not necessary, and we can specify this via another compiler switch.

Further supported backtracing schemes are sub-optimal backtracing, backtracing over \((k\text{-best})\) classes during classification and stochastic backtracing (sampling) under an algebra that computes a probability distribution.

Another performance-critical optimization of GAP-C is table design. The table design phase automatically decides which intermediate results need to be stored in a table to achieve asymptotically optimal runtime, and which recurrences are not tabulated across \(O(n^2)\). In case of an insertion or deletion, the grammar has to parse exactly four numbers of a whole frame, a sub-sequence of four numbers. We add two number codes the measured amount of light for the (non-)integration event of one of the four nucleotides during sequencing. Errors during pyrosequencing that lead, for example, to inserted or deleted frames in a flowgram are called noise. Taking flowgram data as-is in the analysis of meta-genomic samples would inflate the number of operational taxonomic units (Quince et al., 2011). Denoiser clusters similar flowgrams together to reduce the noise.

As part of the clustering step, Denoiser computes sequence alignments of flowgrams. The authors used a custom Haskell embedding of algebraic DP to implement the alignment algorithm for rapid prototyping, easy evaluation of the scoring function and optimizations. The algorithm is sketched in Reeder and Knight (2010), but details are not given. In the following, we create a GAP-L version of the Reeder-Knight flowgram denoiser, using the basic pairwise alignment algorithm from Section 2.1 as starting point.

In the alignment of flowgrams, insertions or deletions refer to a whole frame, a sub-sequence of four numbers. We add two function symbols, \(t_i\) for a terminal insertion and \(t_d\) for deletion, such that we can score them differently. The changes to the signature \texttt{Align} are:

\[
\text{signature} \quad \text{FlowAlign(alphabet, answer)} \{ \\
\text{answer rep(alphabet, alphabet), answer)}; \\
\text{answer del(Subsequence, void), answer)}; \\
\text{answer ins(void, Subsequence), answer)}; \\
\text{answer nil(void, void)}; \\
\text{answer ti(void, int)}; \\
\text{answer td(int, void)}; \\
\text{choice [answer] h([answer])}; \\
\}
\]

After that, we have to adjust the grammar to make use of the modified and added function symbols. In case of an insertion or deletion, the grammar has to parse exactly four numbers of a sequence. We model this with syntactic filters that restrict the size of the parsed sub-sequence:

\[
\text{grammar} \quad \text{flow uses FlowAlign(axiom = ali)} \{ \\
\text{gap = REGION with minsize(4) with maxsize(4)}; \\
\text{ali = \{rep<CHAR, CHAR>, ali\}} | \\
\text{del<gap, EMPTY>, ali}} | \\
\text{ins<EMPTY, gap>, ali}} | \\
\text{nil<EMPTY, EMPTY>} \}
\]
In GAP-L, we can directly communicate this via modifying the existing algebra:

```plaintext
algebra mismatch implements
FlowAlign(alphabet = float, answer = int) {
  int rep(<float a, float b>, int m)
  { return m + abs(round(a) - round(b)); }
  int del(<Subsequence g, void>, int m)
  { return m + round(g[0]) + round(g[1]) + round(g[2]) + round(g[3]); }
  int ins(<void, Subsequence g>, int m)
  { return m + round(g[0]) + round(g[1]) + round(g[2]) + round(g[3]); }
  choice [int] h([int] l)
  { return list(minimum(l)); }
}
```

For computing a relative mismatch score, we use the algebra seqlen that scores like mismatch except for the replace case. In GAP-L, we can directly communicate this via modifying the existing algebra:

```plaintext
algebra seqlen extends mismatch {
  int rep(<float a, float b>, int m)
  { return m + max(abs(round(a)) + round(b)); }
}
```

In Denoiser, the flowgram alignment component provides a mode to compute both relative scores simultaneously, which can be achieved with the following product:

```plaintext
instance RKdenoiser =
  flow(score * (length % mismatch % seqlen))
```

This specifies that for the candidate with the minimal score, also the alignment length, the mismatch score and mismatch length are computed. In all four algebras, insertions and deletions at the terminal end are scored zero, as flowgrams may differ in size, but only an insertion or deletion at the $S'$ end is plausible with this input data. In this way, we compute a variant of a local alignment algorithm.

### 3.2 Programming with modular components

Working in a specific application area, one builds up a library of modules—signatures, grammars and algebras—from which a variety of tasks can be compiled using algebra product. Each novel algebra product gives rise to a program instance (cf. Section 2.1), from which the compiler produces an executable specialized to this task.

Our repository for RNA folding is a result of the study in Janssen et al. (2011). It holds separate collections of signatures, grammars and algebras, and GAP-L main programs that merely consist of instance declarations. Among others, the repository contains the modules shown in Table 2. Counting algebras are not in the repository, as they are constructed by the compiler on request. Instances of the form
Table 2. Modules related to RNA structure prediction based on thermodynamics

<table>
<thead>
<tr>
<th>Type</th>
<th>Name</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>Signature</td>
<td>foldrna</td>
<td>Describes RNA folding space</td>
</tr>
<tr>
<td>Grammar</td>
<td>nodangle</td>
<td>Model without dangling bases</td>
</tr>
<tr>
<td>Grammar</td>
<td>overdangle</td>
<td>Model with overaggressive dangling</td>
</tr>
<tr>
<td>Grammar</td>
<td>microstate</td>
<td>Correct dangling with ...</td>
</tr>
<tr>
<td>Grammar</td>
<td>macrostate</td>
<td>...Without candidate space extension</td>
</tr>
<tr>
<td>Grammar</td>
<td>nodangle_lp</td>
<td>Model allowing lonely pairs</td>
</tr>
<tr>
<td>Algebra</td>
<td>mfe</td>
<td>MFE computation</td>
</tr>
<tr>
<td>Algebras</td>
<td>pfunc</td>
<td>Boltzmann weights (partition function)</td>
</tr>
<tr>
<td></td>
<td>pfunc_id</td>
<td>Individual candidate Boltzmann weight</td>
</tr>
<tr>
<td>Algebras</td>
<td>shapes1...5</td>
<td>Classification by shape abstraction</td>
</tr>
<tr>
<td>Algebra</td>
<td>dotBracket</td>
<td>Printing structures</td>
</tr>
</tbody>
</table>

(1) nodangle (shape_i*pfunc suchthat
pfunc_filter)

for i = 1, ..., 5 were used in Janssen et al. (2011) for all four grammars to evaluate the effect of different treatments of dangling bases on the shape probabilities. The `suchthat` clause activates a filter that weeds out shapes with accumulated probabilities <10^-6, which was shown to speed up the computation without significantly distorting results. Instance

(2) nodangle(((shape5*(mfe % pfunc)) suchthat
pfunc_filter_allPP) *dotBracket)

must be compiled with option -kbacktrace, adds in the computation of mfe structures for each shape and reports them via algebra dotBracket. This is an interesting use of the Cartesian product, where two independent results for each shape are computed simultaneously, the shape’s mfe value and the accumulated contribution of all structures with this shape to the partition function. Instance

(3) nodangle((pfunc[pfunc_id]*(shape2*
mfe*dotBracket)) suchthat sample_filter)

is compiled with option -kbacktrace. It computes a stochastic sample from the partition function, where for each candidate sampled, its level 2 shape, free energy and dot-bracket string are reported. This allows to estimate shape probabilities by sampling and comparing them with their exact computation via (1).

(4) Finally, the general technique of classified DP is realized as follows. Consider an algebra \( A \), which computes from each candidate a classification attribute, not performing any optimization. This splits the candidate space into disjoint classes. Then, a product of form \( (A \star B) \) performs classified DP, where the analysis carried out by \( B \) (which can be a simple algebra as well as an arbitrary product) applies to each class separately.

From the viewpoint of software technology, the important aspect here is that each algebra is designed and tested with a single purpose in mind. Then, product operations allow to define more sophisticated analyses in great variety—without making changes to existing modules.

4 EVALUATION OF BELLMAN’S GAP AS AN IMPLEMENTATION PLATFORM

In this section, we evaluate Bellman’s GAP based on our experience of using it in bioinformatics tool development.

4.1 Code quality

Our goal is not to reproduce existing tools with more efficient code, but to ease development of novel applications. Still, the generated code must reach efficiency comparable with hand-coded implementations. In terms of asymptotics, the compiler always achieves the best possible asymptotic runtime (cf. Sauthoff et al., 2011), but also makes considerations about space consumption and constant factors. We compare against the hand-crafted applications UNAfold and RNAfold. Note that UNAfold is rather new, whereas RNAfold has been carefully engineered over more than a decade and has the reputation of being the fastest RNA folding program. For comparison, we use instance microstate(mfe * dotbracket) for MFE structure prediction. This is equivalent with RNAfold, but also reports co-optimals. Measurements show that in terms of speed, Bellman’s GAP code lies between the hand-coded tools (Fig. 2). Memory consumption is similar to UNAfold, with a tendency to be more space efficient on longer sequences. Note the outlier points above the yellow memory curve. These can be traced back to sequences that have several different co-optimal structures. Especially with grammar microstate, their number can be large; the maximum observed in this data set was 384 for a sequence of length 476. Our automatic backtrace generates a somewhat wasteful list of all co-optimals in such a case before producing output.

Bellman’s GAP has been extensively evaluated against the second-generation implementation of ADP. The tool RNAshapes predicts RNA secondary structure classified by abstract shapes (Steffen et al., 2006), and the tool pknotsRG predicts structures including simple pseudoknots and kissing hairpin interactions (Theis et al., 2010). Both were originally implemented in the Haskell-embedded version of ADP. They were re-coded in GAP-L and compiled with GAP-C, achieving constant-factor speed-ups. pknotsRG, for example, is two to three times as fast while using about half the space. For more details, see Sauthoff (2011).

4.2 Assessing convenience

In the previous sections, we have described how abstractness, modularity and automated compilation help to achieve reliable implementations with less human effort. This cannot be quantified easily. As a crude approximation to implementation effort, we here compare code sizes for ‘real-world’ applications.

In Janssen et al. (2011), the impact of four different implementations of the Turner energy model of RNA folding was investigated. These implementations differ for pragmatic reasons with respect to their treatment of dangling bases. RNAfold, from the Vienna package (Lorenz et al., 2011), provides three variants via options -d0, -d1 and -d2, called NoDangle, Microstate and
OverDangle for comparison, respectively, while RNAshapes uses a fourth model, Macrostate. We compared models by measuring their effect on shape probabilities, which provides a more robust measure than single minimum free energy structure prediction. Because RNAfold does not compute shape probabilities, rather than augmenting the source code from the Vienna package, it was easier for us to re-implement all three versions in Bellman’s GAP, and combine them with probabilistic shape analysis by a product of algebras already available from RNAshapes.

Table 3 reports on the size of the grammars and algebras arising for these applications. We specify sizes of grammar and algebras, as well as the size of the generated code for certain instances. Note the significant difference between the number of grammar non-terminals (recurrences) and the number of tables actually allocated to compute them. The code size reduction (size of GAP-L code/size of C++ code) gives an idea of how much programming and debugging time is saved in ‘real-world’ scenarios by using Bellman’s GAP. In the table, we included, from our ongoing research, a remake of the Rfam model RF00553. Covariance models are specialized toward a particular structure, leading to grammars with hundreds of non-terminal symbols and rules. The example is included to show that Bellman’s GAP successfully handles grammars of this size.

5 CONCLUSION

5.1 Adopting Bellman’s GAP in education

At Bielefeld and Freiburg universities, starting 2011, DP based on Bellman’s GAP has been taught to small cohorts of bioinformatics M.Sc. students. This paragraph reports from our teachers’ observations; it does not pretend to be a formal evaluation of student success. Our students have a limited experience in DP the ‘hard’ way, and generally find it rewarding to invest in the intellectual overhead of adopting the algebraic discipline of DP. Bellman’s GAP as a language appears to be a minor problem, owing to its small size and resemblance of C and Java. A didactic advantage is the high-level error messages generated by the compiler. They lead to less frustration and debugging time among students and reduce supervision required by the teacher. Aside from type errors and missing arguments, the compiler detects also infinite cyclic recursions and warns when some recursion leads to exponential complexity. In early assignments, we rely on automatic table design feature by GAP-C, allowing to keep things simple at the beginning. Later, when space efficiency concerns are covered in the lecture, students design their own table allocation and compare it with the one automatically derived by the compiler. Abstract specifications and ease of implementation also foster a deeper understanding and encourage experimentation. How is the size of the search space affected by splitting a sub-case in two separate cases? A counting algebra provides the answer. Are the shape representative structures the same under two different scoring schemes? Calling nodangle(shape * (score1 % score2).dotbracket) generates first evidence. In this manner, exploration of modeling alternatives is supported.

5.2 Scope of applications

We circumscribe the scope of Bellman’s GAP by some present and forthcoming applications.
Bellman’s GAP for DP

5.3 Future development as a community effort

In contrast to the majority of tools in bioinformatics, which serve biologists to analyze their data, Bellman’s GAP is a tool for bioinformaticians. We have reported our experience, by which Bellman’s GAP helps to create more reliable and more versatile DP algorithm in a shorter time. By the first release of the Bellman’s GAP system, we hope to share this experience with the bioinformatics community, and together create libraries of re-usable specifications for a variety of applications. To this end, we have created a repository of Bellman’s GAP modules related to thermodynamic folding and abstract shape analysis of RNA. The repository named fold-grammars as well as the Bellman’s GAP system are available open source under GPL license from http://bibiserv.cebitec.uni-bielefeld.de/bellmansgap.

ACKNOWLEDGEMENT

We thank Jens Reeder for providing the Haskell-ADP code of the flowgram denoiser. We also appreciate the detailed discussions with participants of the Benasque RNA workshop in 2012.

Funding: This work was partially supported by the German Research Foundation (DFG GRK 635, BA 2168/3-1, MO 2402/1-1).

Conflict of Interest: none declared.

REFERENCES


