How to Multiply Dynamic Programming Algorithms (Supplemental Material)

Christian Höner zu Siederdissen[1], Ivo L. Hofacker[1−3], and Peter F. Stadler[4,1,3,5−7]

[1] Dept. Theoretical Chemistry, Univ. Vienna, Währingerstr. 17, Wien, Austria
[2] Bioinformatics and Computational Biology research group, University of Vienna, A-1090 Währingerstraße 17, Vienna, Austria
[3] RTH, Univ. Copenhagen, Grønnegårdsvej 3, Frederiksberg C, Denmark
[4] Dept. Computer Science, and Interdisciplinary Center for Bioinformatics, Univ. Leipzig, Härtelstr. 16-18, Leipzig, Germany
[5] MPI Mathematics in the Sciences, Inselstr. 22, Leipzig, Germany
[6] FHI Cell Therapy and Immunology, Perlickstr. 1, Leipzig, Germany
[7] Santa Fe Institute, 1399 Hyde Park Rd., Santa Fe, USAx

1 DNA-Protein Alignment

In this section we discuss one elaborate and practically relevant example where a grammar product of two simple grammars yields a complex result grammar. The alignment of two sequences of the same type is typically simplified due to mirrored operations. Recalling the alignment grammar from above, we speak of in/del operations as an insertion in one sequence may just as well be described as a deletion in the other sequence. In addition, it does not matter which sequence is bound to which input tape.

The alignment of a protein sequence to a DNA sequence is, however, more involved. In Fig. 1 we summarize this more elaborate example.

It is important to recognize that the grammar alone is a device that enumerates *all* possible alignments of a DNA sequence with a protein sequence. In particular, the grammar itself will not disallow alignments that are biologically unsound. However, each grammar created using our framework has all of its rules tagged with function symbols. These functions symbols are also known as algebra symbols [1]. Such a grammar is also very similar to S-attribute grammars [2].

The DNA sequence is read in one of three reading frames (RFs), and a deletion or insertion does not yield a "simple" in/del but also a frame shift. This more advanced treatment of DNA characters in triplets is due to the *translation* of DNA into protein in steps of three nucleotides, the "codons" of the genetic code. In Fig. 1 frame shifts (with scoring functions `rf1`, `rf2`) are allowed only at high cost as they change the transcription of following protein characters completely. Staying within a frame is very cheap, even if this involves the deletion of three characters (`del`).

```
Grammar: DNA
NT: F{3}
T:  c
F{i} -> stay <<< F{i}   c c c
F{i} -> rf1  <<< F{i+1} c c
F{i} -> rf2  <<< F{i+2} c
F{i} -> del  <<< F{i}
//
```

$$(F_0) \to (F_0) \qquad (F_1) \to (F_1)(c)(c)(c)$$
$$(F_0) \to (F_0)(c)(c)(c) \qquad (F_1) \to (F_2)(c)(c)$$
$$(F_0) \to (F_1)(c)(c) \qquad (F_2) \to (F_0)(c)(c)$$
$$(F_0) \to (F_2)(c) \qquad (F_2) \to (F_1)(c)$$
$$(F_1) \to (F_0)(c) \qquad (F_2) \to (F_2)$$
$$(F_1) \to (F_1) \qquad (F_2) \to (F_2)(c)(c)(c)$$

```
Grammar: DNAdone
NT: F{3}
T:  empty
F{i} -> nil <<< empty
//
```

$$(F_0) \to (\varepsilon)$$
$$(F_1) \to (\varepsilon)$$
$$(F_2) \to (\varepsilon)$$

```
Grammar: DNAstand
NT: F{3}
F{i} -> del <<< F{i}
//
```

$$(F_0) \to (F_0)$$
$$(F_1) \to (F_1)$$
$$(F_2) \to (F_2)$$

```
Grammar: PRO
NT: P
T:  a
T:  empty
P -> amino <<< P a
P -> del   <<< P
//
```

$$(P) \to (P)$$
$$(P) \to (P)(a)$$

```
Grammar: PROdone
NT: P
T:  empty
P -> nil <<< empty
//
```

$$(P) \to (\varepsilon)$$

```
Grammar: PROstand
NT: P
P -> del <<< P
//
```

$$(P) \to (P)$$

```
Product: DnaPro
DNA >< PRO
+ DNAdone >< PROdone
- DNAstand >< PROstand
//
```

$$\binom{F_0}{P} \to \binom{F_0}{P}\binom{\epsilon}{a} \qquad \binom{F_1}{P} \to \binom{F_1}{P}\binom{c}{a}\binom{c}{\epsilon}\binom{c}{\epsilon}$$
$$\binom{F_0}{P} \to \binom{F_0}{P}\binom{c}{\epsilon}\binom{c}{\epsilon}\binom{c}{\epsilon} \qquad \binom{F_1}{P} \to \binom{F_2}{P}\binom{c}{\epsilon}\binom{c}{\epsilon}$$
$$\binom{F_0}{P} \to \binom{F_0}{P}\binom{c}{a}\binom{c}{\epsilon}\binom{c}{\epsilon} \qquad \binom{F_1}{P} \to \binom{F_2}{P}\binom{c}{a}\binom{c}{\epsilon}$$
$$\binom{F_0}{P} \to \binom{F_1}{P}\binom{c}{\epsilon}\binom{c}{\epsilon} \qquad \binom{F_1}{P} \to \binom{\varepsilon}{\varepsilon}$$
$$\binom{F_0}{P} \to \binom{F_1}{P}\binom{c}{a}\binom{c}{\epsilon} \qquad \binom{F_2}{P} \to \binom{F_0}{P}\binom{c}{\epsilon}\binom{c}{\epsilon}$$
$$\binom{F_0}{P} \to \binom{F_2}{P}\binom{c}{\epsilon} \qquad \binom{F_2}{P} \to \binom{F_0}{P}\binom{c}{a}\binom{c}{\epsilon}$$
$$\binom{F_0}{P} \to \binom{F_2}{P}\binom{c}{a} \qquad \binom{F_2}{P} \to \binom{F_1}{P}\binom{c}{\epsilon}$$
$$\binom{F_0}{P} \to \binom{\varepsilon}{\varepsilon} \qquad \binom{F_2}{P} \to \binom{F_1}{P}\binom{c}{a}$$
$$\binom{F_1}{P} \to \binom{F_0}{P}\binom{c}{\epsilon} \qquad \binom{F_2}{P} \to \binom{F_2}{P}\binom{\epsilon}{a}$$
$$\binom{F_1}{P} \to \binom{F_0}{P}\binom{c}{a} \qquad \binom{F_2}{P} \to \binom{F_2}{P}\binom{c}{\epsilon}\binom{c}{\epsilon}\binom{c}{\epsilon}$$
$$\binom{F_1}{P} \to \binom{F_1}{P}\binom{\epsilon}{a} \qquad \binom{F_2}{P} \to \binom{F_2}{P}\binom{c}{a}\binom{c}{\epsilon}\binom{c}{\epsilon}$$
$$\binom{F_1}{P} \to \binom{F_1}{P}\binom{c}{\epsilon}\binom{c}{\epsilon}\binom{c}{\epsilon} \qquad \binom{F_2}{P} \to \binom{\varepsilon}{\varepsilon}$$

Fig. 1. Atomic grammars for the DNA-Protein alignment example. **(I)** Nucleotides are read in triplets (three nucleotides each). The `DNA` grammar switches between reading frames. `DNAdone` and `DNAstand` handle the termining and looping case. **(II)** The `PRO`tein grammar works similarly, but reads only a single amino acid at a time. The expansion of the `DNA` grammar is more complicated, as the indexed non-terminal symbol F expands to three different non-terminals corresponding to the three possible reading frames. **(III)** The grammar product of `DNA` and `PRO`tein without the looping case "`stand`" and with the terminating case "`done`". In code, `><` represents the direct product ($\otimes$).

Correspondingly, the author of the DNA-Protein grammar will keep in mind that production rules tagged with one of the `rf1`, `rf2` rules should be given a high cost when implementing the scoring algebra.

As each of the three frames, and shifts to the other two frames, is by itself similar to the other two frames, a special encoding saves a lot of work. The F non-terminal indicating the current frame is indexed with indices 0, 1, and 2. Frame shifts are thus calculated *modulo 3* instead of explicitly creating all three frame indices F_0 to F_2 and their corresponding production rules.

The protein grammar, on the other hand, has the same simple structure as our previous atomic components of the alignment grammar. Here, we indeed only read a single amino acid, or handle a deletion.

The complexity of the DNA-protein alignment stems from the fact that we need to "align" the different frame shifting possibilities in the DNA input while matching zero to three nucleotides to zero or one amino acid in the protein input. In addition, once a frame shift has occurred all following alignments of three nucleotides against one amino acid are scored in the new reading frame until another frame shift occurs or the alignment is completed.

In general, our framework simplifies the complexity of designing this algorithm considerably. While the *combined* grammar is highly complex, the individual grammars are rather simple. As already mentioned, the protein "stepping grammar" is one of the simplest possible ones. The DNA grammar is more complex as we need to handle stepping and frame shifts in all three reading frames. But considering that we allow indexed non-terminals and calculations on these indices (modulo 3 in the frame shift case), even the frame shift grammar has only four rules, just twice as much as the simplest stepping grammar.

The resulting 24-production rule grammar is easily calculated in our frame work. We like to point out that we may easily extend this grammar to allow for, say, an alignment of two DNA sequences with two protein sequences. This grammar can be *calculated* at basically no additional cost but would pose a daunting task if implemented by hand.

In addition to the 24-production rule grammar, a set of 24 *function types* is created. This signature, as called in Algebraic dynamic programming [1, 3], associates one type with each of the production rules. For example, the production rule created from the product of `F{i} -> stay <<< F{i} c c c` and `P -> amino <<< P a` creates, among others, the production rule $\binom{F_0}{P} \to \binom{F_0}{P}\binom{c}{a}\binom{c}{\epsilon}\binom{c}{\epsilon}$.

The corresponding evaluation function `stay_amino` has type $\mathbb{S} \times (\mathcal{N} \times \mathcal{A}) \times (\mathcal{N} \times \mathcal{E}) \times (\mathcal{N} \times \mathcal{E}) \to \mathbb{S}$. This type specifies that `stay_amino` excepts the score (of type $\mathbb{S}$) of the alignments as calculated up to this position followed by tuples of characters read from each tape. The first nucleotide character (of the set $\mathcal{N} = \{\text{A,C,G,U}\}$) is aligned with the corresponding amino acid character (of the set $\mathcal{A}$ of the 20 amino acids). The remaining two nucleotides are aligned with elements from the empty set ($\mathcal{E}$ which emits the "noninformative character" represented by the empty tuple `()`). This slightly unusual design allows us to

effectively align the single amino acid character with three nucleotide characters. Finally, the `stay_amino` function emits a score of type $\mathbb{S}$.

The user can now *implement* the required scoring functions. `stay_amino` can, for example, be implemented as follows:
`stay_amino s (c1,a) (c2,()) (c3,()) = s + lookupCodon c1 c2 c3 a`.
Here a function `lookupCodon` is presumed to exist that returns the score (probability, log-odds) of the alignment of the amino acid `a` with the triplet of nucleotides `c1 c2 c3`.

Note that the creation of candidate alignments via the grammar is completely separate from the actual *scoring* of such an alignment via scoring algebras. The amalgamation of the two concepts *grammar* and *algebra* is done by the underlying `ADPfusion` framework [3] which also optimizes the resulting code such that its running time performance is competitive with hand-written `C`-based implementations.

References

1. Giegerich, R., Meyer, C.: Algebraic Dynamic Programming. In: Algebraic Methodology And Software Technology. Volume 2422. Springer (2002) 243–257
2. Lefebvre, F.: An optimized parsing algorithm well suited to rna folding. In: ISMB. (1995) 222–230
3. Höner zu Siederdissen, C.: Sneaking around concatMap: efficient combinators for dynamic programming. In: Proceedings of the 17th ACM SIGPLAN international conference on Functional programming. ICFP '12, New York, NY, USA, ACM (2012) 215–226