Dynamic Programming for Set Data Types Electronic Supplement

Christian Höner zu Siederdissen[1], Sonja J. Prohaska[2], and Peter F. Stadler[2]

[1] Dept. Theoretical Chemistry, Univ. Vienna, Währingerstr. 17, Wien, Austria
[2] Dept. Computer Science, and Interdisciplinary Center for Bioinformatics, Univ. Leipzig, Härtelstr. 16-18, Leipzig, Germany

Illustrative Examples

DP for RNA Folding

A good example for Dynamic Programming over sequences is RNA folding. We consider here only a minimal example based on the grammar with non-terminals S and B denoting arbitrary structures and secondary structures enclosed by a base pair respectively. We write terminals in the usual shorthand notation as $\bullet$ for an unpaired base, while (and) denotes a base pair. There are just five productions in the usual RNA CFG

$$S \to B \mid \bullet S \mid BS \mid \bullet \qquad B \to (S) \tag{1}$$

The corresponding evaluation algebra that counts base pairs amounts to addition, with the terminals $\bullet$ and $(\dots)$ evaluating to 0 and 1, resp.

The productions of the RNA folding grammar, equ. (1), can be viewed as a set of concrete decompositions for a given input. For instance,

$$\begin{aligned} &\text{“}S \to BS\text{”} := \{S_{ij} \mapsto B_{ik} S_{k+1,j} | 1 \leq i \leq k < j \leq n\} \\ &\text{“}B \to (S)\text{”} := B_{ij} \mapsto \langle i, j\rangle +\!\!+ S_{i+1,j-1} \end{aligned} \tag{2}$$

Similar expressions can immediately be written down for “$S \to \bullet S$” and “$S \to B$”. These sets depend explicitly on the input since n is the length of the input sequence. This is not unexpected since the search space of course depends on the input.

In the RNA case, all relevant sets are intervals. The unconstrained structures have empty interfaces. We have $\mathsf{S} := [S, \varnothing]$ and hence $\mathsf{S}^* = [X \setminus S, \varnothing]$ where $X = [1, \dots, n]$ is the set of sequence positions. The B-objects have the endpoints as interfaces $\mathsf{B} := [B \setminus \{i, j\}, \langle i, j\rangle]$. Thus $\mathsf{B}^* := [(X \setminus B) \setminus \{i, j\}, \langle j, i\rangle]$. Thus

$$\text{“}X \to \mathsf{S} +\!\!+ \mathsf{S}^*\text{”} := \{S_{ij} +\!\!+ T_{i-1,j+1} | 1 \leq i \leq j \leq n\} \tag{3}$$

consists of all possible decompositions of X into “inside” secondary structure S_{ij} and “outside” structures $T_{i-1,j+1}$. These outside objects correspond to all secondary structures on the union of $[1, i-1]$ and $[j+1, n]$. Similarly $\mathsf{B} +\!\!+ \mathsf{B}^*$ lists all secondary structures with given base pair $\langle i, j\rangle$.

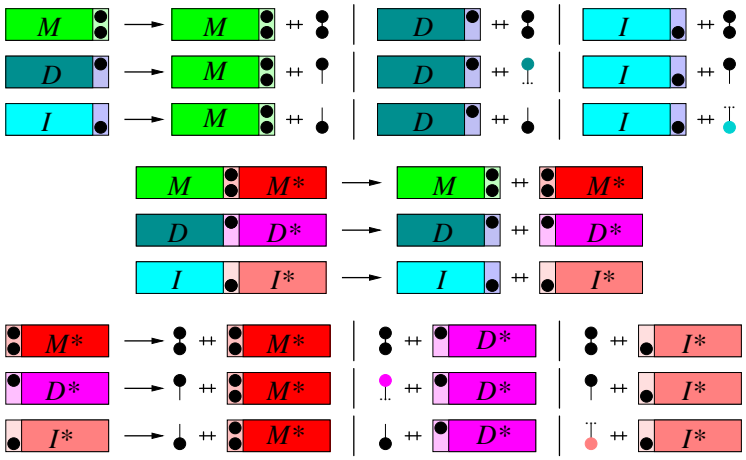


Fig. 1. Derivation of the outside algorithm for a Gotoh-style pairwise sequence alignment. **Top**: graphical notation for the productions. Non-terminals are alignments ending in a (mis)match, insertion, or deletion, terminals are (mismatches), and single base insertions and deletions. For each non-terminal, the interior and the interface is indicated. **Middle:** definition of outside objects by complementing to a full alignment and constraining on the interface. **Bottom:** The grammar derived for the outside elements coincides with the suffix version of Gotoh's algorithm.

DP for Pairwise Sequence Alignments

Pairwise sequence alignment with affine gap costs is solved by Gotoh's well-known algorithm. The corresponding context free grammar has three non-terminals M, D, I, depending on whether the right end of the alignment is a match state, a gap in the first sequence, or a gap in the second sequence. Note that this CFG operates on two rather than a single input string. The productions are of the form

$$\begin{aligned} M &\to M\binom{u}{v} \mid D\binom{u}{v} \mid I\binom{u}{v} \mid \binom{\varepsilon}{\varepsilon} \\ D &\to M\binom{u}{-} \mid D\binom{u}{\cdot} \mid I\binom{u}{-} \\ I &\to M\binom{-}{v} \mid D\binom{-}{v} \mid I\binom{\cdot}{v} \end{aligned} \tag{4}$$

where u and v denote terminal symbols. '$-$' corresponds to gap opening, while '.' denotes the (differently scored, cf. colored terminals in Fig. 1) gap extension.

Outside Objects

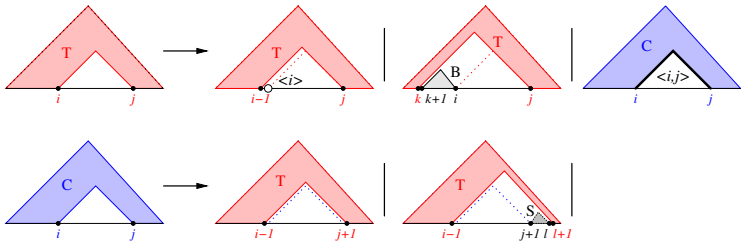
Let us apply the construction of equ.(5) of the main text. In linear grammars, such as the pairwise sequence alignment problem, this is particularly simple. We first note that the "last column", i.e., whether an alignment of prefixes ends in a (mis)match, an insertion, or a deletion forms the interface of the partial solution. Then we see that M^*_{ij} is simply the set of alignments of suffixes, again

with the terminal $\binom{i}{j}$ at its left end. $\mathrm{int}(M^*_{ij})$ is an alignment of the suffixes starting at $i+1$ and $j+1$. The complete situation is summarized in Figure 1. The recursions for the outside objects are readily derived. For instance, there are three decompositions resulting in an M-object: $M_{ij} \mapsto M_{i-1,j-1} \mathbin{+\!\!+} \binom{\bullet}{\bullet}$, $D_{ij} \mapsto M_{i-1,j} \mathbin{+\!\!+} \binom{\bullet}{-}$, and $I_{ij} \mapsto M_{i,j-1} \mathbin{+\!\!+} \binom{-}{\bullet}$. The corresponding outside decompositions are $M^*_{i-1,j-1} \mapsto \binom{\bullet}{\bullet} \mathbin{+\!\!+} M^*_{ij}$, $M^*_{i-1,j} \mapsto \binom{\bullet}{-} \mathbin{+\!\!+} D^*_{ij}$, $M^*_{i,j-1} \mapsto \binom{-}{\bullet} \mathbin{+\!\!+} I^*_{ij}$. Renaming the indices to i and j on the l.h.s. of each decomposition yields $M^*_{i,j} \mapsto \binom{\bullet}{\bullet} M^*_{i+1,j+1} \mid \binom{\bullet}{-} \mathbin{+\!\!+} D^*_{i+1,j} \mid \binom{-}{\bullet} \mathbin{+\!\!+} I^*_{i,j+1}$. Analogous expressions are obtained for D^*_{ij} and I^*_{ij}. As a result we therefore obtain the well-known recursions for Gotoh's algorithm operating on suffixes instead of prefixes. It is worth noting that in this case the inside-style decompositions of the outside objects do not involve the inside objects. This is a general feature of linear grammars, which have only one non-terminal on the r.h.s. of each production.

In the general case, the outside algorithm mixes inside and outside objects. This is the case for instance for RNA folding. The outside objects are structures on the complement of a sequence interval. Since unconstrained structures in our definition have no interface, the sets of positions of S_{ij} and S^*_{ij} are disjoint. Using a notation where the indices denote the extremal nucleotides we have $S^*_{ij} = T_{i-1,j+1}$, where the latter denotes the set structures on the disjoint union of the intervals $[1, i-1]$ and $[j+1, n]$. On the other hand the structures constrained to be enclosed by a base pair have that base pair as their interface. Thus $B^*ij = C_{ij}$, where C_{ij} denotes the set of all structures on $[1, i]\dot\cup[j, n]$ so that (i, j) forms a base pair. Note that C_{ij} corresponds to T_{ij} with the additional constraint that $\langle i, j\rangle$ is a base pair. This definition of the outside objects and the rule for generating the decompositions of the outside objects leads to the following correspondences:

$S_{ij} \mapsto B_{ij}$	yields	$B^*_{ij} \mapsto S^*_{ij}$
$S_{ij} \mapsto \langle i\rangle \mathbin{+\!\!+} S_{i+1,j}$	yields	$S^*_{i+1,j} \mapsto \langle i\rangle \mathbin{+\!\!+} S^*_{ij}$
$S_{ij} \mapsto B_{ik} S_{k+1,j}$	yields	$B^*_{ik} \mapsto S^*_{ij} \mathbin{+\!\!+} S_{k+1,j}$
	and	$S^*_{k+1,j} \mapsto S^*_{ij} \mathbin{+\!\!+} B_{ik-1}$
$B_{ij} \mapsto S_{i+1,j-1} \mathbin{+\!\!+} \langle i, j\rangle$	yields	$S^*_{i+1,j-1} \mapsto B^*_{ij} \mathbin{+\!\!+} \langle i, j\rangle$

Substituting the T and C notation and renaming the indices so that the l.h.s. of the rules always refers to $[1, i]\dot\cup[j, n]$ leads to the following set of concrete decompositions for the outside objects:



(5)

The construction equ.(5) of the main text implies that the outside grammars also make use of inside non-terminals, indicated in black in the diagrams in

equ. (5), whenever there is an inside production that contains more than one non-terminal on its r.h.s. In a more conventional notation we may write equ.(5) as follows:

$$
\begin{aligned}
T_{ij} &\mapsto \langle i \rangle +\!\!+ T_{i-1,j} & \qquad C_{ij} &\mapsto T_{i-1,j+1} \\
T_{ij} &\mapsto T_{k,j} +\!\!+ B_{k+1,i} & \qquad C_{ij} &\mapsto T_{i-1,l+1} +\!\!+ S_{j+1,l} \\
T_{ij} &\mapsto C_{ij} +\!\!+ \langle i, j \rangle
\end{aligned}
$$

It is important to note that $+\!\!+$ operator here as a semantics different from just string concatenation.

Start and Stop Symbols

We have, for brevity of presentation, disregarded the difficulties arising from start and stop symbols. It is always possible to write the grammar with a dedictated start symbol ☺ that never appears on the r.h.s. of a production. Note that ☺ designates a completely unspecified solution, i.e., encodes the complete search space. The corresponding outside object is the empty string ε, referring to an empty set of solutions. The inside production $☺ \to S$ obviously gives rise to the outside production $S^* \to \varepsilon$. Correspondingly, any rule of the form $N \to \varepsilon$ recognizing the empty string must have a corresponding outside production $☺ \to T^*$. In the RNA example above we have written the grammar without an explicity ε symbol. This is possible because parsing also stops at any terminal. Hence we need to deal with all rules of the form $N \to t$. These naturally transform to $☺ \to t +\!\!+ N^*$, thus giving rise to the rules for the start symbol of the outside recursions.

Algorithms in `ADPfusion`

The efficient implementation of dynamic programs for set data types is based on the `ADPfusion`[1] framework. In its inception `ADPfusion` operated on context-free grammars for sequences similar to Algebraic dynamic programming (ADP) by Giegerich et al. [2]. With the extension to single- and multi-dimensional languages [3, 4], `ADPfusion` acquired the possibility to handle different types of input and index spaces.

The choice of `ADPfusion` is based on its agnosticism toward index spaces, input types, as well as the separation of a grammar which gives the structure of the search space and algebras to evaluate candidate structures.

With a basis in stream fusion techniques [5] algorithms developed within this framework have running time performance close to well-optimized `C` code. While we cannot provide comparative figures for the algorithms described in this contribution (since we did not implement them in `C`), the original work [1] provides performance characteristics for several real-world algorithms.

Implementation of Set-based DP Algorithms

The implementation of set-based dynamic programming algorithms comprises two large steps, of which typically only the second needs to be performed by the developer of DP algorithms. The first step is the implementation of novel index structures, as well as terminal and non-terminal symbols. The second step is the implementation of an actual algorithm.

Novel Index Structures, Terminals, and Non-Terminals

Index structures hold the necessary information to identify each substructure within the recursive decomposition of a problem. A context-free grammar on sequences will make use of a *subword*, a pair of indices (i, j) which give the start and end index of the substring currently being evaluated.

An index structure for sets with an interface contains the bit-vector of active set elements, as well as an identifier for the interface.

Terminal symbols extract, given an index, "atomic" elements from the input. For sequences, these will be individual characters. For the sets used in this work, they constitute nodes in the graph or active elements in the set.

Non-terminal symbols finally have two aspects. On the one hand, in terms of a formal grammar, they are simply symbols that are being replaced when following production rules. In terms of efficient dynamic programming implementation however, they hold the results of previous calculations on smaller indices and thereby facilitate a more efficient solution of the problem via memoization.

The Hamiltonian Path Problem

Following the notation from above, the dynamic programming decomposition of the shortest path from node i to node j, traversing all nodes in A is written as:

$$[i, A, j] \mapsto [i, A, k] +\!\!+ \langle k, j \rangle. \tag{6}$$

The translation to `ADPfusion` follows almost mechanistically, though we shall forego the pseudo-code here. We need a non-terminal N indexed by the index structure $[i, A, j]$. In addition, we need two terminal symbols: d which splits of the current head of the traversed path, and p which "peaks" at the second to last element. The terminal symbol p is of purely syntactic nature, given that non-terminals act as score memoizers.

The function f evaluates the score yielded by combining d with (p, N), or adding a new edge (p, d) to the existing path.

$$N_{[i,A,j]} \mapsto_f d_{[k,\emptyset,j]} +\!\!+ (p_{[i,A\setminus k,k]}, N_{[i,A\setminus k,k]}) \tag{7}$$

Finally, in `ADPfusion` we prefer to keep indices out of sight, which means that a suitable encoding of the production rule is:

$$N \mapsto_f d +\!\!+ (p, N). \tag{8}$$

The resulting grammar is a linear language over a more exotic index structure than usual, namely an unordered set with an interface, but the programmer need not be overly concerned as most of this is hidden in the high-level interface. Indeed, the programmer has the advantage that such an index structure is available in our framework. Though even if it were not, the required code is in the order of a couple hundred lines of code, and comparatively simple as each individual concern of index structure, terminal symbol, or memoization logic can be handled and tested separately.

References

1. 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, ACM (2012) 215–226
2. Giegerich, R., Meyer, C.: Algebraic dynamic programming. In Kirchner, H., Ringeissen, C., eds.: Algebraic Methodology And Software Technology. Volume 2422 of Lect. Notes Comp. Sci. Springer, Berlin, Heidelberg (2002) 349–364
3. Höner zu Siederdissen, C., Hofacker, I.L., Stadler, P.F.: How to Multiply Dynamic Programming Algorithms. In: Brazilian Symposium on Bioinformatics (BSB 2013). Volume 8213 of Lect. Notes Bioinf., Springer, Heidelberg (2013) 82–93
4. Höner zu Siederdissen, C., Hofacker, I.L., Stadler, P.F.: Product Grammars for Alignment and Folding. IEEE/ACM Trans. Comp. Biol. Bioinf. **99** (2014)
5. Coutts, D., Leshchinskiy, R., Stewart, D.: Stream Fusion: From Lists to Streams to Nothing at All. In: Proceedings of the 12th ACM SIGPLAN international conference on Functional programming. ICFP'07, ACM (2007) 315–326