Dynamic Programming for Set Data Types

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

Abstract. We present an efficient generalization of algebraic dynamic programming (ADP) to unordered data types and a formalism for the automated derivation of outside grammars from their inside progenitors. These theoretical contributions are illustrated by ADP-style algorithms for shortest Hamiltonian path problems. These arise naturally when asking whether the evolutionary history of an ancient gene cluster can be explained by a series of local tandem duplications. Our framework makes it easy to compute Maximum accuracy solutions, which in turn require the computation of the probabilities of individual edges in the ensemble of Hamiltonian paths. The expansion of the Hox gene clusters is investigated as a show-case application. For implementation details see
`http://www.bioinf.uni-leipzig.de/Software/setgram/`

Key words: formal grammar, outside grammar, dynamic programming, Haskell, Hamiltonian path problems, tandem duplications, Hox clusters

1 Introduction

Dynamic Programming (DP) over rich index sets provides solutions of a surprising number of problems in discrete mathematics. Even for NP-hard problems such as the Travelling Salesman Problem (TSP) exact solutions can be obtained for moderate size problems of practical interest. The corresponding algorithms, however, are usually specialized and use specific properties of the problem in an *ad hoc* manner that does not generalize particularly well.

Algebraic dynamic programming (ADP) [1] defines a high-level descriptive domain-specific language for dynamic programs over sequence data. The ADP framework allows extremely fast development even of quite complex algorithms by rigorously separating the traversal of the state space (by means of context free grammars), scoring (in terms of suitable algebras), and selection of desired solutions. The use of CFGs to specify the state space is a particular strength of ADP since it allows the user to avoid indices and control structures altogether, thereby bypassing many of the pitfalls (and bugs) of usual implementations. Newer dialects of ADP [2, 3] provide implementations with a running time performance close to what can be achieved by extensively hand-optimized versions, while still preserving most of the succinctness and high-level benefits of the original ADP language. The key goal is to develop a framework that makes it easy to implement complex dynamic programs by combining small, simple components. A first step in this direction was the introduction of grammar products [4], which greatly simplifies the specification of algorithms for sequence alignments and related dynamic programming tasks that take multiple strings as input. The second and third steps are introduced in this work: a formal system for dynamic programming over unordered data types, together with the mechanistic derivation of Outside algorithms; both implemented in `ADPfusion` [2].

Sequence data is not the only type of data for which grammar-like dynamic programs are of interest. Inverse coupled rewrite systems (ICOREs) [5] allow the user to develop algorithms over both, sequence and tree-like data. While no implementation for these rewrite systems is available yet, they already simplify the initial development of algorithms. This is important in particular for tree-like data. Their non-sequential nature considerably complicates these algorithms. The grammar underlying the alignment of ncRNA family models with `CMCompare` [6], which simultaneously recurses over two trees, may serve as an example for the practical complications. There are compelling reasons to use DP approaches in particular when more information than just a single optimal solution is of interest. DP over sequences and trees readily allows the enumeration of all optimal solutions, and it offers generic ways to systematically investigate suboptimal solutions and to compute the probabilities of certain sub-solutions. Classified dynamic programming [7], furthermore, enables the simultaneous calculation of solutions with different class features via the evaluation algebra instead of constructing different grammars for each class. Two well-known examples for DP over sequences in computational biology for which these features are extensively used in practise are pairwise sequence alignment and RNA folding. Due to the tight space limits we relegate them to the Electronic Supplement.

A quite different classical example of DP is the Travelling Salesman Problem (TSP). It is easily stated as follows: given a set X of cities and a matrix $d : X \times X \to \mathbb{R}_+$ of (not necessarily symmetric) distances between them, one looks for the tour (permutation) π on X that minimizes the tour length $f(\pi) := d_{\pi(n),\pi(1)} + \sum_{i=1}^{n-1} d_{\pi(i),\pi(i+1)}$. W.l.o.g., we may set $X = \{1, \ldots, n\}$ and anchor the starting point of a tour at $\pi(1) = 1$. The well-known (exponential-time) DP solution for the TSP [8, 9] operates on "sets with an interface" $[A, i]$ representing the set of all tours starting in $1 \in A$, then visiting all other cities in A exactly once and ending in $i \in A$. The length of the shortest path of this type is denoted by $f([A, i])$. For an optimal tour we have $f([X \setminus \{i\}, i]) + f(\langle i, 1\rangle) \to \min$, where $f(\langle i, 1\rangle) = d_{1,i}$ is the length of the edge from i to 1. The $f([A, i])$ satisfy the recursions

$$f([A, i]) = \min_{j \in A} f([A \setminus \{i\}, j]) + f(\langle j, i\rangle) \tag{1}$$

since the shortest path through A to i must consist of a shortest path through A ending in some $j \in A$ and a final step from j to i. The fundamental question that we will address in this contribution is whether we can rephrase this and similar DP algorithms also in an ADP like manner. In other words: how can we separate state space traversal and evaluation, even though we do not have a grammar at hand (because we do not even operate on strings or ordered trees)?

2 ADP over Set-Like Data Types

Generalized Decompositions. The key observation is that we have to generalize the notion of *parsing a string* to much more general ways of traversing the state space. This interpretation of "productions" makes perfect sense for the paths in the TSP solution. The operator $+\!\!+$ provides the decomposition of the set A into A' as well as a terminal e denoting the newest edge added to A' to construct A.

$$\text{“}A \to A' +\!\!+\, e\text{”} := \{[A,i] \mapsto [A \setminus \{i\}, j] +\!\!+ \langle j,i\rangle | A \subseteq X,\, j \in A\} \tag{2}$$

The path variables $[A,i]$ highlight a second important ingredient of the formalism. Each object $[A,i]$ consists of an interior part $\mathrm{int}([A,i]) = A \setminus \{i,1\}$ and the interface $\partial A = \{1,i\}$. The latter consists of the vertices that need to be known explicitly for the evaluation: they will appear explicitly in the evaluation algebra. For fixed A in the production (2), e.g., we have to consider all $j \in A \setminus \{1\}$ as possible endpoints of the paths.

The distinction between interior and interface of each object $A := [\mathrm{int}(A), \partial A]$ allows a more principled way to constructing concrete decompositions:

$$[\mathrm{int}(A), \partial A] \mapsto \underset{i}{+\!\!+} [\mathrm{int}(A_i), \partial A_i] \tag{3}$$

with the following properties:

(C1) $\bigcup_i A_i = A$, i.e., the parts of A form a covering of A.
(C2) $\mathrm{int}(A_i) \cap \mathrm{int}(A_j) \neq \emptyset$ implies $i = j$, i.e., the interiors of the parts are disjoint.
(C3) $\mathrm{int}(A_i) \subseteq \mathrm{int}(A)$, i.e., the interiors behave like isotonic functions.

The intuition behind axiom (C1) is that any decomposition of an object must eventually evaluate all parts. Condition (C2) and (C3) implies that the interiors of the parts can be evaluated independently. To allow meaningful evaluation algebras in the ADP sense we require that concatenation is associative. It may be tempting to think of ∂ and int in terms of generalized topological functions, i.e., as boundary and interior operators. This may not need to be the case in full generality, since we may have situations where A is not just a set.

A terminal is an object for which there is no further concrete decomposition. In the TSP examples, the terminals are on the one hand the edges $\langle j,i\rangle$ that appear explicitly in the decompositions as well as the path $\langle 1\rangle := [\{1\}, 1]$ of length 0 that appears implicitly as the base case of the concrete decompositions. The boundary ∂A is not necessarily just an unstructured set. For the asymmetric TSP, e.g., start and end point of a path $[A,j]$ are distinguished.

So far our discussion has been focussed on the decomposition of inputs in the terms of a grammar. The goal to optimize with an objective function in DP has only entered in passing, as in the TSP example in equ. (1). For DP to work, however, more is required. The grammar performs the decomposition of each sub-input into its constituent elements, or terminal and non-terminal symbols. Each of the different decompositions is then evaluated using an evaluation algebra that defines how $f(A)$ depends on the evaluation $f(A_i)$ of the fragments. In general there are multiple alternative decompositions of A. For the TSP for instance, we have to consider $A \to A \setminus \{i\}$ for all $i \in A$. It also is the job of the score algebra to combine the scores over these alternatives. To minimize f, scores are added over constituents and minimized over alternatives. To compute partition functions they are multiplied over constituents and added up over alternatives. Finally *Bellman's principle* [8] stipulates that decomposition and scoring play together in such a way that optimal solutions are always obtained by composing optimal solutions of smaller problems. We can implement algorithms specified in this formal system very efficiently (both in terms of programming effort and actual running times) using an extension to `ADPfusion`. Details are given in the Electronic Supplement.

Deriving Outside Algorithms. A key advantage of DP algorithms is the generic possibility to compute solutions with constraints, such as alignments that contain a given alignment edge. The basic idea behind this possibility is the combination of an "inside" with an "outside" solution, i.e., a pair of complementary partial solutions. Well known examples are pairwise sequence alignments or RNA folding. In the first example, pairwise alignments of prefixes are the objects of the forward (or "inside") recursion, while suffix-alignments are required as "outside" objects. In the RNA case, this is even more transparent, since "inside" runs over secondary structures on intervals, while the outside algorithm recurses over the complements of the intervals, again proceeding from smaller to larger outside objects. A good example is McCaskill's algorithm for computing the base pairing probabilities in the ensemble of all secondary structures formed by an input RNA molecule [10]. The construction of the outside traversal is a difficulty in ADP that has not been fully solved. For CFGs, thesis [11] shows that a grammar for the outside objects can be derived by doubling the input string and re-interpreting the region outside of interval $[i, j]$ as the interval $[j+1, i'-1]$ where i' is the equivalent position i in the 2nd copy of the input.

To make use of the full potential of dynamic programming it would be highly desirable to construct suitable outside traversals automatically from a given inside traversal. In the remainder of this section we discuss some of the general principles underlying the relationship of inside and outside recursion on a general level. The key observation is that the distinction of inside and outside comes from a generic way of splitting solutions so that

$$[\mathrm{int}(X), \partial A] \to [\mathrm{int}(A), \partial A] +\!\!+ [\mathrm{int}(A^*), \partial^* A^*] \tag{4}$$

corresponds to the set of all solutions that are constrained to $\partial A = \partial^* A^*$, i.e., that contain the particular feature specified by ∂A. Set-like objects have a straightforward explicit definition of their outside objects: $\mathrm{int}(A^*) := X \setminus (\mathrm{int}(A) \cup \partial A)$. The notation $\partial^* A^*$ emphasizes that in the case of structured interfaces corresponding inside and outside objects must consist of the same terminals, but possibly in different orderings. In the Electronic Supplement we illustrate this construction for RNA folding and pairwise alignments.

The straightforward definition of outside objects suggests that it should also be possible to construct inside-style productions for these outside objects in a generic, rule-based manner. It turns out that the solution to this long-standing problem in DP becomes surprisingly simple as soon as we allow ourselves to "parse" also data structures that are not strings or trees. With each concrete inside decomposition $A \mapsto (+\!\!+_i A_i) +\!\!+ (+\!\!+_j \langle t_j \rangle)$, where the A_i are non-terminals and the $\langle t_j \rangle$ are terminals, we associate

$$A_k^* \mapsto A^* +\!\!+ \left(\underset{i \neq k}{+\!\!+} A_i \right) +\!\!+ \left(\underset{j}{+\!\!+} \langle t_j \rangle \right) \tag{5}$$

For examples and a discussion of start non-terminals and empty terminals we refer to the Electronic Supplement. The situation is even simpler for the TSP: we have $[A, (1, i)]^* = [(X \setminus A) \setminus \{1, i\}, (i, 1)]$. In particular, $[A, (1, i)] +\!\!+ [A, (1, i)]^*$ corresponds to the set of all Hamiltonian paths that run from 1 to i through A and then from i back to 1 through $X \setminus A$. The same idea applies to other Hamiltonian path problems.

3 Application to Gene Cluster Histories

Local duplication of DNA segments via unequal crossover is the most plausible mechanism for the emergence and expansions of local clusters of evolutionary related genes. It remains hard and often impossible to disentangle the history of ancient gene clusters in detail even though polynomial-time algorithms exist to reconstruct duplication trees from pairwise evolutionary distance data [12]. The reason is the limited amount of phylogenetic information in a single gene. The situation is often aggravated by the extreme time scales leading to a decay of the phylogenetic signal so that only a few, very well-conserved sequence domains can be compared. A large number of trees then fits the data almost equally well. A meaningful analysis thus must resort to some form of summary that is less detailed than a duplication tree. In the absence of genome rearrangements, and if duplication events are restricted to copying single genes to adjacent positions, we expect phylogenetic distance to vary monotonically with genomic distance. A shortest Hamiltonian path through the phylogenetic distance matrix therefore should conform to the linear arrangement of the genes on the genome. The same high noise level that suggests to avoid duplication trees makes us distrust a single shortest path. Rather, we would like to obtain information on the ensemble of all Hamiltonian paths.

The shortest Hamiltonian path problem, well known to be NP-complete, is closely related to the TSP, and admits a similar dynamic programming solution [8, 9]. We provide here an efficient implementation in our ADP-style framework. Denote by $[i, A, j]$ with $i, j \in A$ the set of all paths starting in i, ending in j, and passing through all other vertices of A in between. It will be convenient to fix the start and end points p and q of the paths, i.e., the search space is $X_{pq} := [p, X, q]$. With fixed p and q we need not treat the ends p and q as interface points, i.e., we can write $[A, j]$ for the path sets, where $p \in A$ and $q \notin A$ for all A. As for the TSP we have $[A, j] \mapsto [A \setminus \{j\}, k] +\!\!+ \langle k, j \rangle$ and $[A, j] +\!\!+ [A, j]^* = X_{pq}$ from which we obtain the outside objects as the path sets $[A, j]^* = [j, X \setminus A]$ with endpoint $q \in$

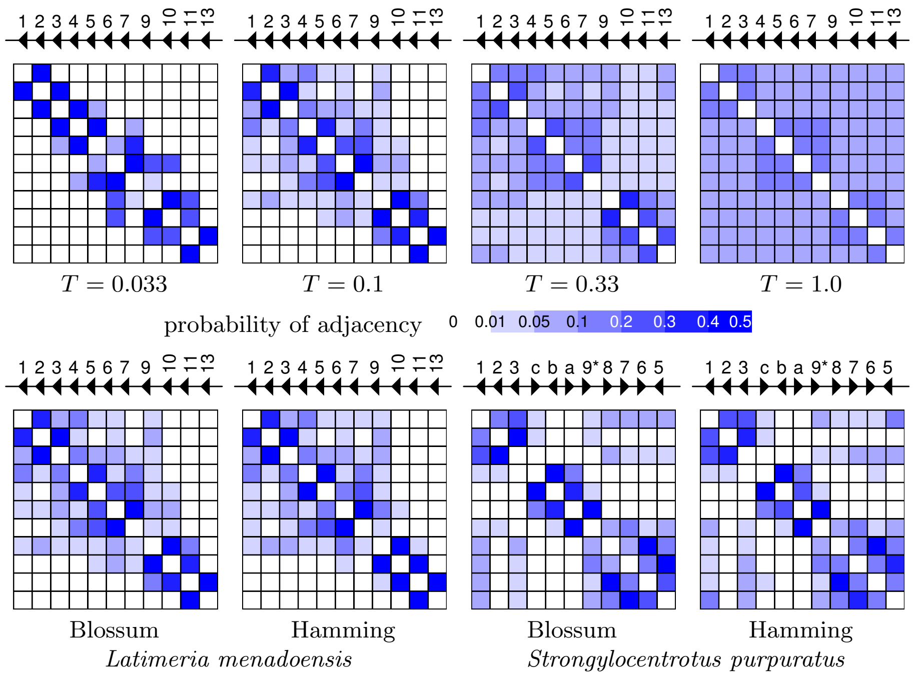


Fig. 1. Posterior probabilities of adjacencies of Hox genes along shortest Hamiltonian paths w.r.t. to phylogenetic distance. **Top:** effect of the temperature parameter T for distances between *Latimeria menadoensis* homeobox sequences. **Below:** Comparison of adjacencies for two different metrics (Hamming distance, and BLOSSUM-45 derived dissimilarities) in *L. menadoensis* (left) and *S. purpuratus*. $T = 0.1$ to emphasize the structure of the ambiguities. Note the adjacencies between the block of anterior Hox genes (1,2,3) and the middle group genes (5,6,7,8), reflecting the break-up and translocation of anterior genes to a genomic location before the posterior genes.

$X \setminus A$. The corresponding concrete decompositions are $[j, B] \mapsto \langle j, k\rangle +\!\!+ [k, B \setminus \{j\}]$ for $k \in B \setminus \{j\}$. Partition functions Z over Hamiltonian paths are computed using $Z(A +\!\!+ B) = Z(A)Z(B)$, $Z([\{p\}, p]) = Z([q, \{q\}) = 1$, and $Z(\langle i, j\rangle) = \exp(-d_{ij}/kT)$ is the Boltzmann factor of the distance between two vertices, i.e., of the terminals. Our generalized ADP framework takes care of computing all $Z([p, A, i]) = Z([A, i])$ and $Z([k, B, q]) = Z([k, B])$. The *a posteriori* probability of observing an adjacency $i \sim j$ in path with fixed endpoints p and q is
$P(i \sim j|p, q) = Z([p, A, i])Z(\langle i, j\rangle)Z([j, X \setminus (A \cup \{i\}), q])/Z(X_{pq})$.
As usual, this is simply the ratio of restricted and unrestricted partiton functions. Summing over the possible end points of the paths yields

$$P(i \sim j) = \frac{1}{Z} \sum_{p,q} Z([p, A, i])Z(\langle i, j\rangle)Z([j, X \setminus (A \cup \{i\}), q]) \,, \qquad (6)$$

where $Z = \sum_{p,q} Z(X_{pq})$ is the partition function over all Hamiltonian paths. $Z(X_{p,q})/Z$ is the probability that the path has p and q as its endpoints.

Hox genes are ancient regulators originating from a single Hox gene in the metazoan ancestor. Over the course of animal evolution the Hox cluster gradually expanded to 14 genes in the vertebrate ancestor. Timing and positioning of Hox gene expression along the body axis of an embryo is co-linear with the genomic arrangement in most species. Only the 60 amino acids of the so-called homeodomain can be reliably compared at the extreme evolutionary distances involved in the evolution of the Hox system. We use either the Hamming distance, measuring the number of different amino-acids, or the transformation $d_{ab} = s(a,a) + s(b,b) - 2s(a,b)$ of the BLOSSUM45 similarity matrix to quantify the evolutionary distances of the homeodomain sequences. Setting k to the average pairwise genetic distance ensures that T quantifies the expected noise level as a fraction of the phylogenetic signal. For $T \to 0$ we focus on the (co)optimal paths only, while $T \to \infty$ leads to a uniform distribution of adjacencies.

We analyzed here the Hox A cluster of *Latimeria menadoensis* (famous as a particularly slowly evolving "living fossil"), which has sufferered the fewest gene losses among vertebrates. The Hox cluster of the sea urchin *Strongylocentrus purpuratus*, in contrast, has undergone fairly recent rearrangements of its gene order [13]. Fig. 1 shows the posterior probabilities of adjacencies. Both examples reflect the well-known clustering into anterior (Hox1-4), middle group genes (Hox4-8), and posterior ones (Hox9-13). The shortest Hamiltonian paths in *L. menadoensis* connect the Hox genes in their genomic order. In the sea urchin, however, we see adjacencies connecting the anterior subcluster (Hox1-3) with the genomic end of the cluster, i.e., the middle group genes (Hox8-Hox5).

4 Discussion

We have taken here the first step towards extending algebraic dynamic programming (ADP) beyond the realm of string-like data structures. Our focus is an efficient, yet notationally friendly way to treat DP on unordered sets. Our extension of ADP builds on the same foundation (namely `ADPfusion` [2]) as our grammar product formalism [14, 4]. Our formalism explicitly redefines the rules of parsing to match the natural subdivisions of the data type in question. In the case of sets, these are bipartitions and the splitting of individual elements, rather than the subdivision of an interval or the removal of a boundary element that are at the heart of string grammars. As a showcase example we considered in detail the shortest Hamiltonian path problem, which arises e.g. in the context of the evolution of ancient gene clusters. In this context we are interested in particular in probabilities and hence in restricted partition functions. An ADP-style implementation and a principled approach to constructing outside algorithms is of particular practical relevance here.

Our current framework still lacks generality and completeness in several respects. The theoretical foundations for the automated calculation of outside grammars for, basically, traversals of arbitrary data types is our most immediate concern. In this context McBride's notion of a derivative operator acting on data types [15] is highly relevant, even though it does not seem to be directly applicable. Even more generally, it might be possible to generate decomposition schemes, i.e. "grammar rules", from an analysis of the data structure itself.

Acknowledgements. This work was funded, in part, by the Austrian FWF, project "SFB F43 RNA regulation of the transcriptome", the Templeton Foundation, grant # 24332 "Origins and Evolution of Regulation in Biological Systems", and the DFG project "MI439/14-1".

References

1. 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
2. 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
3. Sauthoff, G., Janssen, S., Giegerich, R.: Bellman's GAP - A Declarative Language for Dynamic Programming. In: Proceedings of the 13th international ACM SIGPLAN symposium on Principles and practices of declarative programming. PPDP'11, ACM (2011) 29–40
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. Giegerich, R., Touzet, H.: Modeling Dynamic Programming Problems over Sequences and Trees with Inverse Coupled Rewrite Systems. Algorithms (2014) 62–144
6. Höner zu Siederdissen, C., Hofacker, I.L.: Discriminatory power of RNA family models. Bioinformatics **26**(18) (2010) 453–459
7. Voß, B., Giegerich, R., Rehmsmeier, M.: Complete probabilistic analysis of RNA shapes. BMC biology **4**(1) (2006) 5
8. Bellman, R.: Dynamic programming treatment of the travelling salesman problem. J. ACM **9** (1962) 61–63
9. Held, M., Karp, R.M.: A dynamic programming approach to sequencing problems. J. SIAM **10** (1962) 196–201
10. McCaskill, J.S.: The equilibrium partition function and base pair binding probabilities for RNA secondary structure. Biopolymers **29** (1990) 1105–1119
11. Janssen, S.: Kisses, ambivalent models and more: Contributions to the analysis of RNA secondary structure. PhD thesis, Univ. Bielefeld (2014)
12. Elemento, O., Gascuel, O.: An efficient and accurate distance based algorithm to reconstruct tandem duplication trees. Bioinformatics **8 Suppl. 2** (2002) S92S99
13. Cameron, R.A., Rowen, L., Nesbitt, R., Bloom, S., Rast, J.P., Berney, K., Arenas-Mena, C., Martinez, P., Lucas, S., Richardson, P.M., Davidson, E.H., Peterson, K.J., Hood, L.: Unusual gene order and organization of the sea urchin Hox cluster. J Exp Zoolog B Mol Dev Evol **306** (2006) 45–58
14. 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
15. McBride, C.: Clowns to the left of me, jokers to the right (pearl): dissecting data structures. In: ACM SIGPLAN Notices. Volume 43., ACM (2008) 287–295

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,

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

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

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

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 $\langle i,j\rangle$.

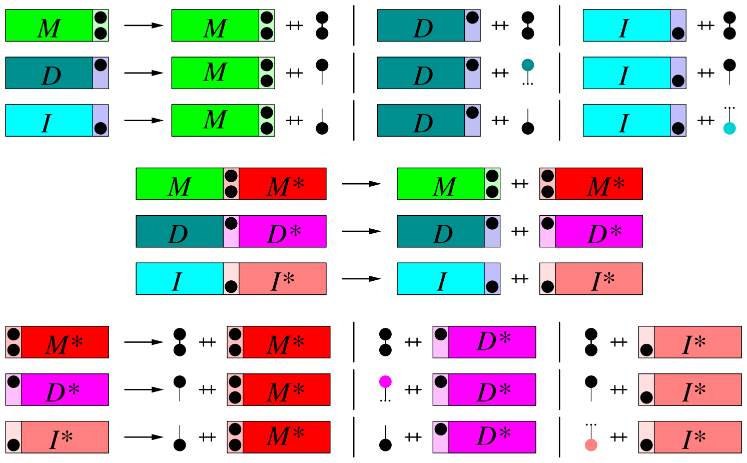


Fig. 1. Derivation of the outside algorithm for a Gotoh-style pairwise sequence alignment. **Top**: graphical notation for the productions. Non-terminals are alignments ending in a (mis)match, insertion, or deletion, terminals are (mismatches), and single base insertions and deletions. For each non-terminal, 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

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

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

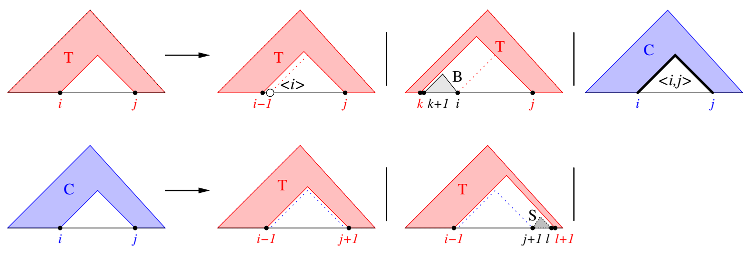
Outside Objects

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:

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

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