Available online at www.sciencedirect.com
ScienceDirect
Information and Computation 204 (2006) 1325–1345
Information and Computation
www.elsevier.com/locate/ic

Table design in dynamic programming

Peter Steffen *, Robert Giegerich

Faculty of Technology, Bielefeld University, Postfach 10 01 31, 33501 Bielefeld, Germany

Received 9 August 2004; revised 17 January 2006
Available online 8 August 2006

Abstract

Dynamic Programming solves combinatorial optimization problems by recursive decomposition and tabulation of intermediate results. The first step in the design of a dynamic programming algorithm is to decide on the set of tables that will hold optimal solutions to subproblems. This step predetermines the shape of the dynamic programming recurrences as well as the asymptotic efficiency of the algorithm in time and space. We study dynamic programming in a formal framework where design of tables and problem decomposition can be done independently. Our main result shows that choosing a good table design for a given decomposition is an NP-complete problem. A heuristic or approximate approach is therefore needed to automate good table design. We report on a strategy that combines user annotation and a brute force algorithm, which is shown to perform well in a large application.

Keywords: Dynamic programming; Programming methodology; NP completeness; Space-time trade-off

1. Introduction

1.1. Motivation

Dynamic Programming is an elementary and widely used programming technique. Its roots reach back half a century, when the method and its applications were explored in the work of Bellman

* Corresponding author. Fax: +49 521 106 6411.
E-mail addresses: psteffen@techfak.uni-bielefeld.de (P. Steffen), robert@techfak.uni-bielefeld.de (R. Giegerich).

0890-5401/$ - see front matter © 2006 Elsevier Inc. All rights reserved.
doi:10.1016/j.ic.2006.02.006

and others [4,5]. Today, introductory textbooks on algorithms usually contain a section devoted to dynamic programming [2,8,9,14,20,26], where simple problems like matrix chain multiplication, longest common subsequence, polygon triangulation, string comparison, neatly printing of paragraphs, planning a company party, or construction of optimal binary search trees are commonly used for the exposition. This programming technique is mainly taught by example. Once designed, all dynamic programming algorithms look kind of similar: they are cast in terms of recurrences between table entries that store solutions to intermediate problems, from which the overall solution is constructed in a more or less sophisticated case analysis. The first design step is normally a decision about the tables that are required. Their choice is not difficult with problems that require a single or a very small number of tables. However, this simplicity is quickly lost when turning to more sophisticated problems.

In biological sequence analysis, our specific field of application, dynamic programming algorithms are used on a great variety of problems, such as protein homology search, gene structure prediction, motif search, analysis of repetitive genomic elements, RNA secondary structure prediction, or interpretation of data from mass spectrometry [14,11,3]. The higher sophistication of these problems is reflected in a large number of recurrences—sometimes filling several pages—using more complicated case distinctions, numerous tables and elaborate scoring schemes. Computational efficiency is important, both in time and space, as genomic data tend to be very large.

As a consequence, among experts in the field the design of successful dynamic programming recurrences used to be considered a matter of experience, talent, and luck. Design as well as programming errors are detected long after programs go into production use, and occasionally it takes an exceptionally long time until a simple (in hindsight!) but important improvement is discovered.

An *algebraic* style of *dynamic programming* (ADP) has recently been advocated [12,13], designed to make dynamic programming algorithm development for biosequence analysis more productive and the resulting implementations more reliable. Although originally developed for biosequence analysis, the ADP approach pertains to all dynamic programming problems that have sequential input and whose subproblems are related to contiguous subsequences of the input. This includes all the examples, both from text books and biosequence analysis, mentioned above. ADP allows us to formulate dynamic programming algorithms on a more convenient level of abstraction, based on algebras and tree grammars. It cleanly separates (conceptually) the traversal of the search space from the actual evaluation of solutions, and it also separates (conceptually) efficiency concerns from the logic of the algorithm.

In the ADP approach the problem decomposition is defined using a tree grammar. Therefore, the design of a dynamic programming algorithm can be done independent of a particular choice of tables. Some (but not all) of the nonterminal symbols in the grammar must eventually be mapped to dynamic programming tables—their choice is what we call the table design problem. A particular choice is called a table configuration. Given the grammar and the table configuration, we can automatically generate dynamic programming recurrences. Thus, the dynamic programmer is liberated from error-prone subscript fiddling. This is the state of development reported in [13].

Here, we study the problem of automating not only the derivation of recurrences, but also table design. A *good* table configuration is crucial for efficiency—it makes the difference between exponential and polynomial runtime of alternative implementations of the same algorithm, and, if the runtime is polynomial, the table configuration determines the polynomial degree and the space requirements. A configuration that minimizes both in a precise sense defined later (Definition 10) is called *optimal*.

1.2. Main results

Finding good and optimal table configurations is the problem studied in this article. Previously, this has been the programmer's responsibility—explicitly so when using the ADP approach, or implicitly otherwise. A most ambitious goal would be to completely automate this step, virtually freeing the program designer from efficiency concerns. However, our main result here shows that a complete automation of table design is computationally infeasible: both the choice of a good table configuration and the choice of an optimal configuration are NP-complete problems. Consequently, we develop a pragmatic approach that tries to reduce the problem size by various means to the point where the problem can be solved to optimality.

1.3. Related work

Dynamic programming problems as addressed within the ADP framework can be seen as a special case of the extensively studied algebraic path problems. See [24] for a review, and the recent book by Pachter and Sturmfels [22], which summarizes applications of the algebraic view to problems in bioinformatics. Algebraic path problems are optimization problems over graphs defined via a semiring structure $(S, \oplus, \odot)$, where S is the domain of scores, $\oplus$ the objective function (such as minimization) applied when joining paths, and $\odot$ is the function that combines scores when extending paths. ADP is somewhat richer semantically, as every function in an evaluation algebra is a separate instance of the $\odot$ operation, which must satisfy the semiring axioms, and may operate on several arcs simultaneously. As a consequence, the optimal "path" is actually a tree. In the graph problem framework, the underlying graph structure is usually given explicitly as input. In the ADP approach, the input is always a sequence, and the graph structure remains implicit. For each problem instance, data dependencies (paths) are determined according to the tree grammar. Many mathematical observations from graph problems carry over to the ADP approach, which can be seen as extending algebraic graph problems towards easier programming in specialized, but also more sophisticated application domains. In the context of algebraic path problems, dynamic programming is one of many possible solution strategies, and the problem of minimizing the number of dynamic programming tables, our topic here, is not explicitly addressed.

The approach to represent dynamic programming problems as graphs is also followed by Bodlaender and Telle [6]. They examine to systematically derive space-efficient variants of dynamic programming algorithms that not only calculate an optimal value, but also calculate the corresponding candidates. They demonstrate the method on two dynamic programming problems on graphs and give some ideas for generalization. The problems studied there are all based on a single recurrence with only a single table type. It is not clear whether these techniques can be applied to more general dynamic programming problems as addressed within the ADP framework.

Memory requirements in dynamic programming is a recurring theme in the literature. One common general technique is the reuse of memory for entries that are no longer needed during a calculation [14]. Applying this technique is often easy in dynamic programming problems where only the optimal result is to be computed, but becomes difficult, when also the candidates responsible for the optimal solutions need to be constructed in a backtrace phase [6]. Several improvements exist for available dynamic programming algorithms, most notably the sequence alignment problem [15,21], which can be improved from $O(n^2)$ to $O(n)$ space requirements. Most often, these improvements

rely on deeper knowledge about the problem domain, and can not be applied to general dynamic programming problems.

Some related work is also done in the program transformation community. In [19], Liu and Stoller describe a method to automatically transform recursive programs into dynamic programming programs that achieve optimal time and space requirements. The method is based on a static analysis of the data dependencies, which allows to store only those solutions to subproblems that would otherwise be calculated more than once. Their technique is capable of generating space efficient versions of many standard dynamic programming problems. Our method is different in the sense that we allow to recalculate subproblems in order to save space, as long the overall runtime of the algorithm stays asymptotically optimal.

1.4. Basic terminology

Alphabets. An *alphabet* $\mathcal{A}$ is a finite set of symbols. Sequences of symbols are called strings. ε denotes the empty string, $\mathcal{A}^1 = \mathcal{A}$, $\mathcal{A}^{n+1} = \{ax | a \in \mathcal{A}, x \in \mathcal{A}^n\}$, $\mathcal{A}^+ = \bigcup_{n \geqslant 1} \mathcal{A}^n$, $\mathcal{A}^* = \mathcal{A}^+ \cup \{\varepsilon\}$. $|a_1 \cdots a_n| = n$ for $n \geqslant 0$.

Signatures and algebras. A *signature* Σ over some alphabet $\mathcal{A}$ consists of a sort symbol S together with a family of operators. Each operator o has a fixed arity $o : s_1 \cdots s_{k_o} \to S$, where each s_i is either S or $\mathcal{A}$. A Σ-*algebra* $\mathcal{I}$ over $\mathcal{A}$, also called an interpretation, is a set $\mathcal{S}_\mathcal{I}$ of values together with a function $o_\mathcal{I}$ for each operator o. Each $o_\mathcal{I}$ has type $o_\mathcal{I} : (s_1)_\mathcal{I} \cdots (s_{k_o})_\mathcal{I} \to \mathcal{S}_\mathcal{I}$ where $\mathcal{A}_\mathcal{I} = \mathcal{A}$.

A *term algebra* T_Σ arises by interpreting the operators in Σ as *constructors*, building bigger terms from smaller ones. When variables from a set V can take the place of arguments to constructors, we speak of a term algebra with variables, $T_\Sigma(V)$, with $V \subset T_\Sigma(V)$. By convention, operator names are capitalized in the term algebra.

Trees and tree patterns. Terms will be viewed as rooted, ordered, node-labeled trees in the obvious way. All inner nodes carry (non-nullary) operators from Σ, while leaf nodes carry nullary operators from Σ or symbols from $\mathcal{A}$. A term/tree with variables is called a *tree pattern.* A tree containing a designated occurrence of a subtree t is denoted $C[\ldots t \ldots]$.

A *tree language* over Σ is a subset of T_Σ. Tree languages are described by tree grammars, which can be defined in analogy to the Chomsky hierarchy of string grammars. In tree grammars, nonterminal symbols and variables coincide. Here, we use regular tree grammars, originally studied in [7]. Our further specialization lies solely in the distinguished role of $\mathcal{A}$.

Definition 1 (*Tree grammar over* Σ *and* $\mathcal{A}$).
A regular tree grammar $\mathcal{G} = (V, Z, P)$ over Σ and $\mathcal{A}$ is given by

- a set V of nonterminal symbols,
- a designated nonterminal symbol Z, called the axiom, and
- a set P of productions of the form $v \to t$, where $v \in V$ and $t \in T_\Sigma(V)$. $v \to t_1 | \cdots | t_n$ shall denote the short form for $v \to t_1, \ldots, v \to t_n$.

The derivation relation for tree grammars is $\Rightarrow^*$, with $C[\ldots v \ldots] \Rightarrow C[\ldots t \ldots]$ if $v \to t \in P$. The language of $v \in V$ is $\mathcal{L}(v) = \{t \in T_\Sigma | v \Rightarrow^* t\}$, the language of $\mathcal{G}$ is $\mathcal{L}(\mathcal{G}) = \mathcal{L}(Z)$.

For convenience, we add a lexical level to the grammar concept, allowing strings from $\mathcal{A}^*$ in place of single symbols. $L = \{char, string, empty\}$ is the set of lexical symbols. By convention,

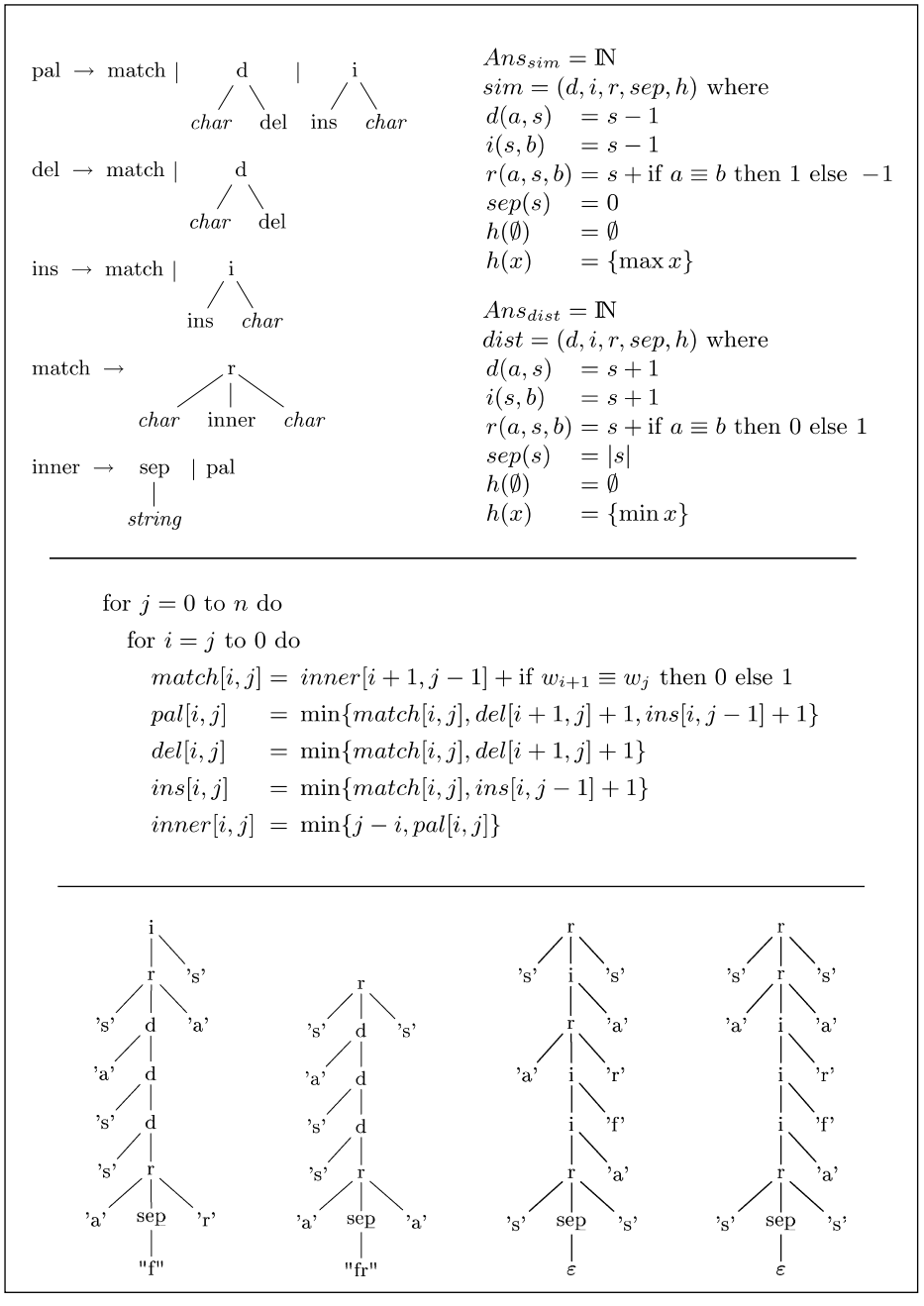


Fig. 1. Top: Example yield grammar and two evaluation algebras. The axiom symbol is *pal*, and *char* denotes an arbitrary character. Middle: The corresponding recurrences specialized for algebra *dist*. For lack of space, the initialization equations are omitted. Bottom: Four candidate structures in the search space for the best approximate separated palindrome structure for the string "sassafras". Under algebra *dist* the candidates evaluate to scores (from left to right) 7, 5, 4 and 3.

$\mathcal{L}(char) = \mathcal{A}$, $\mathcal{L}(string) = \mathcal{A}^*$, and $\mathcal{L}(empty) = \{\varepsilon\}$. At this point, the reader may want to glance ahead at the tree grammar shown in Fig. 1, upper left.

The yield function y on the trees in T_Σ is defined by $y(a) = a$ for $a \in \mathcal{A}$, and $y(f(x_1, \ldots, x_n)) = y(x_1) \cdots y(x_n)$, for $f \in \Sigma$ and $n \geqslant 0$. Note that nullary constructors by definition have yield ε, hence the $y(t)$ is the string of leaf symbols from $\mathcal{A}$ in their left to right order in t. The yield size of $v \in V$ is $\sup\{|y(t)| \mid v \Rightarrow^* t\}$. For the lexical symbols, we obtain the yield sizes 0 for *empty*, 1 for *char*, and ∞ for *string*.

1.5. Running example: palindromic structures in strings

As a running example, we shall use the analysis of palindromic structures in strings. A *separated* palindrome [14] is a string of the form $uv(u^{-1})$. In "abcdeba", for example, the separator v could be chosen to be "cde", "bcdeb", or "abcdeba". Intuitively, we might say that the first choice is the best, as it maximizes the length of u. Generally, we have some scoring scheme that determines which palindromic structure is best for a given string. In *approximate* separate palindromes we allow differences between u and u^{-1}, using the familiar string edit model with replacements, deletions and insertions. With such generalization, the number of alternative palindromic structures for a given string becomes very large. Using different scoring schemes, we may formulate the optimization objectives of maximizing self-similarity, or minimizing differences, with slightly different applications.

The choice of this example is motivated by the following properties:

- It uses familiar string editing terminology and does not require much background explanation.
- It has relevant applications, as the secondary structures formed by RNA molecules are isomorphic to approximate separated palindromes which, as a further generalization, can also be nested recursively.
- It has about the minimal size required to demonstrate the effects of different table designs.
- It is small enough to easily verify the recurrences that are generated from the more abstract description.

The last aspect may be seen as the drawback of the example not being sophisticated enough: It can be solved easily by ad hoc means, and does not motivate the use of a formal method and the desire to automate table design. This aspect is addressed in Section 3.5, where we report on a "real life" application.

2. The algebraic approach to dynamic programming

In order to study the table design problem in general, i.e., independent of a particular dynamic programming algorithm,[1] we need a framework that (1) comprises a clearly defined and practically significant class of dynamic programming problems, (2) separates the issue of tabulation from the

[1] We study the computational complexity of table design, an activity heretofore performed by human programmers. This must not be confused with studying the complexity of a particular application problem, which may be solved by dynamic programming, but also by other means.

issue of problem decomposition, and hence, (3) allows us to reason about different implementations of the same dynamic programming algorithm for any problem in this class.

Therefore, we direct our attention to dynamic programming problems and algorithms that can be expressed in the formal framework of algebraic dynamic programming (ADP), which achieves (1–3). In the following we give a brief introduction into ADP recapitulated from [13]. In the conclusion section, we shall discuss the extent to which our results pertain to dynamic programming in general.

2.1. Core concepts of algebraic dynamic programming

Dynamic programming is a programming paradigm which offers much variation. One might informally circumscribe it as solving combinatorial optimization problems by recursive decomposition and tabulation of intermediate results. But not only optimization, but also counting and enumeration problems can be solved in this way. The problem decomposition is often guided by parsing the input in multiple ways; however, someone might also consider algorithms like Dijkstra's shortest path and its generalizations [10,17] as dynamic programming algorithms, where the tabulation is related to the output rather than the input.

The ADP approach comprises a well-defined class of dynamic programming problems, which, as stated before informally, can be described by grammars and algebras. The grammar defines the search space for all possible inputs, the algebra the scoring of arbitrary solution candidates. A problem instance is given by a particular input sequence. We shall now formally introduce these concepts.

Definition 2. Let $\mathcal{A}$ be an alphabet and Σ be a signature, $w \in \mathcal{A}^*$ and $t \in T_\Sigma$. t is called a potential candidate for w if $y(t) = w$.

A candidate represents a certain internal structure associated with w. We use the term *potential* candidate here, because further conditions will be imposed on candidates soon. Fig. 1, bottom shows four alternative candidates for the same input.

Definition 3 (*Evaluation algebra*). Let Σ be a signature with sort symbol *Ans*. A Σ-evaluation algebra $\mathcal{I}$ is a Σ-algebra augmented with an objective function $h_\mathcal{I} : \mathcal{P}(Ans_\mathcal{I}) \to \mathcal{P}(Ans_\mathcal{I})$, where $\mathcal{P}(Ans_\mathcal{I})$ denotes the power set of $Ans_\mathcal{I}$. The score of candidate $t \in T_\Sigma$ achieved under Σ-evaluation algebra $\mathcal{I}$ is $t_\mathcal{I}$. If there are several alternative candidates, say $\{t_1, ..., t_k\}$, then $h_\mathcal{I}(\{t_1, ..., t_k\})$ denotes our choice.

It remains to define precisely the search space for a given input w. Usually, the candidate set is much smaller than the potential candidates $\{t \in T_\Sigma | y(t) = w\}$. We need a formalism to describe the candidate set.

Definition 4 (*Yield grammars and yield languages*). Let $\mathcal{G}$ be a tree grammar over Σ and $\mathcal{A}$, and y the yield function. The pair $(\mathcal{G}, y)$ is called a yield grammar. It defines the yield language $\mathcal{L}(\mathcal{G}, y) = \{y(t) | t \in \mathcal{L}(\mathcal{G})\}$.

Deriving a string in a yield grammar can be seen as a two stage process: first a tree in $\mathcal{L}(\mathcal{G})$ is generated, and then all the tree structure is erased. Recovering it is the task of search space construction.

Definition 5 (*Yield parsing*). Given a yield grammar $(\mathcal{G}, y)$ over $\mathcal{A}$ and a sequence $w = w_1 \cdots w_n \in \mathcal{A}^*$, the yield parsing problem is to find $P_{\mathcal{G}}(w) = \{t \in \mathcal{L}(\mathcal{G}) | y(t) = w\}$. A potential candidate t is a proper candidate if $t \in \mathcal{L}(\mathcal{G})$.
Proper candidates will just be called candidates in the sequel. Note that the input string w must be "parsed" into trees $t \in \mathcal{L}(\mathcal{G}) \subseteq T_\Sigma$, each of which in turn has a derivation according to the tree grammar $\mathcal{G}$. These derivations must exist—they ensure that candidate t corresponds to a proper problem decomposition—but otherwise, they are irrelevant and will play no part in the sequel.

Yield parsing takes a string as its input, and therefore, although based on a tree grammar, it is more closely related to string parsing than to tree parsing methods. Its closest relatives are methods for parsing ambiguous context free languages, such as Earley's or the CYK algorithm [1]. While CYK is actually a dynamic programming algorithm and can parse any context free language in $O(n^3)$ time, this efficiency result does not carry over to yield parsing. CYK requires a grammar transformation; its analog for tree grammars is prevented by Σ. This is known as the *yield parsing paradox* and is discussed in detail in [13].

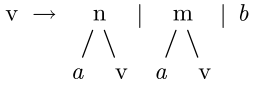
For the present development, we assume the availability of a correct yield parser $P_{\mathcal{G}}$, given as a family of parser functions. We denote by p_q the yield parser function for the nonterminal or lexical symbol q, and with $p_q(i, j)$ the application of p_q on the input subword $w_{i+1} \cdots w_j$. It returns the set of all $t \in \mathcal{L}(q)$ such that $y(t) = w_{i+1} \cdots w_j$.

Given that yield parsing constructs the search space, all that is left to do is to evaluate the candidates in a given algebra rather than simply returning the trees.

Definition 6 (*Algebraic dynamic programming*).

- An ADP problem is specified by a signature Σ over $\mathcal{A}$, a yield grammar $(\mathcal{G}, y)$ over Σ, and a Σ-evaluation algebra $\mathcal{I}$ with objective function $h_{\mathcal{I}}$.
- An ADP problem instance is posed by a string $w \in \mathcal{A}^*$. The search space it spawns is the set of all its parses, $P_{\mathcal{G}}(w)$.
- Solving an ADP problem is computing $h_{\mathcal{I}}\{t_{\mathcal{I}} \mid t \in P_{\mathcal{G}}(w)\}$ in polynomial time and space with respect to $|w|$.

Taking the above definition naively and computing all candidates before evaluating them would hardly allow us to achieve polynomial efficiency. The number of parses for $w \in \mathcal{A}^*$ can be exponential in $|w|$. An example is the tree grammar



where a yield string $a^n b$ has 2^n parses. In such a case, the size of the answer dominates the computational cost both in terms of time and space. In the subsequent analysis, we assume that eventually, only a single, optimal answer is to be reported. Hence, the number of solutions may be reduced by applying the objective function already to alternative intermediate results from subproblems. But even so, there is the possibility that the (single) result for a subproblem is calculated many times, resulting in exponential runtime. The remedy to this source of inefficiency is tabulation of intermediate results, and re-use rather than re-calculation.

These two considerations are well-known as Bellman's Principle of Optimality. In the algebraic framework, it reads as follows:

Definition 7 (*Algebraic version of Bellman's Principle*). An evaluation algebra $\mathcal{I}$ satisfies Bellman's Principle, if for each k-ary operator f in Σ and all answer sets $z_1, \ldots, z_k$, the objective function $h_{\mathcal{I}}$ satisfies

$$\begin{aligned} & h_{\mathcal{I}}(\{\, f_{\mathcal{I}}(x_1, \ldots, x_k) \mid x_1 \in z_1, \ldots, x_k \in z_k \,\}) \\ = {} & h_{\mathcal{I}}(\{\, f_{\mathcal{I}}(x_1, \ldots, x_k) \mid x_1 \in h_{\mathcal{I}}(z_1), \ldots, x_k \in h_{\mathcal{I}}(z_k) \,\}) \end{aligned}$$

as well as

$$h_{\mathcal{I}}(z \cup z') = h_{\mathcal{I}}(h_{\mathcal{I}}(z) \cup h_{\mathcal{I}}(z')).$$

The practical consequence of the optimality principle is that we may push the application of the objective function inside the computation of subproblems. The parsers p_q may apply the choice function to their alternative results, and store the outcome for later use. Thus we may prevent combinatorial explosion without affecting the overall result. The question, now, is *which* intermediate results need tabulation. This is a classical space-time trade-off; using more tables we can make the program run faster. How to exploit this trade-off in an optimal way is the subject of our current investigation.

Fig. 1 collects our formulation of the approximate separated palindrome example in the ADP framework. It shows (top left) the yield grammar defining approximate, separated palindromic structures. Recall that any string can be parsed as such a palindrome in many ways, with the evaluation algebra determining the best palindromic structure. Four candidates for the string "sassafras" are shown in the bottom part of the Figure. Two evaluation algebras are given (top right), algebra *sim* maximizing self-similarity, and algebra *dist* minimizing differences. The middle part of Fig. 1 shows the explicit recurrences derived from the grammar and algebra *dist*.

2.2. Algebraic dynamic programming in practice

The preceding section is a condensed summary of main concepts underlying the algebraic approach to dynamic programming, only to the extent they are required for our present study. ADP has been implemented as an embedded domain specific language, and is used for (but not restricted to) the development of biosequence analysis algorithms. This language offers various additional features useful in practice, like a convenient syntax for writing grammars, a more elaborate lexical level, many-sorted signatures with a choice function for each sort, syntactic predicates associated with productions, and more. The reader is referred to [13], where the ADP approach is presented in full detail, and the literature cited therein. This article also contains a section devoted to the frequently asked question why context free grammars are just not quite adequate to capture the essence of dynamic programming over sequence data, and yield grammars must be used instead.

In previous programming practice, table design was a responsibility of the programmer. The desire to automate it arises from two sources: (1) As ADP cuts down program development efforts, we embark on more sophisticated problems and the grammars get larger. We report on one such case in Section 3.5. It is no longer easy to show that one has found an optimal table design.

(2) In a recently started project on RNA secondary structure analysis, specialized grammars that model families of structures with a common shape will be generated from the data. This is done in an iterative way which does not permit human intervention, such that a good automated table design is essential.

3. Tabulation and efficiency analysis

When *all* nonterminal symbols are tabulated, the runtime efficiency of the tabulating yield parser is known. It depends on the *width* of the grammar:

Definition 8 (*Width of productions and grammar*). Let t be a tree pattern, and let k be the number of nonterminal or lexical symbols in t whose yield size is not bounded from above. We define $width(t) = k - 1$. Let $P_v = \{t | v \to t \in P\}$. We define $width(v) = \max_{t \in P_v}\{width(t)\}$ and $width(\mathcal{G}) = \max_{v \in V}\{width(v)\}$.

In this case, the execution time is $O(n^{2+width(\mathcal{G})})$ [13]. This implies minimal runtime, but maximal space consumption, and is what we want to improve upon by a more clever table design.

In Fig. 1, middle part, we show the recurrences that implement the tabulating yield parser for the example grammar. (We do not discuss here how these recurrences are derived from the grammar.) Efficiency analysis in this case is simple. With all intermediate results stored in the five tables $match, \ldots, inner$, the runtime is solely determined by the outer for-loops which leads to an execution time of $O(n^2)$. In case of productions with $width(v) \geqslant 1$, for example $v \to N(string, v)$, the recurrence equations and the eventual implementation of the corresponding parsers would contain additional loops for moving subword boundaries resulting in optimal runtimes of $O(n^3)$ or more. While tabulating everything yields optimal efficiency in terms of execution time, it may require more space than really needed. Conversely, the yield parser may implement a parser for each nonterminal symbol as a recursive function—resulting in low space requirements for tables (none, to be precise), but exorbitant inefficiency, and most likely a runtime stack overflow. The solution is to assign a table configuration to the grammar that achieves an optimal balance.

3.1. Efficiency analysis for a given table configuration

In order to make more precise statements about the usage of tables and recursive functions, we define:

Definition 9 (*Table configuration*). Let $\mathcal{G} = (V, Z, P)$ be a tree grammar over Σ and $P_{\mathcal{G}}$ the corresponding yield parser. A *table configuration* ϑ is a subset $\vartheta \subseteq V$ denoting that for all $q \in \vartheta$ the parser p_q is to be tabulated and for all $q \notin \vartheta$ the parser p_q is to be implemented as a recursive function. $P_{\mathcal{G}}^{\vartheta}$ shall represent the yield parser $P_{\mathcal{G}}$ for yield grammar $(\mathcal{G}, y)$ under table configuration ϑ. $\mathcal{P}(V)$ denotes the set of all possible table configurations.

Note that the semantics of the tabulating yield parsers $P_{\mathcal{G}}^{\vartheta}$ are equivalent for all $\vartheta \in \mathcal{P}(V)$. We will distinguish between *good* and *optimal* table configurations.

Definition 10 (*Good and optimal table configurations*). A configuration is *good* if it uses the minimal number of tables that leads to polynomial runtime. A configuration is *optimal* if it uses the minimal number of tables that leads to a runtime with the best possible polynomial degree.

Note that our notion of optimality includes a space-time trade-off. A grammar with a good table configuration may still contain a subgrammar whose dependencies raise runtime complexity to an arbitrary (but fixed) polynomial degree. Extra tables can be assigned to reduce this degree—hence an optimal configuration, in general, holds more tables than a good one. For example, the second configuration shown in Fig. 2 is good (using 1 table), while the first configuration is optimal (using 3 tables).

Definition 11 (*Dependence mapping*). Let $S = V \cup L$ be the set of nonterminal and lexical symbols in $\mathcal{G}$. The *dependence mapping* u for $q \in V$ and input length n is given by $u(q,n) = (d_1, \ldots, d_r)$ where each $d_k, 1 \leqslant k \leqslant r$ represents a dependence property $d_k \in S \times \mathbb{N}$ such that $d_k = (q_k, i_{q_k} + j_{q_k})$ if and only if the parser $p_q(0,n)$ uses the result of $p_{q_k}(i_{q_k}, n - j_{q_k})$. For convenience, we simply denote $(q_k, s_{q_k}) \in u(q,n)$.

Theorem 1. *The runtime of an ADP algorithm implemented by a tabulating yield parser $P_{\mathcal{G}}^{\vartheta}$ on input w of length n is given by $r(P_{\mathcal{G}}^{\vartheta}, n)$ where:*

$$\bar{r}(\vartheta, q, 0) = 1 \tag{1}$$

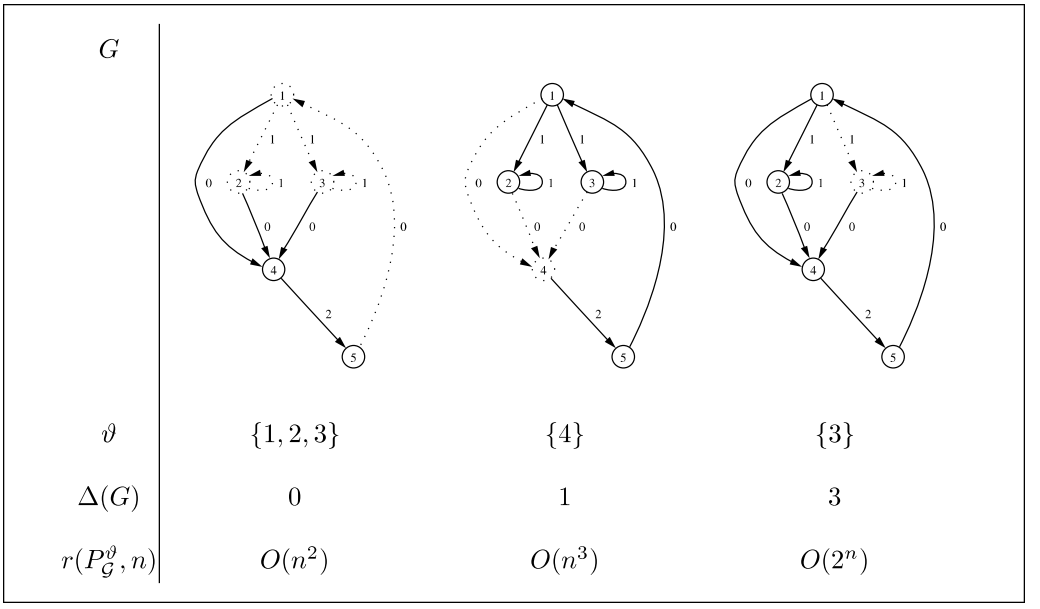


Fig. 2. Three dependence graphs for the yield grammar of Example 1.5, with corresponding table configurations ϑ and circuit degrees $\Delta(G)$. The vertices are numbered as follows: $1 \to$ `pal`, $2 \to$ `del`, $3 \to$ `ins`, $4 \to$ `match`, $5 \to$ `inner`. Table access dependences are drawn as dotted edges. The runtime of the yield parser $P_{\mathcal{G}}$ under table configuration ϑ is denoted by $r(P_{\mathcal{G}}^{\vartheta}, n)$.

$$\bar{r}(\vartheta,q,n) = \sum_{(q',s_{q'})\in u(q,n)} \begin{cases} 1 & \text{for } q' \in L \\ 1 & \text{for } q' \in V, q' \in \vartheta \\ \bar{r}(\vartheta,q',n-s_{q'}) & \text{otherwise} \end{cases} \tag{2}$$

$$r(\vartheta,q,n) = \begin{cases} n^2 \cdot \bar{r}(\vartheta,q,n) & \text{for } q \in \vartheta \\ \bar{r}(\vartheta,q,n) & \text{otherwise} \end{cases} \tag{3}$$

$$r(P_G^{\vartheta},n) = O(\max_{q\in V} r(\vartheta,q,n)) \tag{4}$$

Proof. The dependence mapping $u(q,n)$ provides all calls to lexical or nonterminal parsers in the calculation of parser p_q on input length n. Without loss of generality we assume that each parser terminates at an input of length 0 (1). Lexical parsers and table accesses (2) can be performed in constant time. In case of a non-tabulated parser $p_{q'}$ (2), the parser $p_{q'}$ is called with an input length reduced by $s_{q'}$. The outer for-loops for a tabulated parser lead to a minimal effort of at least $O(n^2)$ (3) and the overall runtime of the algorithm (4) is determined by the parser with the maximal asymptotic runtime. □

This of course gives sparse information about a closed form for $r(P_G^{\vartheta},n)$. All we know is that if all productions are tabulated, the runtime is solely affected by the constants in (2) and the outer for-loops of (3).

In case of non-tabulated productions, (2) is defined recursively, which makes reasoning about complexity more difficult. Tabulating no production at all will in most cases lead to an exponential runtime. In the following we will investigate how to find the minimal numbers of tables needed to achieve a polynomially bounded runtime $r(P_G^{\vartheta},n)$.

3.2. Staying polynomial

The number of dependences for a parser p_q is bounded by $|u(q,n)| = O(n^{width(q)})$. Therefore, $r(P_G^{\vartheta},n)$ can only become exponential in the presence of recursive calls between parser functions (Eq. 2).

For the following developments it is convenient to introduce some terminology for graphs: A directed graph $G = (V,E)$ is given by a nonempty set $V = \{v_1,\ldots,v_s\}$ of vertices and a set $E = \{e_1,\ldots,e_m\}$ of edges consisting of ordered pairs of elements of V. A directed graph with multiple edges is called a directed *multigraph* with edges given as a multiset E. A *loop* is an edge connecting a vertex with itself. In the following we only consider directed multigraphs with loops allowed which we will simply call graphs. In a *weighted graph* each edge e_i is also associated with a weight $w(e_i)$.

A directed *walk* is a sequence $\pi = v_0e_1v_1e_2\cdots e_lv_l$, where $v_i \in V, e_j \in E$ for $0 \leqslant i \leqslant l, 1 \leqslant j \leqslant l$, and each edge e_j is directed out of vertex v_{j-1} and directed into vertex v_j. Note that the edges and vertices in π are not necessarily distinct. The length of a walk is the number of edges in the sequence and is denoted by $l(\pi)$. The weight of a walk is the sum of the weights of its edges and is denoted by $\tilde{w}(\pi) = \sum_{i=1}^{l(\pi)} w(e_i)$.

A *circuit* is a closed walk with $v_0 = v_l$ and all edges distinct. Under this definition a loop is also a circuit. Two circuits are distinct if one is not a cyclic permutation of the other. The *circuit degree* $\delta(v)$ of a vertex v shall be the number of distinct circuits containing v. We define the circuit degree of

a graph G as $\Delta(G) = \max_{v \in V} \delta(v)$. A circuit which contains no vertex more than once (apart from the initial one), is called *elementary*.

The dependence mapping given by u for a yield parser $P_{\mathcal{G}}^{\vartheta}$ can be represented by a weighted directed *dependence graph* G with the nonterminal set V as vertex set and an edge from q to q_k with weight s_{q_k} for each $(q_k, s_{q_k}) \in u(q, n)$. Such a graph represents all dependences, regardless whether by table access or by function call. A graph limited to function call dependences can be derived by restricting the edge set to edges directed into vertices from $V \setminus \vartheta$.

In the following we restrict the analysis to grammars $\mathcal{G}$ with $width(\mathcal{G}) = 0$. This leads to parsers with constant dependence, i.e. all references to parser functions have the form $p_q(i + i_q, j - j_q)$, with $i_q, j_q \in \mathbb{N}$ and $i_q, j_q \geqslant 0$. This restriction is necessary in order to obtain graphs that are independent from the input length n contained in the dependence mapping u.

Definition 12 (*Dependence graph*). Let $P_{\mathcal{G}}^{\vartheta}$ be a yield parser for yield grammar $(\mathcal{G}, y)$ under table configuration ϑ and let u be the corresponding dependence mapping. The *dependence graph* $G(P_{\mathcal{G}}^{\vartheta})$ is given by the grammar's nonterminal set V as vertex set and edges $E = \{e_1, \ldots, e_m\}$. Edge $e_i = (q, q_k)$ with $w(e_i) = s_{q_k}$ is in E if and only if $q_k \notin \vartheta$ and $(q_k, s_{q_k}) \in u(q, n)$. For convenience, we denote G for $G(P_{\mathcal{G}}^{\vartheta})$.

The use of the dependence graph is influenced by [16] where such graphs are used to represent systems of uniform recurrence equations. Fig. 2 shows three graphs with different table configurations and circuit degrees $\Delta(G)$ for the palindrome example.

Each edge in the dependence graph represents a function call between two parser functions. A walk π in the graph can therefore be interpreted as a sequence of consecutive function calls where the weight $\tilde{w}(\pi)$ of the walk is the length of the "consumed" input and the length $l(\pi)$ is the number of required function calls.

This leads to the following relation between the runtime of a yield parser and its dependence graph:

Theorem 2. *Let $P_{\mathcal{G}}^{\vartheta}$ be a tabulating yield parser with corresponding dependence graph G. Then, $P_{\mathcal{G}}^{\vartheta}$ has polynomial runtime if and only if $\Delta(G) \leqslant 1$.*

Proof. If there is no vertex on at least two different circuits, each recursive function is called at most once for each combination of parameters, such that the runtime is polynomial. If there is such a vertex v, a call on v can trigger two recursive calls to v. Since the recursion depth is linear, this leads to $O(2^n)$ calls, and the runtime is exponential. □

3.3. NP-completeness of table design

Theorem 2 gives a convenient property to distinguish between polynomial and exponential runtimes of a tabulating yield parser. In order to find good table configurations, we need to search for configurations that lead to dependence graphs with $\Delta(G) \leqslant 1$ while using a minimal number of tables.

A simple algorithm is to systematically test all possible table configurations for their impact on the number of circuits in the dependence graph. Such approach would be exponential in the number of nonterminal symbols, so the question arises whether a better algorithm exists.

Graph theory is a stomping ground for NP-complete and harder problems and provides a rich theory on these kinds of questions. And indeed, we can utilize a powerful result from graph theory to give an answer to the question stated above:

Theorem 3. (*NP-completeness of good table design*) *Finding the minimal number of tables to prevent exponential runtime of an ADP algorithm implemented by a tabulating yield parser $P_{\mathcal{G}}$ is NP-complete.*

Proof. Recall the construction of the dependence graph in Definition 12. The edge set E for the minimal table configuration $\vartheta = \emptyset$ represents all dependences between parsers for the situation that no results are tabulated. Adding a table for the parser p_q to the configuration results in deletion of all edges directed into the corresponding vertex q. Obviously, for the number of circuits in the graph this has the same effect as deleting the vertex q itself. This leads to the observation that finding the minimal number of tables to prevent exponential runtime is equivalent to finding the minimal number of vertices which must be deleted from the dependence graph such that the resulting graph contains no vertex which is part of two or more circuits. And this is exactly a classic node-deletion problem:

For a fixed graph property Π, find the minimum number of nodes (or vertices) which must be deleted from a given graph so that the result satisfies Π [18].

It was shown in [18] that if Π is a "nontrivial" property which is "hereditary" on induced subgraphs, then the node-deletion problem for Π is NP-hard. Furthermore, if testing for Π can be performed in polynomial time, then the node-deletion problem for Π is NP-complete.

In our application, the property Π is: the graph contains no vertex which is part of two or more circuits. To finish the proof, we show that Π satisfies the properties claimed above. A graph property is *nontrivial* if infinitely many graphs satisfy it and infinitely many graphs fail to satisfy it. This is clearly given for Π. A property is *hereditary* if for a given graph satisfying Π, every node-induced subgraph also satisfies Π. It is obvious that all subgraphs of a graph containing no vertex being part of two or more circuits retain this property. Finally, Π can be tested for in polynomial time. A simple algorithm is to calculate all strongly connected components of the graph—which is linear in the number of vertices and edges [27]—and to check whether each of them is either a single vertex or an elementary circuit—which is linear as well. Hence, finding the minimal number of tables to prevent exponential runtime is NP-complete. □

Of course, in practical applications we are mostly interested in the *optimal* configurations, not necessarily in the good ones. We can use the same approach to show the NP-completeness of the problem of finding optimal configurations.

Theorem 4 (*NP-completeness of optimal table design*). *Finding the minimal number of tables that are necessary to achieve the best possible asymptotic runtime of an ADP algorithm implemented by a tabulating yield parser $P_{\mathcal{G}}$ is NP-complete.*

Proof. The optimal execution time of a yield parser $P_{\mathcal{G}}$ with $width(\mathcal{G}) = 0$ is $O(n^2)$. This is clear, since when all parsers are tabulated, the dependence graph is empty. Circuits in the corresponding dependence graph, arising from non-tabulated productions, represent table configurations that lead to suboptimal runtimes of at least $O(n^3)$. Therefore, in order to obtain optimal runtime for the algorithm, the dependence graph must be made cycle free. Achieving this by the minimal number

of tables is equivalent to the node-deletion problem for acyclicity of a directed graph. And this is, as expected, also NP-complete [18]. □

In the course of proving Theorems 3 and **??**, we have restricted the yield grammars considered. Our results, however, pertain to general yield grammars as well, because (1) arbitrary yield grammars contain the restricted ones as a subclass, such that their table design problem is at least NP-hard. (2) Testing goodness or optimality of a given table configuration works as described earlier also in the general case, hence NP-completeness is retained.

3.4. A pragmatic implementation strategy

Motivated by the above NP-completeness results, we present a pragmatic approach to the table design problem. It uses annotations by the programmer, preprocessing analyses, and a brute force algorithm.

The programmer's annotation distinguishes between two types of nonterminal symbols: the ones that shall be tabulated, and the ones that shall be implemented as nontabulated recursive functions. We denote them ϑ^a and ϱ^a, respectively.

Similarly, we consider two additional sets of nonterminals, ϑ^p and ϱ^p, derived from preprocessing phases. We use different phases, depending on whether we search for good or for optimal configurations:

Good configurations. Productions having more than one self-reference need to be tabulated in any case: $\vartheta^p = \{v | v \in V, \Delta(G(P_{\mathcal{G}}^{V\setminus\{v\}})) > 1\}$. Conversely, nonterminals not contained in any circuit can by no means be responsible for exponential runtime: $\varrho^p = \{v | v \in V, \delta(v) = 0\}$ with $G = G(P_{\mathcal{G}}^{\vartheta^a \cup \vartheta^p})$.

Optimal configurations. We derive the best possible polynomial degree by calculating the runtime for a full table configuration: $r_{opt} = r(P_{\mathcal{G}}^{V}, n)$ (see Theorem 1). When not tabulated, even a single self-recursive production or a production containing an inner loop can be responsible for an additional runtime factor. We identify these candidates and mark them for tabulation: $\vartheta^p = \{v | v \in V, r(P_{\mathcal{G}}^{V\setminus\{v\}}, n) > r_{opt}\}$. Clearly, nonterminals with a constant runtime need no tabulation: $\varrho^p = \{v | v \in V, r(\vartheta^a \cup \vartheta^p, v, n) = O(1)\}$.

The remaining nonterminals are free for optimization by the brute force approach: $F = V\setminus\{\vartheta^a \cup \varrho^a \cup \vartheta^p \cup \varrho^p\}$. Let Ω be a predicate on table configurations that determines whether the configuration is relevant for optimization at all. Corresponding to Theorem 2, we use $\Omega(\vartheta) = \Delta(G(P_{\mathcal{G}}^{\vartheta})) \leqslant 1$ for good configurations and $\Omega(\vartheta) = r(P_{\mathcal{G}}^{\vartheta}, n) \equiv r_{opt}$ for the optimal ones.

Let $\vartheta^i = \vartheta^a \cup \vartheta^p$ be the initial configuration. Then, $\Theta = \{\vartheta \cup \vartheta^i | \vartheta \in \mathcal{P}(F), \Omega(\vartheta \cup \vartheta^i)\}$ describes the set of configurations satisfying the predicate and $\Theta_{\min} = \{\vartheta | \vartheta \in \Theta, |\vartheta| = \min_{c\in\Theta} |c|\}$ describes the corresponding minimal configurations. Our brute force algorithm actually enumerates these sets and tests for minimality.

Should this strategy not be effective (due to computational effort for $\Theta_{\min}$), one must restart with a more detailed annotation by the programmer.

In the implementation, we must also deal with inner loops arising from productions with $width(v) \geqslant 1$, which played no role in the NP completeness proofs. During the construction of the dependence graphs we represent dependences from inside inner loops by *two* edges instead of a single one. Since

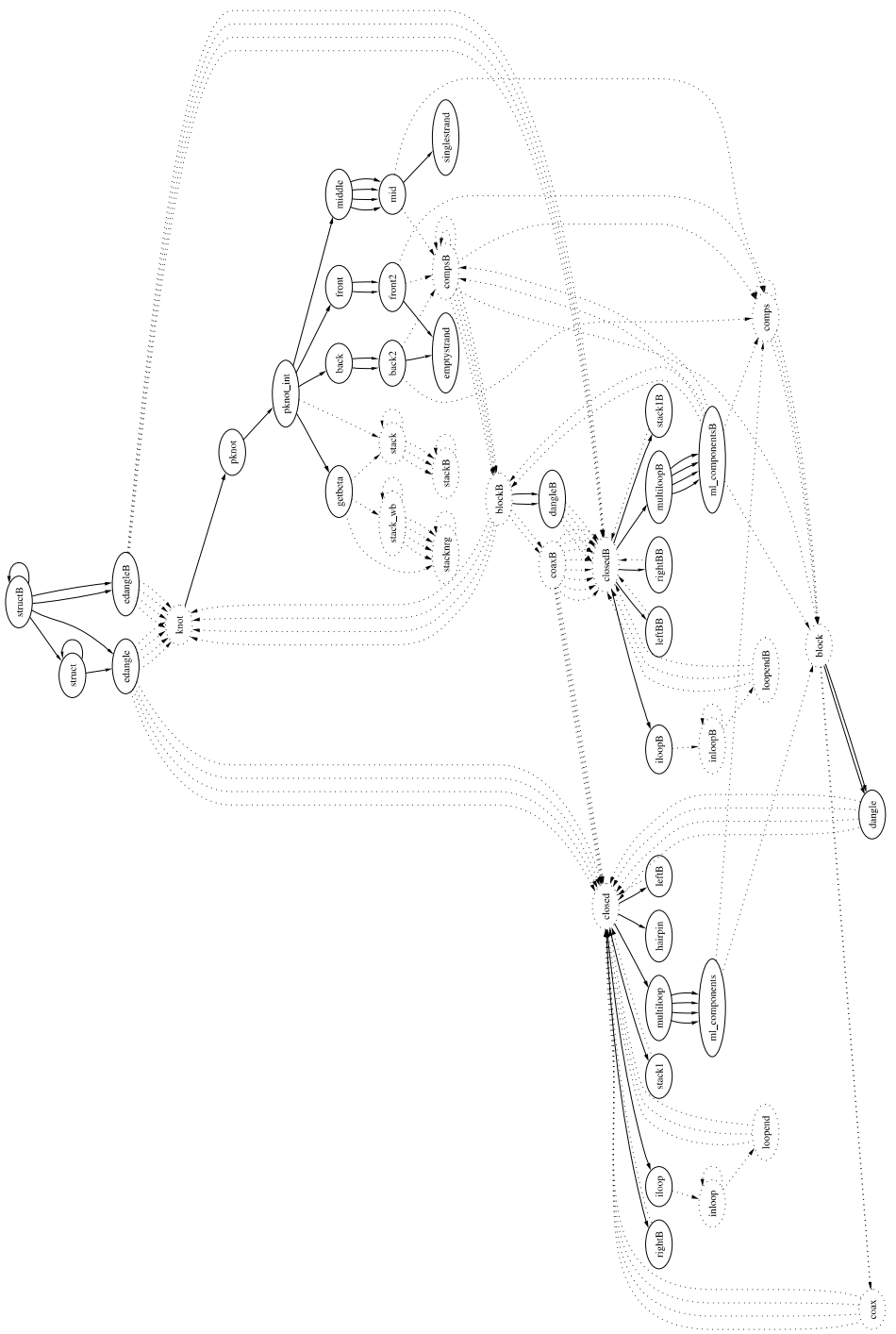


Fig. 3. Dependence graph for *pknotsRG*—manual table design (17 tables). This table configuration leads to time and space requirements of $O(n^4)$ and $O(n^2)$. The tabulated nonterminals are shown as nodes in dotted ovals; their calling arcs are also dotted.

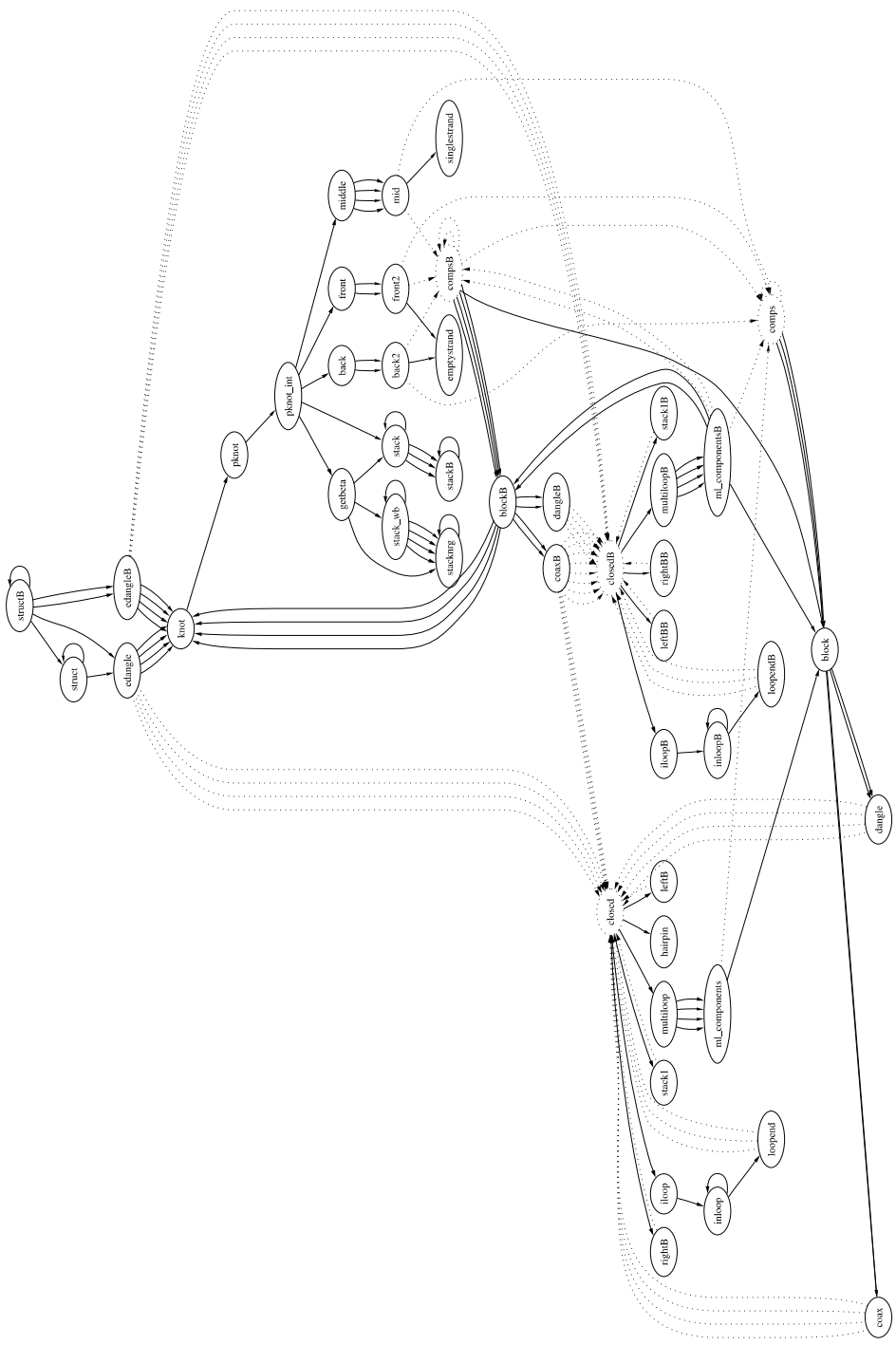


Fig. 4. Dependence graph for *pknotsRG*—good table design (4 tables). This table configuration leads to time and space requirements of $O(n^7)$ and $O(n^2)$, respectively. Albeit polynomial, the runtime of $O(n^7)$ is surely unacceptable for practical usage.

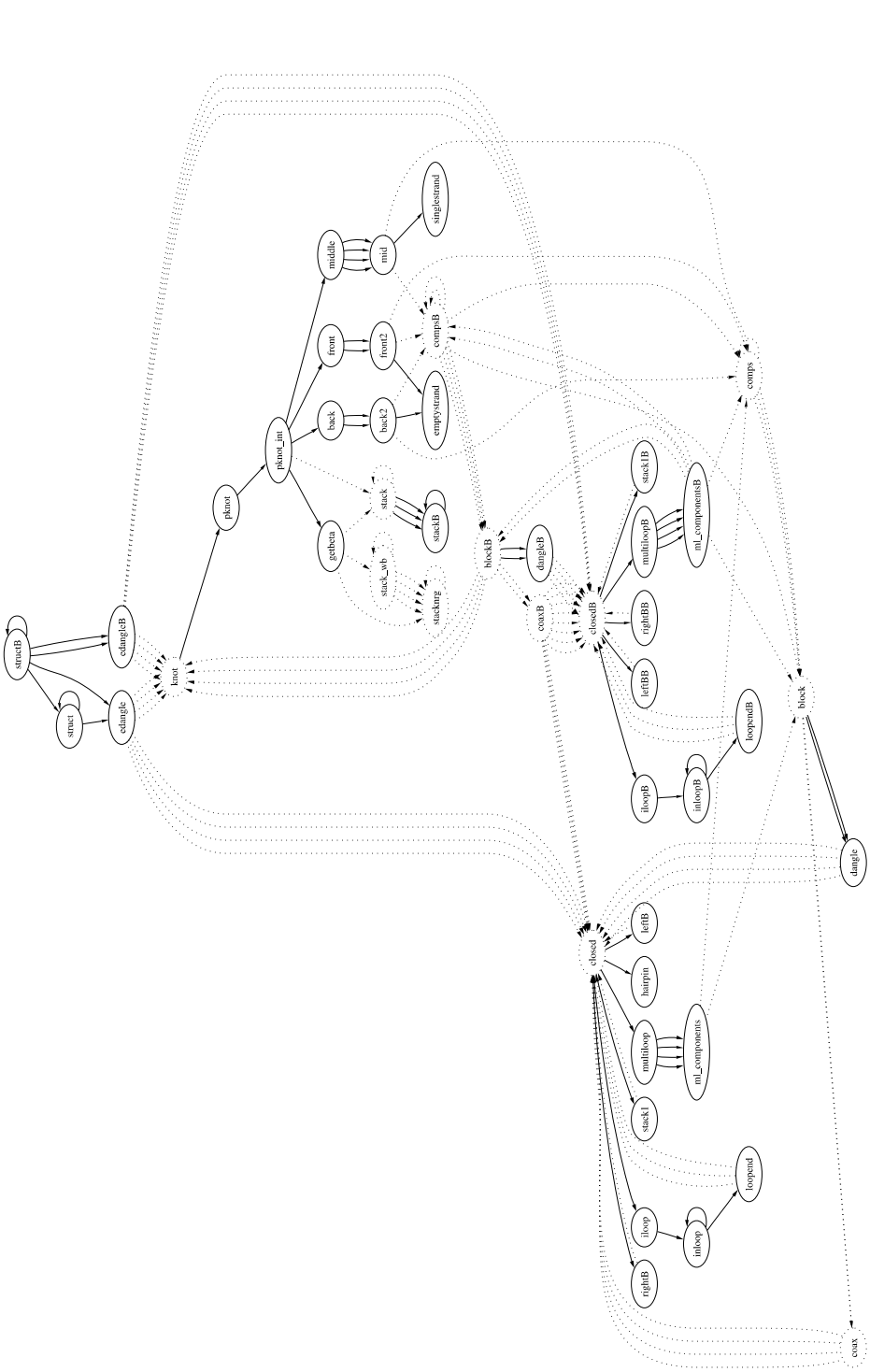


Fig. 5. Dependence graph for *pknotsRG*—optimal table design (12 tables). This table configuration leads to time and space requirements of $O(n^4)$ and $O(n^2)$, respectively, the same as the manual design, but uses only about 70% of the table space.

our testing criterion is $\Delta(G) \leqslant 1$, this ensures that all configurations with circuits containing an inner loop are rejected.

3.5. Practical experience

The table design problem studied here mathematically arises in practice only when the algorithmic task at hand has a certain minimal level of sophistication. To support our claim of practical relevance, we give a short report on such an application. Our largest ADP-program so far is *pknotsRG* [25,23], which predicts RNA secondary structures including the so-called pseudoknots. Its time and space requirements are $O(n^4)$ and $O(n^2)$. It uses a yield grammar with 47 nonterminal symbols. The productions split up in a total of 140 alternatives, which indicates the complexity of the case analysis involved. Therefore, space does not permit to explain the algorithm here. We show the dependence graphs for three different table configurations in Figs. 3 – 5. The original implementation (Fig. 3)—developed using manual table design—has a configuration of 17 tables. With moderate annotation, our strategy derived good configurations of 4 tables (Fig. 4) and optimal configurations of 12 tables (Fig. 5).

Comparing the hand-made to the optimal design, we found that while saving nearly 30% space, the runtime of the optimized implementation increased by 19%—the constant factor due to the additional recursive function calls. In many applications in the bioinformatics domain, including this one, available space is the limiting factor. In such cases, one is thankful to accept a small slowdown in exchange for the ability to handle larger input data.

4. Conclusion

The main difficulty in the traditional development of dynamic programming algorithms can be seen in the lack of a systematic approach for the definition of suitable recurrence equations. The ADP approach and its associated program development discipline [13] alleviate these difficulties somewhat, but also shed light on a deeper challenge. We have shown that the algorithm designer—whenever the algorithm is nontrivial and space usage is an issue—has to solve an NP-complete design problem. He does so explicitly within the ADP framework, and implicitly in the more traditional approach to dynamic programming. Although a *human* is not limited by the laws of *computational* complexity, we still may take our result as the indicator of an intrinsic difficulty in the design of dynamic programming algorithms. What has been shown here using the ADP framework pertains *prima facie* to dynamic programming over sequential data, but it may well hold for dynamic programming in general. It is unlikely that dynamic programming over more structured data (trees, graphs, etc.) should have a simpler table design problem.

The efficiency analysis we are trying to automate and the complexity result attained for this problem must not be mistaken to imply complexity results about particular application problems. When our algorithm determines that a dynamic programming problem P, specified in algebraic style, has an optimal table configuration of k tables leading to runtime $O(n^r)$, this does not imply that problem P cannot be solved faster than in $O(n^r)$. Maybe P can be solved by other algorithmic approaches, and a table design problem does not arise at all.

Our pragmatic table design algorithm described above performs well on the largest problem instances we have encountered so far. It demonstrates that there are significant time-space trade-offs, whose systematic exploitation is yet to be studied in more detail. However, it is certainly not the only, and probably not the best approach. In our project mentioned earlier, where yield grammars are to be generated automatically from the data (the evaluation algebra is fixed for all these grammars), we now have the obligation to also generate some "user" annotation required to make our approach practical. An alternative would be to give up user annotation and optimality, and instead develop an approximation scheme for the table design problem.

Acknowledgment

We are grateful to Marc Rehmsmeier for a careful reading of this manuscript.

References

[1] A. Aho, J. Ullman, The Theory of Parsing, Translation and Compiling, Prentice-Hall, Englewood Cliffs, NJ, 1973.
[2] A. Aho, J. Hopcroft, J. Ullman, Data Structures and Algorithms, Addison-Wesley, Reading, MA, 1982.
[3] V. Bafna, N. Edwards, On de novo interpretation of tandem mass spectra for peptide identification, in: Proceedings of the Seventh Annual International Conference on Computational Molecular Biology, ACM Press, New York, 2003, pp. 9–18.
[4] R. Bellman, Dynamic Programming, Princeton University Press, Princeton, NJ, 1957.
[5] R. Bellman, S. Dreyfus, Applied Dynamic Programming, Princeton University Press, Princeton, NJ, 1962.
[6] H. Bodlaender, J. Telle, Space-efficient construction variants of dynamic programming, Nordic J. Comput. 11 (4) (2004) 374–385.
[7] W. Brainerd, Tree generating regular systems, Inform. Control 14 (1969) 217–231.
[8] G. Brassard, P. Bratley, Algorithmics: Theory and Practice, Prentice-Hall, Englewood Cliffs, NJ, 1988.
[9] T. Cormen, C. Leiserson, R. Rivest, Introduction to Algorithms, MIT Press, Cambridge, MA, 1990.
[10] E.W. Dijkstra, A note on two problems in connexion with graphs, Numer. Math. 1 (1959) 269–271.
[11] R. Durbin, S. Eddy, A. Krogh, G. Mitchison, Biological Sequence Analysis, Cambridge University Press, Cambridge, MA, 1998.
[12] R. Giegerich, A systematic approach to dynamic programming in bioinformatics, Bioinformatics 16 (2000) 665–677.
[13] R. Giegerich, C. Meyer, P. Steffen, A discipline of dynamic programming over sequence data, Sci. Comput. Program. 51 (3) (2004) 215–263.
[14] D. Gusfield, Algorithms on Strings, Trees, and Sequences, Computer Science andComputational Biology, Cambridge University Press, Cambridge, MA, 1997.
[15] D. Hirschberg, A linear space algorithm for computing maximal common subsequences, Commun. ACM 18 (6) (1975) 341–343.
[16] R.M. Karp, R.E. Miller, S. Winograd, The organization of computations for uniform recurrence equations, J. ACM 14 (3) (1967) 563–590.
[17] D.E. Knuth, A generalization of Dijkstra's algorithm, Inform. Process. Lett. 6 (1) (1977) 1–5.
[18] J.M. Lewis, M. Yannakakis, The node-deletion problem for hereditary properties is NP-complete, J. Comput. System Sci. 20 (2) (1980) 219–230.
[19] Y. Liu, S. Stoller, Program optimization using indexed and recursive data structures, in: PEPM '02: Proceedings of the 2002 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation, ACM Press, New York, NY, 2002, pp. 108–118.
[20] K. Mehlhorn, Data Structures and Algorithms, Springer-Verlag, New York, 1984.
[21] E. Myers, W. Miller, Optimal alignment in linear space, Comput. Appl. Biosci. 4 (1) (1988) 11–17.

[22] in: L. Pachter, B. Sturmfels (Eds.), Algebraic Statistics for Computational Biology, Cambridge University Press, Cambridge, MA, 2005.
[23] J. Reeder, R. Giegerich, Design, implementation and evaluation of a practical pseudoknot folding algorithm based on thermodynamics, BMC Bioinformatics, 5 (104) (2004), in press.
[24] G. Rote, Path problems in graphs, in: in: G. Tinhofer, E. Mayr, H. Noltemeier, M. Syslo (Eds.), Computational Graph Theory, Computing Supplementum, vol. 7, Springer-Verlag, New York, 1990, pp. 155–189.
[25] A. Sczyrba, J. Krüger, H. Mersch, S. Kurtz, R. Giegerich, RNA-related tools on the bielefeld bioinformatics server, Nuclic Acids Res. 31 (13) (2003) 3767–3770.
[26] R. Sedgewick, Algorithms, Second ed., Addison-Wesley, Reading, MA, 1989.
[27] R.E. Tarjan, Depth-first search and linear graph algorithms, SIAM J. Comput. 1 (2) (1972) 146–160.