Sparsification in Algebraic Dynamic Programming

Mathias Möhl*
Bioinformatics, Institute of Computer Science
Freiburg University, Germany

Christina Schmiedl*
Bioinformatics, Institute of Computer Science
Freiburg University, Germany

Shay Zakov
Department of Computer Science
Ben Gurion University of the Negev, Israel

Abstract

Sparsification is a technique to speed up dynamic programming algorithms which has been successfully applied to RNA structure prediction, RNA-RNA-interaction prediction, simultaneous alignment and folding, and pseudoknot prediction. So far, sparsification has been more a collection of loosely related examples and no general, well understood theory. In this work we propose a general theory to describe and implement sparsification in dynamic programming algorithms. The approach is formalized as an extension of Algebraic Dynamic Programming (ADP) which makes it applicable to a variety of algorithms and scoring schemes. In particular, this is the first approach that shows how to sparsify algorithms with scoring schemes that go beyond simple minimization or maximization, like enumeration of suboptimal solutions and approximation of the partition function. As an example, we show how to sparsify different variants of RNA structure prediction algorithms.

1 Introduction

Dynamic programming (DP) algorithms enjoy a great popularity in many areas of bioinformatics, ranging from simple sequence alignments, over RNA structure prediction to predictions of RNA interactions or the Viterby algorithm used for Hidden Markov Models. Since efficiency is a major concern for large scale analysis, various techniques have been developed to speed-up DP algorithms over sequence data. The techniques used comprise the Four-Russians method [1, 2, 3], Valiant's approach [4, 5, 6], as well as various sparsification approaches [7, 8, 9, 10, 11, 12]. Among those techniques, sparsification has been most popular, as it is applicable to a wide range of applications, comparably easy to implement, and yields good speed-ups in both theory and practice.

Sparsification approaches consider the intermediate results computed so far to identify parts of the computation that can be ignored since they cannot contribute to a solution anymore. As a consequence, the DP tables become sparse and require less computation time and in some implementations also less space. Sparsification has been recently applied to RNA structure prediction [8, 10], simultaneous alignment and folding [9], RNA-RNA-interaction prediction [11], and the prediction of RNA pseudoknot structures [12]. In all those applications sparsification yielded a significant speed-up. For RNA structure prediction, for example, it reduces the complexity from $O(n^3)$ to $O(n^2\psi(n))$, where $\psi(n)$ has been shown to be significantly smaller than n.

While it is intuitively clear that all sparsification approaches mentioned above are somehow related, they have not been characterized in term of a common framework that highlights their similarities and differences. In order to talk about DP algorithms at a general level we make use of Algebraic Dynamic Programming (ADP) [13]. ADP is a formalism that allows to formulate DP algorithms over sequence data in a simple, declarative, and formally precise way. There also exist compilers [14, 15] that yield automatically generated implementations for ADP programs. As ADP is very general, it has been used

*joint first authors

to develop tools for a variety of different tasks like RNA shape analysis (RNAshapes [16]), prediction of RNA pseudoknot structure (pknotsRG [17]), or the prediction of microRNA/target duplexes (RNAhybrid [18]).

This paper has two main contributions. First, we extend the ADP formalism to handle sparsification. This leads to a more general understanding of the entire concept of sparsification and allows to quickly implement sparsified variants of DP algorithms formulated in ADP. Second, we show that the general notion of sparsification we propose also works for advanced scoring schemes. So far, sparsification has only been applied to simple minimization and maximization problems, but our notion of sparsification generalizes effortlessly to the enumeration of suboptimal solutions and approximation of the partition function.

2 A quick overview on ADP

DP algorithms are usually described with recursion equations. Those equations contain all essential aspects of the program including the recursive structure of the problem decomposition, the scoring scheme, and the question which sub-results should be tabulated. ADP is an alternative way to describe DP algorithms over sequence data which clearly separates all those aspects. As a detailed introduction of ADP is out of the scope of this paper, we will only give a quick overview based on the example of the Nussinov algorithm. More details about ADP can be found in [13].

The Nussinov algorithm [19] is a classical algorithm to predict for a given RNA sequence a nested structure with maximal number of base pairs (where only certain pairs, namely A-U, G-U and C-G pairs are allowed). It recursively computes the maximum number of base pairs of a subsequence from position i to j as $N(i,i-1) = 0$, $N(i,i) = 0$ and for $j > i$

$$N(i,j) = \max \begin{cases} N(i,j-1) \\ \max\limits_{\substack{k \text{ s.t.} \\ (k,j) \text{ valid base pair, } i \le k < j}} N(i,k-1) + N(k+1,j-1) + 1 \end{cases} \tag{1}$$

The first recursion case represents structures where the last position j is unpaired and the second case covers pairing of j with any position k. An analogous ADP program would consist of the four grammar rules

$$N \quad \rightarrow \quad \begin{array}{c}\text{nil}\\ | \\ \varepsilon\end{array} \;\Big|\; \begin{array}{c}\text{single}\\ | \\ a\end{array} \;\Big|\; \begin{array}{c}\text{unpaired}\\ \diagup\;\diagdown \\ N \quad a\end{array} \;\Big|\; \begin{array}{c}\text{paired}\\ \diagup\;\diagup\;\diagdown\;\diagdown \\ N \quad a \quad N \quad \hat{a}\end{array} \qquad \ldots h \tag{2}$$

representing the two base cases and the two recursive cases, and the algebra

$$\begin{array}{lll} \text{nil}() = 0 & \text{single}(a) = 0 & \text{unpaired}(x,a) = x \\ \text{paired}(x,a,y,\hat{a}) = x+y+1 & h(X) = \max X & \end{array} \tag{3}$$

representing the scoring. The grammar is similar to a context free grammar (CFG) but with trees on the right hand side of the rules. If only the leaves of the trees are considered, a normal context free grammar is obtained. In the example, the grammar has a nonterminal symbol N and terminal symbols a and $\hat{a}$ representing arbitrary bases such that $(a,\hat{a})$ is a valid base pair. The inner nodes of the trees are functions of the algebra. During the run of an ADP program, the input sequence is parsed as in a usual CFG. But since the right hand side of the grammar rules are trees, each derivation constructs a tree instead of a string by recursively replacing each nonterminal by the subtree derived by it[1]. The input sequence *CAG*

[1]Note that these trees are not identical to the derivation trees of CFGs.

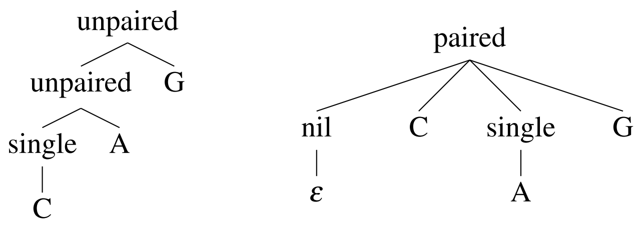


Figure 1: Two parses of the input "CAG" for the grammar given in (2).

has, for example, different parses (see Fig. 1) representing the structure without base pairs and the one with the base pair $C-G$. Each such tree can be interpreted as a term whose evaluation yields some score. In this example, the first tree evaluates to 0 and the second one to 1 since the trees represent structures with 0 and 1 base pair, respectively.

The choice function h is some additional annotation in the grammar that we ignored so far. It describes the optimization criterion, i.e. which of the parses are the solution(s). Formally, it takes a set of parses (or more precisely a multiset of their scores) as argument and returns the optimal score(s). In most cases the result is a singleton set, but ADP programs can also, for example, enumerate all possible parses or pick the best 10.[2]

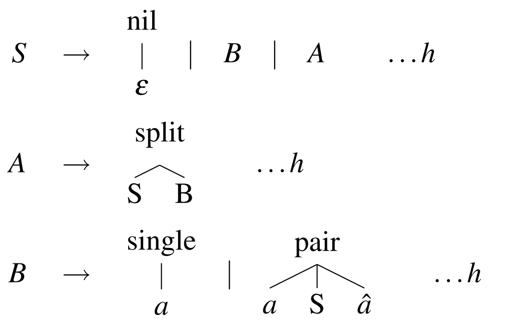
When the choice function is applied at the very end, after the parser constructed all possible parses, the ADP program behaves like a simple generate and test program that enumerates all possible structures and then picks the one with maximum number of base pairs. This can be turned in efficient dynamic programming by interleaving a CYK like grammar parser with the application of the choice function. More precisely, the CYK parser runs as usual recursively computing for all subsequences from some i to some j and each nonterminal X the set of all possible parses. But instead of storing in a table entry $X[i,j]$ the set of all parses, the parses are first evaluated to their score and then the choice function is applied such that only the (multisets of) optimal scores need to be stored in the table. Assume, for example, that "CAG" is a part of some longer input sequence. Then only the second possible parse for "CAG" would be considered when recursively constructing parses for any larger parts of the input, as its score is the maximum among the two parses of the fragment.

You can also choose another recursive decomposition. The grammar and algebra shown in Fig. 2 create a Nussinov variant which is more suitable for sparsification. The decomposition says that any RNA structure (S) can either be split into two parts (A) or is a closed structure (B) which consists either of a single base or an outermost basepair with an arbitrary structure below.

3 A general extension for sparsification in ADP

So far, sparsification is not a well defined concept but merely a collection of examples. The essential basis of all those approaches [8, 9, 11, 12] is to introduce specific tests at some places in the recursion to determine whether the currently computed value is a *candidate*. For (some of the) subsequent computations then only candidates need to be considered whereas other entries can be safely ignored without affecting the correctness of the recursion. The test for candidacy depends on the respective algorithm (e.g. the OCT and STEP criteria in Sect. 4). We propose the following general extension of ADP that is neither limited to any fixed such criterion nor to a specific choice function h.

[2]$h(X)$ must not even be a subset of X. For partition function computation, for example, $h(X)$ computes the sum of all elements of X.



$$\begin{aligned}\mathrm{nil}() &= 0\\ \mathrm{split}(x,y) &= x+y\\ \mathrm{single}(a) &= 0\\ \mathrm{pair}(a,x,\hat{a}) &= x+1\\ h(X) &= \max X\end{aligned}$$

Figure 2: Grammar (on the left) and algebra (on the right) for a variant of the Nussinov algorithm that is suitable for sparsification.

Definition 1 (sparsification operator)**.** *For an ADP program with nonterminals A and B and a choice function h we define*

$$B \setminus_h A := \{B - B' \text{ with } B' \text{ s.t. } h(B \uplus A) = h((B - B') \uplus A) \text{ and } |B'| \text{ is maximal}\}$$

In the equation each nonterminal implicitly represents its set of solutions on the respective part of the input.[3]*Note that there might be more than one set which satisfies the above criterion for B'. In this case, we assume that B' is selected arbitrarily among these sets.*

A new nonterminal B^c representing only the candidates among the possible parses of B can be obtained with a grammar rule $B^c \longrightarrow B \setminus_h A$. Intuitively, B^c then contains only the parses of B that are required provided that A is also considered in the recursion. By Def. 1, any rule $S \longrightarrow A|B$ with choice function h can, for example, safely be replaced by $S \longrightarrow A|B^c$. For $h(X) = \max X$ it denotes that $B^c[i,j]$ is $B[i,j]$ if $B[i,j] > A[i,j]$ and empty otherwise. But it generalizes nicely to other choice functions and also result sets of size greater one, as we will see in Sect. 5.

To implement the sparsification operator, the parser must be extended for grammar rules like $B^c \longrightarrow B \setminus_h A$. Once some table entries $A[i,j]$ and $B[i,j] = \{b_1,\ldots,b_k\}$ are computed, $B^c[i,j]$ is computed as follows. We initialize $B^c[i,j] = \emptyset$ and check sequentially for each b_x $i \le x \le k$ if $h(B^c[i,j] \uplus \{b_x,\ldots,b_k\} \uplus A[i,j]) = h(B^c[i,j] \uplus \{b_{x+1},\ldots,b_k\} \uplus A[i,j])$. If this is not the case, b_x is added to $B^c[i,j]$. If we assume that the solution sets have constant length (usually even singleton sets), this check requires constant time.

To benefit from sparsification also in terms of space complexity, B^c should be stored in some kind of sparse table data structure (and B should not be stored at all). This data structure should allow to jump from one non-empty entry to the next non-empty one in the same row or column in constant time. This is essential, since the main benefit of sparsification happens when B^c occurs in the right hand side of some other grammar rule, e.g. the rule for A in Fig. 3(a). Usually, to compute $A[i,j]$, the parser would consider each of the linearly many possible split points k to combine $S[i,k-1]$ with $B^c[k,j]$. But if B^c is sparse, all k for which $B^c[k,j]$ is empty can be discarded. The possibility to go from one non-empty entry of B^c to the next one is hence essential to reduce the number of points k. This can be achieved, for example, by representing the sparse table by lists containing the non-empty entries of each individual row and column.

[3] $B \uplus A$ creates a multiset, i.e. if some entry occurs once in both A and B it occurs twice in the result. This is important, for example, for the partition function calculation covered in Sect. 5

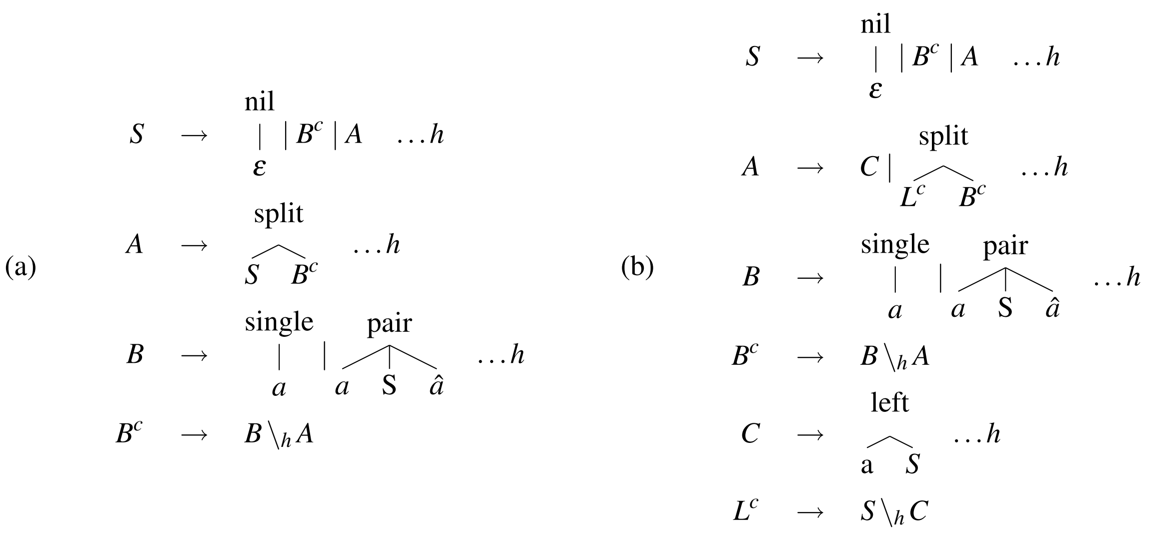

(a)
S → nil | ε | B^c | A ...h
A → split S B^c ...h
B → single a | pair a S â ...h
B^c → B _h A
(b)
S → nil | ε | B^c | A ...h
A → C | split L^c B^c ...h
B → single a | pair a S â ...h
B^c → B _h A
C → left a S ...h
L^c → S _h C

Figure 3: Sparsified Nussinov grammar with (a) OCT and (b) OCT-STEP criterion

4 Sparsified variants of the Nussinov algorithm

To keep the presentation simple, we show how to apply our sparsification operator to the Nussinov algorithm. It can be used analogously for more complex recursions.

The grammar in Fig. 2 is *semantically ambiguous* [20] in the sense that one RNA structure corresponds to several different parses: Since $A[i,j]$ can be split into $S[i,i-1]$ (of length 0) and $B[i,j]$, $B[i,j]$ can be derived from $S[i,j]$ as $S[i,j] \rightarrow B[i,j]$ or $S[i,j] \rightarrow A[i,j] \rightarrow S[i,i-1]B[i,j] \rightarrow B[i,j]$. By constraining S in the rule of A to have at least length 1 this ambiguity is eliminated. While this only avoids to consider the same part of the search space more than once, sparsification allows to completely ignore regions of the search space if it is guaranteed to find a solution outside these regions.

4.1 OCT sparsification

OCT sparsification reduces the complexity of the rule for A in the grammar of Fig. 2 by reducing the number of splits of A into some S and some B.

Definition 2 (OCT criterion [10])**.** *A subsequence from i to j is optimally co-terminus (OCT) if $i = j$ or if every optimal folding of it contains the base pair (i,j), i.e. for $h(X) = \max X$ $B[i,j] = \{b\}$ and $A[i,j] = \{a\}$ with $b > a$.*

One can show that there is an optimal split point k for $A[i,j]$ such that $B[k,j]$ is OCT [10]. This means, that we only have to examine all split points for $A[i,j]$ where the second fragment is OCT. This leads to the grammar shown in Fig. 3(a). We show now (for $h(X) = \max X$) that $B^c[i,j]$ contains exactly the OCT fragments. Since A has at least length 2, for each i $B^c[i,i] = B[i,i]$ follows directly from Def. 1. So let us consider the case where $B[i,j]$ has at least length 2. If there exists a parse for $A[i,j]$ which has a higher or equal score than $B[i,j]$ it holds: $h(B \uplus A) = h(A) = h((B - B') \uplus A)$ for $B' = B$. Hence, $B^c[i,j]$ is empty if $A[i,j] \geq B[i,j]$. Analogously, if $A[i,j] < B[i,j]$ then $h(B \uplus A) = h(B) \neq h(A)$ implies $h(B - B') = h(B)$ and consequently $B^c[i,j] = B[i,j]$.

4.2 OCT-STEP sparsification

The Nussinov algorithm can be sparsified even more by using the STEP criterion.

Definition 3 (STEP criterion [10])**.** *A subsequence from i to j is called a step sequence, if in every optimal folding the base i is paired.*

One can show that there is an optimal split point k for $A[i,j]$ such that either $k = i+1$, or $S[i,q-1]$ is a STEP and $B[q,j]$ is an OCT sequence [10]. For the split point $k = i+1$, we need to introduce the new rule C. Furthermore we can use $C[i,j]$ to create a sparse table L^c (for left paired), which contains the STEP entries. The grammar is displayed in Fig. 3(b). We show now (for $h(X) = \max X$) that a subsequence from i to j is stored in $L^c[i,j]$ if and only if it is a STEP sequence. If there exists a parse for $C[i,j]$ which has a higher or equal score than $S[i,j]$ it holds: $h(S \uplus C) = h(C) = h((S-L') \uplus C)$ for $L' = S$. Hence, $L^c[i,j]$ is empty if $C[i,j] \geq S[i,j]$. If $S[i,j] > C[i,j]$ then $h(S \uplus C) = h(S) = h(S-L') \neq h(C)$ implies $L' = \emptyset$ and therefore $L^c[i,j] = S[i,j]$. A comparison of the efficiency of OCT and OCT-STEP variants is available in the appendix.

5 Sparsification for advanced scoring schemes

So far, sparsification has only been applied to simple maximization and minimization problems. However, Def. 1 generalizes naturally to more complex scoring schemes. In the grammars of Fig. 3 we can, for example, set $h(X) = \max_k X$ to enumerate the multiset of scores of the k best solutions. In this case, by Def. 1 $B^c[i,j]$ contains exactly the scores of the *sub*optimally co-terminus structures in the sense that the structure is among the k best structures and contains the base pair (i,j). This is exactly what we need to keep the tables as sparse as possible but still to be able to reconstruct the k best solutions using the standard back-tracing techniques. A practical evaluation comparing the sparseness for finding the best solution to finding the k best solutions can be found in the appendix.

Another important choice function is $h(X) = \sum_{x \in X} x$ which is required to compute the partition function. Partition function computation is not sparse, as each structure contributes to the sum (except if it has score 0) and hence cannot be ignored. Our ADP sparsification operator automatically shows this behavior: For $h(X) = \sum_{x \in X} x$, Def. 1 implies that B' can only contain entries with a score of 0. Hence, the sparsified grammars can also be used for partition function calculation, but the resulting computations are not expected to be sparse, since the problem itself is not sparse. One can also approximate the partition function by rounding scores with a negligible contribution to 0. In this case, those parts of the search space would again automatically become sparse in the implementation.

6 Conclusions

We presented the first systematic approach to describe the existing approaches of sparsification in dynamic programming in terms of a general framework. Our sparsification operator is able to express different sparsification criteria including the OCT and STEP criteria or any other criterion which is based on checking whether certain solutions B are "still required when" considering some other solutions A. Although we have discussed this here for the Nussinov algorithm, it generalizes to more complex algorithms for RNA structure prediction [8], simultaneous alignment and folding [9], RNA-RNA-interaction prediction [11], and the prediction of RNA pseudoknot structures [12]. This work also shows how to generalize these approaches to the enumeration of suboptimal solutions or the approximation of the partition function.

Since our prototypical Haskell implementation is not intended for efficient implementations, the next important step will be to integrate the theory into the current ADP to C++ compiler Bellman's GAP [15]. Concerning the sparsification of advanced scoring schemes, the next interesting question is whether an approximated partition function can still be used, for example, to get a reasonable approximation of the base pair probabilities. Another interesting question is whether there exist general techniques to transform regular grammars to sparse equivalent grammars.

Acknowledgments We are grateful to Rolf Backofen, Robert Giegerich, and Michal Ziv-Ukelson for their helpful comments. This work is partially supported by DFG grant BA 2168/3-1.

References

[1] Graham, S.L., Harrison, M.A., Ruzzo, W.L.: An improved context-free recognizer. ACM Transactions on Programming Languages and Systems **2**(3) (1980) 415–462

[2] Lifshits, Y., Mozes, S., Weimann, O., Ziv-Ukelson, M.: Speeding up HMM decoding and training by exploiting sequence repetitions. Algorithmica **54**(3) (2009) 379–399

[3] Frid, Y., Gusfield, D.: A simple, practical and complete O(n3/log n)-time algorithm for RNA folding using the Four-Russians speedup. Algorithms Mol Biol **5** (2010) 13

[4] Valiant, L.: General context-free recognition in less than cubic time. Journal of Computer and System Sciences **10** (1975) 308–315

[5] Akutsu, T.: Approximation and exact algorithms for RNA secondary structure prediction and recognition of stochastic context-free languages. Journal of Combinatorial Optimization **3** (1999) 321–336

[6] Zakov, S., Tsur, D., Ziv-Ukelson, M.: Reducing the worst case running times of a family of RNA and CFG problems, using valiant's approach. In Moulton, V., Singh, M., eds.: Proc. of the 10th Workshop on Algorithms in Bioinformatics (WABI). Volume 6293 of Lecture Notes in Computer Science., Springer Berlin / Heidelberg (2010) 65–77

[7] Eppstein, D., Galil, Z., Giancarlo, R., Italiano, G.F.: Sparse dynamic programming I: linear cost functions. J. ACM **39**(3) (1992) 519–545

[8] Wexler, Y., Zilberstein, C., Ziv-Ukelson, M.: A study of accessible motifs and RNA folding complexity. Journal of Computational Biology **14**(6) (2007) 856–72

[9] Ziv-Ukelson, M., Gat-Viks, I., Wexler, Y., Shamir, R.: A faster algorithm for RNA co-folding. In Crandall, K.A., Lagergren, J., eds.: WABI 2008. Volume 5251 of Lecture Notes in Computer Science., Springer (2008) 174–185

[10] Backofen, R., Tsur, D., Zakov, S., Ziv-Ukelson, M.: Sparse RNA folding: Time and space efficient algorithms. J. Discrete Algorithms **9**(1) (2011) 12–31

[11] Salari, R., Möhl, M., Will, S., Sahinalp, S., Backofen, R.: Time and space efficient RNA-RNA interaction prediction via sparse folding. In Berger, B., ed.: Proc. of RECOMB 2010. Volume 6044 of Lecture Notes in Computer Science., Springer Berlin / Heidelberg (2010) 473–490

[12] Mohl, M., Salari, R., Will, S., Backofen, R., Sahinalp, S.C.: Sparsification of RNA structure prediction including pseudoknots. Algorithms Mol Biol **5**(1) (2010) 39

[13] Giegerich, R., Meyer, C., Steffen, P.: A discipline of dynamic programming over sequence data. Sci. Comput. Program. **51**(3) (2004) 215–263

[14] Giegerich, R., Steffen, P.: Challenges in the compilation of a domain specific language for dynamic programming. In: Proceedings of the 2006 ACM symposium on Applied computing. SAC '06, New York, NY, USA, ACM (2006) 1603–1609

[15] Sauthoff, G., Janssen, S., Giegerich, R.: Bellman's GAP - A Declarative Language for Dynamic Programming. In: 13th International ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming. PPDP 2011, ACM (2011)

[16] Steffen, P., Voss, B., Rehmsmeier, M., Reeder, J., Giegerich, R.: RNAshapes: an integrated RNA analysis package based on abstract shapes. Bioinformatics **22**(4) (2006) 500–3

[17] Reeder, J., Steffen, P., Giegerich, R.: pknotsRG: RNA pseudoknot folding including near-optimal structures and sliding windows. Nucleic Acids Research **35**(Web Server issue) (2007) W320–4

[18] Rehmsmeier, M., Steffen, P., Hochsmann, M., Giegerich, R.: Fast and effective prediction of microRNA/target duplexes. RNA **10**(10) (2004) 1507–17

[19] Nussinov, R., Pieczenik, G., Griggs, J.R., Kleitman, D.J.: Algorithms for loop matchings. SIAM Journal on Applied Mathematics **35**(1) (July 1978) 68–82

[20] Reeder, J., Steffen, P., Giegerich, R.: Effective ambiguity checking in biosequence analysis. BMC Bioinformatics **6** (2005) 153

7 Appendix

7.1 Evaluation for the sparsified variants of the Nussinov algorithm

In order to evaluate the effectiveness of the sparsification approaches mentioned in Sect. 4, we extended the existing Haskell implementation of ADP [13] with our sparsification operator. Note that the Haskell implementation of ADP is intended as a flexible and extensible platform for quick experiments and not optimized for performance. Therefore, this section is only a feasibility evaluation for which we used the Haskell interpreter hugs to measure the number of split points of the different Nussinov variants. For implementations of practical relevance, an analogous extension of the ADP to C++ compiler Bellman's GAP [15] is required (which we have planned as future work).

Based on the extension of the Haskell ADP system, we implemented the different Nussinov variants in order to measure their degree of sparseness. We took 100 random sequences of length 100 and 200 and folded them using the non-sparse Nussinov variant, the OCT variant and the OCT-STEP variant. The average number of split points required on a single 2.3 Ghz Opteron 2356 processor with 4 GB memory is given in Table 1. Also the average proportion of split points examined by the sparse variants compared to the original Nussinov algorithm are shown. One can see that the proportion of split points analyzed by the sparse variants compared to the original Nussinov algorithm decreases with increasing sequence length. The OCT-STEP variant is slightly more sparse than the OCT variant. Note that we compare our sparse variants to the original Nussinov algorithm with a recursion $A \rightarrow SB$ which hence considers only split points where the outermost bases of B can pair. The evaluation done in [10] compares the sparse variants to a basic variant with significantly more split points due to a recursion of the form $A \rightarrow SS$. We also compared our implementation to this variant and could confirm the results reported in [10].

Table 1: Number of split points for the different Nussinov variants for random sequences of different length.

Grammar	average number of split points over 100 random sequences	
	of length 100	of length 200
Nussinov, Fig. 2	65288	510473
NussinovOCT, Fig. 3(a)	10620 (16.3 %)[1]	53947 (10.6 %)[1]
NussinovOCTSTEP, Fig. 3(b)	9430 (14.4 %)[1]	43679 (8.6 %)[1]

[1] average proportion of split points examined by the sparse variants compared to the original Nussinov algorithm

7.2 Evaluation for enumerating suboptimal solutions

As explained in Sect. 5, sparsification can also be applied to advanced scoring schemes as, for example, enumerating the best k suboptimal solutions. To measure how effective sparsification is in this scenario, we counted the number of split points examined during bottom-up parsing. So if a split point of a fragment s and a fragment b is considered, k^2 instances of the split point are examined, as k suboptimal solutions need to be stored for each fragment in the original Nussinov algorithm. When sparsification is applied, the number of suboptimal solutions that need to be stored can be reduced. We computed for 100 random sequences of length 50 the average number of split points examined during the evaluation of suboptimal solutions. The results can be found in Table 2. We also show the average proportion of split points examined by the sparse variants compared to the original Nussinov algorithm. One can see that the proportion of split points analyzed by the sparse variants compared to the original Nussinov algorithm decreases with increasing k.

Table 2: Number of split points for the different Nussinov variants for enumerating suboptimal solutions

Grammar	average number of split points over 100 random sequences		
	$k = 1$	$k = 5$	$k = 10$
Nussinov, Fig. 2	8573	128461	416778
NussinovOCT, Fig. 3(a)	2277 (26.6 %)[1]	26032 (20.3 %)[1]	77986 (18.7 %)[1]
NussinovOCTSTEP, Fig. 3(b)	2124 (24.8 %)[1]	17576 (13.7 %)[1]	49595 (11.9 %)[1]

[1]average proportion of split points examined by the sparse variants compared to the original Nussinov algorithm