Volume 9, issue 1
March 2011
ISSN 1570-8667

ELSEVIER

JOURNAL OF DISCRETE ALGORITHMS

20th Anniversary Edition of the Annual Symposium on Combinatorial Pattern Matching (CPM 2009)
Guest Editors
Gregory Kucherov
Esko Ukkonen

Available online at
ScienceDirect
www.sciencedirect.com

Journal of Discrete Algorithms 9 (2011) 12–31

Contents lists available at ScienceDirect

Journal of Discrete Algorithms

www.elsevier.com/locate/jda

Sparse RNA folding: Time and space efficient algorithms

Rolf Backofen[a], Dekel Tsur[b], Shay Zakov[b,*], Michal Ziv-Ukelson[b]

[a] *Albert Ludwigs University, Freiburg, Germany*
[b] *Department of Computer Science, Ben-Gurion University of the Negev, Israel*

ARTICLE INFO

Article history:
Available online 17 September 2010

Keywords:
RNA folding
Time complexity
Space complexity
Dynamic programming
Sparsification

ABSTRACT

The currently fastest algorithm for *RNA Single Strand Folding* requires $O(nZ)$ time and $\Theta(n^2)$ space, where n denotes the length of the input string and Z is a sparsity parameter satisfying $n \leqslant Z < n^2$. We show how to reduce the time and space complexities of this algorithm in the sparse case. The space reduction is based on the observation that some solutions for sub-instances are not examined after a certain stage of the algorithm, and may be discarded from memory. The running time speed up is achieved by combining two independent sparsification criteria, which restrict the number of expressions that need to be examined in bottleneck computations of the algorithm. This yields an $O(n^2 + PZ)$ time and $\Theta(Z)$ space algorithm, where P is a sparsity parameter satisfying $P < n \leqslant Z \leqslant n(P+1)$. For the base-pairing maximization variant, the time complexity is further reduced to $O(LZ)$, where L denotes the maximum number of base-pairs in a folding of the input string and satisfies $L \leqslant n\backslash 2$.

The presented techniques also extend to the related *RNA Simultaneous Alignment and Folding* problem. For an input composed of two strings of lengths n and m, the time and space complexities are reduced from $O(nm\tilde{Z})$ and $\Theta(n^2m^2)$ down to $O(n^2m^2 + \tilde{P}\tilde{Z})$ and $\Theta(nm^2 + \tilde{Z})$ respectively, where $\tilde{Z}$ and $\tilde{P}$ are sparsity parameters satisfying $\tilde{P} < nm \leqslant \tilde{Z} < nm(\tilde{P}+3)$.

A preliminary extended abstract of this work previously appeared in Backofen et al. (2009) [5]. Code implementations (in Java) may be downloaded from: http://www.cs.bgu.ac.il/~zakovs/RNAfold/SparseFold.zip.

1. Introduction

The structure of RNA is evolutionarily more conserved than its sequence and is thus key to its functional analysis [7]. Unfortunately, although massive amounts of sequence data are continuously generated, the number of known RNA structures is still relatively limited, since experimental methods, such as NMR and Crystallography, require expertise and long experimental time. Therefore, computational methods for predicting RNA structures are of significant value [40,21,39].

RNA is typically produced as a single stranded molecule, which then folds upon itself to form a number of short base-paired helices. This base-paired structure is called the *secondary structure*, or the *folding* of the RNA molecule. Secondary structures rarely contain pseudoknots (i.e. crossing base pairs). Under the assumption that the structure does not contain pseudoknots, a model was proposed by Tinoco et al. [31] to estimate the stability (in terms of free energy) of a folded RNA molecule by summing all contributions from the stabilizing, consecutive base pairs and from the loop-destabilizing terms in the secondary structure. Based on this model, dynamic programming algorithms were suggested for estimating

* Corresponding author.
E-mail addresses: backofen@informatik.uni-freiburg.de (R. Backofen), dekelts@cs.bgu.ac.il (D. Tsur), zakovs@cs.bgu.ac.il (S. Zakov), michaluz@cs.bgu.ac.il (M. Ziv-Ukelson).

1570-8667/$ – see front matter © 2010 Elsevier B.V. All rights reserved.
doi:10.1016/j.jda.2010.09.001

Table 1
Time and space complexities of RNA folding algorithms. For SSF variants, n denotes the length of the input string. For SAF, n and m denote the lengths of the two input strings. $D(n)$ stands for the time complexity of computing the distance product of two $n \times n$ matrices, where the best current bound on $D(n)$ is $O(n^3 \log^3 \log n / \log^2 n)$ [8]. The sparsity parameters are bounded as follows: for SSF, $L \leqslant n/2$, $P < n \leqslant Z \leqslant n(P+1)$, and for SAF, $\tilde{P} < nm \leqslant \tilde{Z} < nm(\tilde{P}+3)$.

	Previous results		Our results	
	Time	Space	Time	Space
SSF base-pairing maximization	$\Theta(n^3)$ [26] $O(nZ)$ [34] $\Theta(D(n))$ [1]	$\Theta(n^2)$	$O(LZ)$	$\Theta(Z)$
SSF energy minimization	$\Theta(n^3)$ [41] $O(nZ)$ [34] $\Theta(D(n))$ [1]	$\Theta(n^2)$	$O(n^2 + PZ)$	$\Theta(Z)$
SAF	$\Theta(n^3m^3)$ [28] $O(nm\tilde{Z})$ [38] $\Theta(D(nm))$ [37]	$\Theta(n^2m^2)$	$O(n^2m^2 + \tilde{P}\tilde{Z})$	$\Theta(nm^2 + \tilde{Z})$

the most stable structures [33,26,41,1,34], applying various scoring criteria such as the maximal number of base pairs [26] or the minimal free energy [41]. This optimization problem is denoted here as the *RNA Single Strand Folding* (SSF) problem, and the time and space complexities of the classical algorithms for solving it are $\Theta(n^3)$ and $\Theta(n^2)$, respectively, where n denotes the length of the input RNA string. Recently, these results were sped up to yield $O(nZ)$ time and $\Theta(n^2)$ space algorithms [34], where Z is a sparsity parameter that satisfies $n \leqslant Z < n^2$. On a more theoretical front, Akutsu suggested an $\Theta(D(n))$ time and $\Theta(n^2)$ space algorithm for this problem [1], where $D(n)$ is the time for computing the distance product of two $n \times n$ matrices. The best current bound on $D(n)$ is $O(n^3 \log^3 \log n / \log^2 n)$ [8]. An additional technique which also allows to obtain a slightly sub-cubic running time was presented by Frid and Gusfield [14], resulting with an $\Theta(n^3 / \log n)$ running time algorithm.

Another approach to RNA folding prediction is *RNA Simultaneous Alignment and Folding* (SAF) [28,25,18,38,35]. This approach consists of finding an optimal alignment between a set of RNA strings, where an alignment score is evaluated with respect to some common folding of the input strings. However, as stated in [16], even for the simple case where the input consists of only two strings, this approach requires “extreme amounts of memory and space” with time complexity of $\Theta(n^3m^3)$ and space complexity of $\Theta(n^2m^2)$, where n and m are the lengths of the input RNA strings to be aligned. Thus, most existing practical implementations of this algorithm [25,18,35] use restricted versions of the original problem. Since these restrictions introduce another source of error, it is of utmost practical importance to the research on RNA to improve both the space and time complexities of the full version of SAF. A first non-heuristic speedup, which does not sacrifice the optimality of results, was recently described in [38]. This work extends the approach of [34] and yields an $O(nm\tilde{Z})$ time and $\Theta(n^2m^2)$ space algorithm for the SAF problem, where $\tilde{Z}$ is a sparsity parameter that satisfies $nm \leqslant \tilde{Z} < n^2m^2$. However, experimental analysis of this algorithm indicates that the high memory requirements is a major bottleneck in practice, both in constraining the lengths of the input strings, as well as in exhausting the benchmark machine's memory, which in turn results in a page-fault slowdown (for example, for $n = m = 300$, the data structure requires about 8 Gigabyte of memory). Theoretical approaches for improving the running time of the SAF problem were recently presented [37,15]. Nevertheless, these methods are not expected to yield a significant improvement to the running time in practice, and do not improve the space complexity of the original algorithm.

Our contribution

The main contributions in this paper are as follows.

(1) *Reducing the space requirements of the SSF problem in the sparse cases*. The space requirement reduction is based on the observation that some solutions for subproblems are not examined after a certain stage of the algorithm, and may be discarded from memory. This yields an $O(nZ)$ time and $\Theta(Z)$ space algorithm for this problem (Section 3).
(2) *Reducing the time complexity of the SSF problem in the sparse cases*. We describe a faster algorithm that exploits an additional sparsity parameter P, satisfying $P < n \leqslant Z \leqslant n(P+1)$. By combining *forward dynamic programming* (previously used in [17,32,22] for related problems) with the utilization of the triangle inequality property [34], we reduce the number of sub-instance pairs that need to be considered by the algorithm and obtain an $O(n^2 + PZ)$ time and $\Theta(Z)$ space algorithm (Section 4.1). For the base-pairing maximization variant of the problem we show that $P = L \leqslant n/2$, where L denotes the maximum cardinality of a folding of the input string, and further reduce the running time to $O(LZ)$ (Section 4.2).
(3) *Extending the time and space complexity reductions to the SAF problem*. The presented sparsification techniques are adapted to the SAF problem (Section 5). The time and space complexities of the sparse algorithm are $O(n^2m^2 + \tilde{P}\tilde{Z})$ and $\Theta(nm^2 + \tilde{Z})$, respectively, where $\tilde{P} \leqslant nm \leqslant \tilde{Z} \leqslant nm(\tilde{P}+3)$.

In addition to the optimal folding score computation, we show a trace-back procedure which outputs a corresponding optimal secondary structure. Note that it is an interesting challenge in its own right to recover an optimal folding within the time and space complexity bounds of the space-reduced algorithm, since due to the sparse representation only partial information is kept. The presented strategy may be applied to various scoring schemes, including the *base pairing maximization* and *free energy minimization* scoring schemes. The improved complexities are summarized in Table 1.

We note that the algorithms described here are practical in the sense that the hidden constants are small (i.e. no complex data structures or subroutines are used, which might imply a long running time in practice). In the context of practical contribution, we also point out that our space complexity improvements are more significant than the time complexity improvements, since both Z and $\tilde{Z}$ were experimentally shown to be significantly less than n^2 and n^2m^2, respectively [34,38]. Furthermore, reducing the space complexity of the SAF problem is a key result in practice, as in the previous results the space complexity typically dictated the computational bottleneck [16,38].

RoadMap: The rest of the paper proceeds as follows. Preliminary notation and definitions are given in Section 2. Section 3 presents the basic dynamic programming algorithm for the SSF problem, and shows how to improve its space complexity. Section 4 introduces an additional sparsity property of the SSF domain, and shows how to utilize this property in order to speed up the algorithm. In Section 5, the presented modifications are scaled up to the SAF domain. Section 6 presents experimental results for the base-paring maximization variant of the SSF problem, and a concluding discussion summarizes the paper in Section 7.

2. Preliminaries

RNA is typically produced as a single stranded molecule, composed as a sequence of *bases* (or *nucleotides*), which then folds upon itself to a *structural conformation*. This structural conformation is stabilized by hydrogen bonds which are formed between different bases in the sequence. There are four types of bases, denoted by the letters A, C, G, and U. Every base in the sequence can form a bond with at most one other base, where bases of type C typically pair with bases of type G, A typically pairs with U, and another weaker pairing can occur between G and U. Two bases which can pair with each other are called *complementary bases*, and the bond which is formed between two complementary bases is called a *base-pair*. The set of formed base-pairs is called the *secondary structure*, or the *folding* of the RNA molecule, as apposed to the *tertiary structure* which is the actual three-dimensional molecule structure. Paired bases almost always occur in a nested fashion in RNA secondary structures. Informally, this means that if we draw arcs connecting base pairs over an RNA sequence, none of the arcs cross each other. When non-nested base pairs occur, they are called *pseudoknots*. In this work, we consider only pseudoknot-free foldings. In what follows we give the basic formal definitions and notations for the *RNA Single Strand Folding* problem (SSF).

An SSF instance is an RNA string $S = s_1 s_2 \cdots s_n$, over the alphabet of base types $\{A, C, G, U\}$. Denote by $|S|$ the length of S, and call an instance S with $|S| = 0$ an *empty instance*. Denote by $S_{i,j}$ the substring (or sub-instance) of S between indices i and j (inclusive), where $S_{i,i-1}$ is defined to be an empty instance. For two sub-instances $S_{i,j}$ and $S_{k,l}$, say that $S_{k,l}$ is a sub-instance of $S_{i,j}$ if $i \leqslant k \leqslant l \leqslant j$, where it is a *strict* sub-instance if $S_{k,l} \neq S_{i,j}$.

Definition 1. A *folding* F of a sub-instance $S_{i,j}$ is a set of index pairs that satisfies the following:

1. For every $(k,l) \in F$, $i \leqslant k < l \leqslant j$.
2. There are no $(k,l), (k',l') \in F$, such that $k \leqslant k' \leqslant l \leqslant l'$.

Say that an index k is *paired* in a folding F if k appears in a base-pair in F, otherwise k is *unpaired* in F.

Let q be an index such that $i < q \leqslant j$. Say that a folding F of $S_{i,j}$ *admits the split-point* q if for every $(k,l) \in F$, either $l < q$ or $k \geqslant q$ (see Fig. 1). In other words, a folding F admits the split-point q if q implies a partition of F into two independent foldings: one for the non-empty prefix sub-instance $S_{i,q-1}$, and one for the non-empty suffix sub-instances $S_{q,j}$. Denote by $Q_{i,j} = \{q\colon\ i < q \leqslant j\}$ the set of all possible *split-points* with respect to $S_{i,j}$.

We distinguish between two kinds of foldings. A folding F is called *partitionable* with respect to $S_{i,j}$ if it admits at least one split-point in $Q_{i,j}$. Otherwise, F is called *co-terminus* with respect to $S_{i,j}$. Observe that in the case where $Q_{i,j} = \emptyset$ (i.e. $j = i$ or $j = i - 1$) the only possible folding is the empty folding, and it is a co-terminus folding for such instances by definition. In the case where $Q_{i,j} \neq \emptyset$ (i.e. $j > i$), a folding is co-terminus if and only if it contains the base pair (i,j): On one hand, a folding containing (i,j) clearly admits no split-point in $Q_{i,j}$. On the other hand, if j is unpaired in F then F admits the split-point j, and if F contains a base-pair (q,j) for some $i < q < j$, then F admits the split-point q (Fig. 1).

An SSF *scoring scheme* $\textsc{score}$ is a function which evaluates the qualities of foldings with respect to instances, where $\textsc{score}(S,F) > \textsc{score}(S,F')$ implies that the folding F is of better quality than F', with respect to the instance S. Next, we formally define the SSF problem.

Problem 1 *(SSF).* Let $\textsc{score}$ be some SSF scoring scheme, and S an SSF instance with $|S| = n$. Let $D_{i,j}$ denote the set of all possible foldings of the sub-instance $S_{i,j}$.

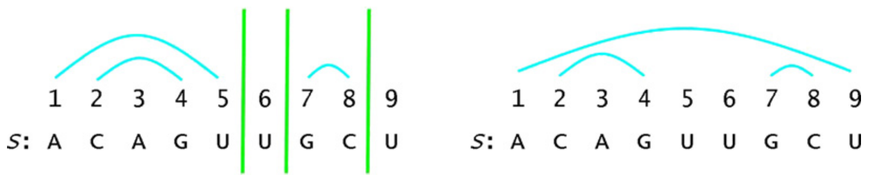

Fig. 1. Example of foldings. The left folding is a *partitionable* folding, admitting the split-points 6, 7, and 9. The right folding is a *co-terminus* folding, containing the base-pair (1, 9) and admitting no split-points.

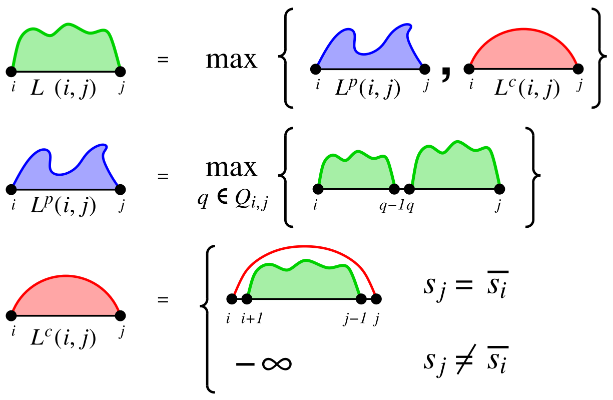

Fig. 2. A schematic illustration of the recursion of Eqs. (2.1), (2.2), and (3.1) (L^c is illustrated with respect to the base-pairing maximization scoring scheme, where the notation $s_j = \bar{s_i}$ implies that the bases s_i and s_j are complementary).

- The *SSF-optimization* problem is to calculate $L(1, n)$, where
$$L(i, j) = \max_{F \in D_{i,j}} \{\text{SCORE}(S_{i,j}, F)\}.$$
- The *SSF-search* problem is to find a folding F^* of S that satisfies $\text{SCORE}(S, F^*) = L(1, n)$ (call such folding *optimal* under the scoring scheme SCORE).

We call $L(i, j)$ the *solution* for the sub-instance $S_{i,j}$. We also distinguish between co-terminus and partitionable solutions, where $L^c(i, j)$ and $L^p(i, j)$ denote the maximum scores (or the solutions) of a co-terminus and a partitionable folding of the sub-instance $S_{i,j}$, respectively. Since every folding is either co-terminus or partitionable, we get the following equation:

$$L(i, j) = \max\{L^c(i, j), L^p(i, j)\}. \tag{2.1}$$

Many of the current popular folding prediction tools follow the same algorithmic framework. These tools use dynamic programming for computing solutions for all sub-instances of the input instance S in a bottom-up manner, and then apply a trace-back procedure over the dynamic programming table in order to report one or more optimal (or sub-optimal) foldings of S. The tools differ mainly in the scoring schemes they define, as well as in some implementation aspects, though the algorithms they apply are similar from a combinatorial point of view. Typical scoring schemes are based on folding free energy [30,41,21], and on reductions to stochastic context free grammar parsing and parameter learning approaches [27,11,12,3].

Note that for small *trivial* instances $S_{i,j}$ with $j = i$ or $j = i - 1$, $Q_{i,j} = \emptyset$ and thus such instances have no partitionable folding by definition. We define in such cases that $L^p(i, j) = -\infty$. The only folding for such instances is the empty folding which contains no base-pairs, and by definition it is considered a co-terminus folding with respect to these instances. We assume that for trivial instances, the solution $L(i, j) = L^c(i, j)$ can be computed explicitly in constant time, and for instances with $j > i$, the scoring scheme sustains the following recursive equations:

$$L^c(i, j) = f\big(L(i + 1, j - 1), s_i, s_j\big), \tag{2.2}$$

$$L^p(i, j) = \max_{q \in Q_{i,j}} \{L(i, q - 1) + L(q, j)\}, \tag{2.3}$$

where f is a function that can be evaluated in constant time. That is, the co-terminus solution of $S_{i,j}$ can be obtained in constant time from the solution of the internal sub-instance $S_{i+1,j-1}$, combined with some local score expression due to the addition of the base-pair (i, j). The partitionable solution can be obtained by examining all possible split-points $q \in Q_{i,j}$, and summing the solutions of the prefix $S_{i,q-1}$ and the suffix $S_{q,j}$. Fig. 2 illustrates the recursive score computation of Eqs. (2.1), (2.2), and (2.3).

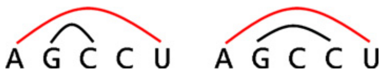

Fig. 3. An OCT instance. Both optimal foldings of the presented string pair between the first and last bases of the string.

A simple example for a scoring scheme that follows the above equations is the *base pairing maximization* (*bpm*) scoring scheme [26], which aims to find foldings with maximum number of base-pairs between complementary bases and no base-pairs between non-complementary bases. Here, the computation of the co-terminus score is given by

$$L^c(i,j) = \begin{cases} L(i+1, j-1) + 1, & s_j \text{ and } s_i \text{ are complementary}, \\ -\infty, & \text{otherwise}. \end{cases} \tag{2.4}$$

Most scoring schemes follow similar recursive score computations (see the discussion in Section 7), and all techniques presented in this paper, except for the "step encoding" technique of Section 4.2, may be easily extended to more realistic scoring schemes by applying minor adjustments. The core property which yields the presented computational improvements is the (*inverse*) *triangle inequality property*, which is common to most scoring schemes, thus allowing to reduce the complexity in a family of RNA folding algorithms. This property is defined below:

Property 1 *(Triangle inequality). A scoring scheme sustains the triangle inequality property if for every sub-instance $S_{i,j}$ and for every split-point $q \in Q_{i,j}$, $L(i,j) \geqslant L(i,q-1) + L(q,j)$.*

Observe that any scoring scheme that follows Eqs. (2.1) and (2.3), sustains the triangle inequality property. In the rest of this paper, we use L instead of $L(1,n)$ whenever the context is clear.

3. Space efficient algorithm for SSF

In this section we show how to reduce the space requirement of algorithms for the SSF problem. We do so by observing that a property that was previously suggested for reducing the time complexity of SSF algorithms, may also be utilized to reduce their space complexity. Section 3.1 describes the technique to reduce the time complexity of the dynamic programming implementation of the recursive computation, due to [34]. Section 3.2 shows how to extend this technique to reduce also the space requirement of the algorithm by sparsifying the dynamic programming table. Section 3.3 shows how to solve the search problem by tracing-back the sparse dynamic programming table.

3.1. Using OCT sub-instances for time complexity reduction

Note that the time complexity bottleneck in algorithms which implement the recursive computation of Eqs. (2.1), (2.2), and (2.3) is dictated by the consideration of $O(n)$ split-points q in the computation of $L^p(i,j)$, according to Eq. (2.3). In this section, as well as in Section 4, we describe techniques that reduce the number of split-points that need to be considered in the computation of $L^p(i,j)$, and thus improve the time complexity of such algorithms.

Based on the triangle inequality property, Wexler et al. [34] observed that it is sufficient to examine only a subset of the split-points in order to compute $L^p(i,j)$. We present here a slightly different notation for the same concept, and supply a proof for the main claim in [34] for completeness.

Definition 2 *(OCT).* A sub-instance $S_{i,j}$ is *optimally co-terminus* (OCT) if every optimal folding of $S_{i,j}$ is co-terminus (that is, if $L(i,j) = L^c(i,j) > L^p(i,j)$, see Fig. 3).

Note that any sub-instance of length 1 is an OCT by definition, since it has no partitionable folding. For a sub-instance $S_{i,j}$ with $j > i$, call a split-point $q \in Q_{i,j}$ for which $L^p(i,j) = L(i,q-1) + L(q,j)$, an *optimal split-point* with respect to $S_{i,j}$.

Lemma 1. *(See Wexler et al. [34].) For every sub-instance $S_{i,j}$ with $j > i$, there is an optimal split-point $q \in Q_{i,j}$ such that $S_{q,j}$ is an OCT.*

Proof. Let q be an optimal split-point with respect to $S_{i,j}$ such that $|S_{q,j}|$ is minimal. If $q = j$, then $S_{q,j}$ is an OCT. Otherwise, let q' be any index in $Q_{q,j}$. From the selection of q and since $|S_{q',j}| < |S_{q,j}|$, it follows that $L(i,q-1) + L(q,j) > L(i,q'-1) + L(q',j)$. From the triangle inequality property we have that $L(i,q'-1) \geqslant L(i,q-1) + L(q,q'-1)$. Therefore, $L(q,j) > L(q,q'-1) + L(q',j)$ for every $q' \in Q_{q,j}$, and since $L^p(q,j) = \max_{q' \in Q_{q,j}}\{L(q,q'-1) + L(q',j)\} < L(q,j)$, it follows that $S_{q,j}$ is an OCT. $\square$

Define the following subset of split-points with respect to $S_{i,j}$:

$$Q^{oct}_{i,j} = \{q \in Q_{i,j}\colon\ S_{q,j} \text{ is an OCT}\}.$$

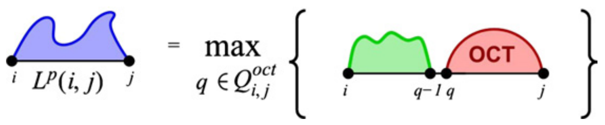

Fig. 4. A schematic illustration of Eq. (3.1). Only split-points that induce OCT suffixes are examined.

The following equation restates Eq. (2.3), based on Lemma 1, by restricting the split-points considered by the maximization term to those in $Q^{oct}_{i,j}$. Fig. 4 illustrates the modified computation of L^p.

$$L^p(i,j) = \max_{q \in Q^{oct}_{i,j}} \{L(i,q-1) + L(q,j)\}. \tag{3.1}$$

We define the following sparsity measure of RNA strings.

Definition 3. For an SSF instance S, $Z(S)$ is the number of substrings of S which are OCTs.

In the rest of this paper, we use Z instead of $Z(S)$ whenever the context is clear. In the sparse case, only a small portion of all $\Theta(n^2)$ substrings of S are OCTs. Since every sub-instance of length 1 is an OCT, $Z \geqslant n$. For the minimum free energy scoring scheme, an estimation of the expected value of a parameter related to Z can be found in [34]. This estimation is based on a probabilistic model for polymer folding and measured by simulations, and it shows that on average Z is significantly smaller than $\Theta(n^2)$.

The standard dynamic programming algorithm for SSF performs a bottom-up computation of the recurrence. The algorithm computes the upper triangle of a table $M_{n\times n}$, where each cell $M[i,j]$ stores the corresponding value $L(i,j)$. The entries of M are traversed in an order which guarantees that all values that are needed for the computation of $M[i,j]$ according to the recurrence formula are computed and stored in M prior to the computation of $M[i,j]$. Upon termination, $M[1,n]$ holds the value $L(1,n)$. This algorithm requires $\Theta(n^2)$ space in order to maintain the table M. For each of the $\Theta(n^2)$ sub-instances $S_{i,j}$, the time complexity for computing the value of the corresponding entry $M[i,j]$ is dictated by the computation of $L^p(i,j)$. If this computation is conducted according to Eq. (2.3), $\Theta(n)$ operations are performed on average, and thus the total time complexity is $\Theta(n^3)$. Using the improvement of [34], the computation takes $\Theta(|Q^{oct}_{i,j}|)$, and the total time complexity reduces to $O(nZ)$, since

$$\sum_{i=1}^{n-1}\sum_{j=i+1}^{n}|Q^{oct}_{i,j}| \leqslant \sum_{i=1}^{n-1}\sum_{j=i+1}^{n}|Q^{oct}_{1,j}| \leqslant \sum_{i=1}^{n-1} Z < nZ.$$

3.2. Reducing the space complexity of the optimization problem

Our space reduction strategy is based on the observation that some of the values stored by the algorithm of [34] are not necessary throughout the complete run of the algorithm. In the following lemma we characterize the values that need to be maintained in memory for the computation of $L(i,j)$.

Lemma 2. *For a sub-instance $S_{i,j}$, it is possible to compute $L(i,j)$ by examining only values $L(a,b)$ for strict sub-instances $S_{a,b}$ of $S_{i,j}$, which sustain that either $a=i$, $a=i+1$, or $S_{a,b}$ is an OCT.*

Proof. Immediate from Eqs. (2.1), (2.2) and (3.1). □

Consider a dynamic programming algorithm that fills the table M by traversing its entries row by row from bottom to top, and each row from left to right. Lemma 2 implies that at the stage where $M[i,j]$ is computed, it is sufficient to keep only the values in the currently computed ith row, the values in the recently computed $(i+1)$th row, and values in entries that correspond to OCT sub-instances of S. Thus, there is no need to maintain the complete table M in memory, but rather, at each stage, entries which are guaranteed not to be further examined by the algorithm may be discarded. Such a modification may be implemented by maintaining two $\Theta(n)$ arrays for the current and last computed rows, and in addition an implementation of the sparse table M by using n OCT-lists. Each OCT-list corresponds to a column in M, and contains solutions only for OCT sub-instances. Note that these OCT-lists are always updated by adding elements to their ends, and the elements in these lists are always queried in a sequential manner. Thus, a simple implementation (such as a *Linked List* implementation) can allow that both insertion and query time for the entries in the sparse representation of M would take $O(1)$ running time. This gives a total space complexity of $\Theta(n+Z) = \Theta(Z)$. The modification does not affect the running time of the algorithm, maintaining the time complexity of $O(nZ)$.

Algorithm 1 gives the pseudo-code of the algorithm described above, and Fig. 5 illustrates its run. The time and space complexities are summarized in the following lemma.

Lemma 3. *Algorithm 1 computes $L(1,n)$ for a given instance S of length n, in $O(nZ)$ time and $\Theta(Z)$ space.*

Algorithm 1: Space efficient RNA folding

```
input : An RNA string $S = s_1 s_2 \cdots s_n$
output: $L(1, n)$
1  for $i \leftarrow n$ down to 1 do
2      set explicitly the values of $M[i, i-1]$ and $M[i, i]$ to the solutions for the corresponding trivial sub-instances;
3      mark $S_{i,i}$ as an OCT;
4      for $j \leftarrow i+1$ to $n$ do
5          set $M[i, j] \leftarrow \max_{q \in Q^{oct}_{i,j}} \{M[i, q-1] + M[q, j]\}$;
6          compute $L^c(i, j) = f(M[i+1, j-1], s_i, s_j)$;
7          if $M[i, j] < L^c(i, j)$ then
8              set $M[i, j] \leftarrow L^c(i, j)$;
9              mark $S_{i,j}$ as an OCT;
10     discard from memory the values in all entries in row $i+1$ of $M$ that do not correspond to OCT sub-instances;
11 return $M[1, n]$;
```

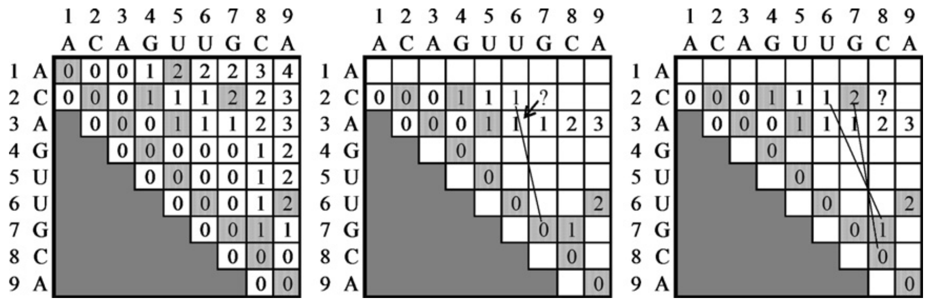

Fig. 5. An exemplification of Algorithm 1. The left figure shows the complete table M with respect to the RNA string $S = \text{ACAGUUGCA}$, and the base pairing maximization scoring scheme. Grayed entries correspond to OCT sub-instances. The figure in the middle demonstrates a snapshot of the entries maintained by the algorithm at the stage where $M[2, 7]$ is computed. At this stage, the algorithm examines the sum of $M[2, 6]$ and $M[7, 7]$ to compute $L^p(2, 7)$ (note that $Q^{oct}_{2,7} = \{7\}$), and $M[3, 6] + 1$ to compute $L^c(2, 7)$ (s_7 and s_2 are complementary bases). Since $L^c(2, 7) = 2 > 1 = L^p(2, 7)$, the sub-instance $S_{2,7}$ is marked as an OCT. The right figure shows the computation of the next entry, $M[2, 8]$. Here, the evaluation of $L^p(2, 8)$ examines the sum of $M[2, 6]$ and $M[7, 8]$, and the sum of $M[2, 7]$ and $M[8, 8]$ (since $Q^{oct}_{2,8} = \{7, 8\}$). $L^c(2, 8) = -\infty$, since s_8 and s_2 are not complementary.

3.3. Folding reconstruction

The previous section showed how to reduce the space requirement of the SSF optimization problem. In this section, we explain how to reduce the space requirement for the search problem as well. The standard technique for reporting an optimal folding applies a trace-back procedure over the full folding score table M, in $O(n^2)$ time (or in $O(n)$ time, if additional $O(n^2)$ pointers are maintained throughout the score computation phase) [13]. In this section we show how to reconstruct an optimal folding given the sparse representation of M, without exceeding the time and space complexities of our folding algorithm. Note that this is a challenging task, as the classical trace-back algorithm requires the availability of the full table M, while our algorithm stores only partial information.

Assume that the full table M is given, with annotated OCT sub-instances. The basic recursive folding reconstruction algorithm [13] could be modified as follows to utilize the OCT sub-instances:

1. For $j \leqslant i$, the only (optimal) folding of $S_{i,j}$ is the empty folding, and the algorithm halts without reporting any base-pair.
2. For $j > i$, if $S_{i,j}$ is an OCT, the algorithm reports the pair (i, j) and is called recursively on the sub-instance $S_{i+1,j-1}$.
3. Otherwise, $S_{i,j}$ is partitionable, and therefore the algorithm finds an index $q \in Q^{oct}_{i,j}$ for which $M[i, j] = M[i, q-1] + M[q, j]$ and then continues by computing an optimal folding of $S_{i,q-1}$ and of $S_{q,j}$. An optimal folding of $S_{i,q-1}$ is obtained by calling the algorithm recursively with the sub-instance $S_{i,q-1}$. As for computing an optimal folding of $S_{q,j}$, note that $S_{q,j}$ is an OCT, and consider the two cases, where either $q = j$ or $q < j$. If $q = j$, then there is no need for another recursive call. Otherwise $q < j$, and an optimal folding of $S_{q,j}$ is obtained by first reporting the base-pair (q, j) and then calling the algorithm recursively with the sub-instance $S_{q+1,j-1}$.

When calling the above algorithm to compute the folding traceback of $S_{i,j}$, recursive calls with three different sub-instances could be initiated at the top level: $S_{i-1,j-1}$, $S_{q+1,j-1}$ and $S_{i,q-1}$, thus index j is eliminated from further consideration as an end index. Therefore, each recursive call is performed with a different end index j, and altogether there are at most n recursive calls in the whole computation. For a recursive call in which the end index is j, at most $O(|Q^{oct}_{1,j}|)$ operations are preformed in order to find an index $q \in Q^{oct}_{i,j}$ for which $M[i, j] = M[i, q-1] + M[q, j]$. Since $\sum_{1 \leqslant j \leqslant n} |Q^{oct}_{1,j}| = Z$, the total running time is $O(Z)$.

We next address the challenge of reconstructing an optimal folding from the sparse table M computed in Section 3.2. The algorithm described above cannot be applied directly in this case, due to the fact that when the algorithm needs to find $q \in Q^{oct}_{i,j}$ for which $M[i,j] = M[i,q-1] + M[q,j]$, the values $M[i,j]$ and $M[i,q-1]$ may have been discarded from memory (while $M[q,j]$ is maintained in memory since $S_{q,j}$ is an OCT). In order to overcome this difficulty we adopt a similar approach as of the algorithm of Hirschberg [19], namely performing on-demand value re-computations of discarded entries. Thus, it remains to show how to recover such deleted entries.

Lemma 4. *Given the sparse table M that contains folding scores for OCT sub-instances, there is an algorithm that recovers the set of entries $M[i,i], M[i,i+1], \dots, M[i,j]$, for any given pair of indices i and j, in $O(Z)$ time.*

Proof. The entries of the form $M[i,j']$ which have been discarded from memory correspond to partitionable sub-instances, where $L(i,j') = L^p(i,j')$, and thus may be recomputed based solely on Eq. (3.1). Observe that this computation examines only entries of the form $M[i,q-1]$ for $q \leqslant j'$, and $M[q,j']$ for OCT sub-instances $S_{q,j'}$. Re-computing the entries of the ith row from left to right guaranties that upon computing $M[i,j']$, all necessary values for the computation of $L^p(i,j')$ are already stored in M. For each $i < j' \leqslant j$, there are $O(|Q^{oct}_{1,j'}|)$ operations performed along this computation, due to the consideration of split-points in the set $Q^{oct}_{i,j'}$. As before, summing this expression over all $i < j' \leqslant j$ accumulates to $O(Z)$ time complexity. $\square$

We next show that, throughout the full run of the algorithm, the process of restoring row entries is applied once for every reported base-pair. Consider the case where the trace-back algorithm is applied on $S_{i,j}$ and assume that the set of entries $M[i,i+1], M[i,i+2], \dots, M[i,j]$ was already restored. Note that a recursive call with a sub-instance of the form $S_{i,q-1}$ does not require the restoration of the entries $M[i,i+1], M[i,i+2], \dots, M[i,q-1]$, as (by the assumption) they have already been restored and are maintained in M. The other two possible recursive calls with sub-instances of the form $S_{i+1,j-1}$ or $S_{q+1,j-1}$, do require re-computation of entries in M (in rows $i+1$ or $q+1$, correspondingly). However, observe that each call of the latter kind is preceded by a detection of a base-pair. Since throughout the full run of the algorithm at most $n/2$ base pairs are detected, we get that the row entry recovery only needs to be executed $O(n)$ times (in addition to the recovery of $M[1,1], M[1,2], \dots, M[1,n]$ during initialization). Thus, according to Lemma 4, the entry recovery procedure contributes an additional $O(nZ)$ factor to the total time complexity of the trace-back algorithm, matching the time complexity of the computation of M.

Furthermore, note that upon performing such a re-computation of an entry set, there is no need to further maintain the values in $M[i,i+1], M[i,i+2], \dots, M[i,j]$ in the case where $S_{i,j}$ is co-terminus, nor to keep the values in $M[i,q], M[i,q+1], \dots, M[i,j]$ in the case where $S_{i,j}$ is partitionable. This allows to discard these values from memory before the re-computation of the entry set for the corresponding sub-instance, guaranteeing that at each stage, at most n recovered entries are maintained in the sparse table M, in addition to the already existing OCT corresponding entries. Therefore, the space complexity of the trace-back algorithm remains $\Theta(Z+n) = \Theta(Z)$.

Algorithm 2 below implements the space efficient trace-back scheme. Its time and space complexities are stated in Lemma 5.

```
Algorithm 2: Folding-Traceback (M)
  input  : A sparse table $M$ that contains solutions for all OCT sub-instances of an instance $S$
  output: An optimal folding of $S$
1 call Restore-entries ($M, 1, n$);
2 call Rec-Folding-Traceback ($M, 1, n$);
```

```
Procedure Restore-entries($M, i, j$)
  input  : A sparse table $M$ that contains solutions for all OCT sub-instances of an instance $S$, and two internal indices $i$ and $j$
  output: The table $M$ after restoring values in the entries $M[i,i], M[i,i+1], \dots, M[i,j]$
1 for $j' = i+1$ to $j$ do
2     if $M[i,j']$ is discarded then
3         set $M[i,j'] \leftarrow \max_{q \in Q^{oct}_{i,j'}} \{M[i,q-1] + M[q,j']\}$;
```

Lemma 5. *Given the sparse table M that contains folding scores for all OCT sub-instance of an instance S, Algorithm 2 computes an optimal folding of S in $O(nZ)$ time and $\Theta(Z)$ space.*

4. Reducing the time complexity

In this section we show how to further restrict the set of split-points which are examined in the computation of L^p, thus reducing the bottleneck computation of the algorithm. We introduce an additional sparsity parameter P, satisfying

Procedure Rec-Folding-Traceback(M, i, j)

```
input : A sparse table $M$ that contains folding scores of all OCT sub-instances of an instance $S$, as well as folding scores for the sub-instances
        $S_{i,i}, S_{i,i+1}, \ldots, S_{i,j}$ for the two given indices $i$ and $j$
output: An optimal folding of $S_{i,j}$
1  if $i < j$ then
2      if $S_{i,j}$ is an OCT then
3          output the pair $(i, j)$;
4          discard from memory the values in all non-OCT entries $M[i,i], M[i,i+1], \ldots, M[i,j]$;
5          call Restore-entries $(M, i+1, j-1)$;
6          call Rec-Folding-Traceback $(M, i+1, j-1)$;
7      else
8          find $q \in Q^{oct}_{i,j}$ s.t. $M[i,j] = M[i,q-1] + M[q,j]$;
9          discard from memory the values in all non-OCT entries $M[i,q], M[i,q+1], \ldots, M[i,j]$;
10         if $q < j$ then
11             output the pair $(q, j)$;
12             call Restore-entries $(M, q+1, j-1)$;
13             call Rec-Folding-Traceback $(M, q+1, j-1)$;
14         call Rec-Folding-Traceback $(M, i, q-1)$;
```

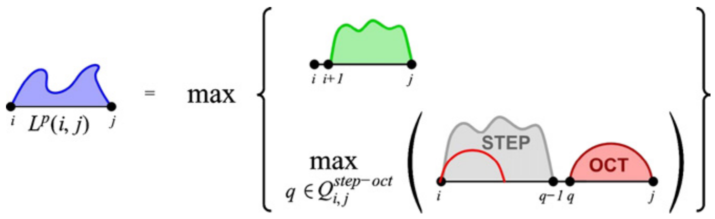

Fig. 6. A schematic illustration of Eq. (4.1). The split-point $i+1$ is examined explicitly, as well as all split-points that induce a STEP-prefix-OCT-suffix pattern.

$P < n \leqslant Z \leqslant n(P+1)$, and show in Section 4.1 how to reduce the running time of the algorithm from $O(nZ)$ to $O(n^2 + PZ)$. In Section 4.2 we further reduce the time complexity to $O(LZ)$ for the base-pairing maximization scoring scheme. Both algorithms have the same space complexity as of Algorithm 1, which is $O(Z)$.

4.1. An $O(n^2 + PZ)$ algorithm

Similarly to the previously presented technique, we next show another dominance relation which can be utilized to further constrain the set of split-points examined in the computation of $L^p(i, j)$.

Definition 4 *(STEP).* Call a sub-instance $S_{i,j}$, with $j > i$, a STEP, if $L(i, j) > L(i, i) + L(i+1, j)$ (i.e. none of the optimal foldings of $S_{i,j}$ admits the split-point $i+1$).

Observe that if $S_{i,j}$ is a STEP, it implies that i is paired in every optimal folding of $S_{i,j}$. In the following lemma we further restrict the split-points which need to be examined in a recursive computation of $L^p(i, j)$.

Lemma 6. *For any sub-instance $S_{i,j}$ such that $j > i$, there is an optimal split-point q with respect to $S_{i,j}$ such that either $q = i+1$, or $S_{i,q-1}$ is a STEP and $S_{q,j}$ is an OCT.*

Proof. According to Lemma 1, there is an optimal split-point $q \in Q^{oct}_{i,j}$ (where the suffix $S_{q,j}$ is an OCT) such that $L^p(i, j) = L(i, q-1) + L(q, j)$. If the prefix $S_{i,q-1}$ is a STEP, the lemma holds. Otherwise, $L(i, q-1) = L(i, i) + L(i+1, q-1)$, and from the triangle inequality property, $L^p(i, j) = L(i, i) + L(i+1, q-1) + L(q, j) \leqslant L(i, i) + L(i+1, j) \leqslant L^p(i, j)$. Therefore, $L^p(i, j) = L(i, i) + L(i+1, j)$, and $i+1$ is an optimal split-point with respect to $S_{i,j}$. $\square$

Define the following subset of split-points with respect to $S_{i,j}$:

$$Q^{\text{step-oct}}_{i,j} = \{q \in Q_{i,j}\colon\ S_{i,q-1} \text{ is a STEP and } S_{q,j} \text{ is an OCT}\}.$$

The following equation restates Eq. (3.1), based on Lemma 6. Fig. 6 illustrates the modified computation of L^p.

$$L^p(i, j) = \max_{q \in \{i+1\} \cup Q^{\text{step-oct}}_{i,j}} \bigl\{L(i, q-1) + L(q, j)\bigr\}. \tag{4.1}$$

Let $P = P(S)$ denote the maximum number of STEP sub-instances of S ending at index j, taking over all $1 \leqslant j \leqslant n$, i.e.

$$P(S) = \max_{1 \leqslant j \leqslant n} \bigl|\{i\colon\ S_{i,j} \text{ is a STEP}\}\bigr|.$$

Note that by definition, $P < n$. Also note that any OCT sub-instance of length greater than 1 is a STEP (though the opposite is not necessarily true), thus there are at most nP OCT sub-instances of length greater than 1, and n OCT sub-instances of length 1, and in total $Z \leqslant n(P+1)$. For the base-pairing maximization scoring scheme, $S_{i,j}$ is a STEP if $L(i,j) > L(i,i) + L(i+1,j) = L(i+1,j)$. Note that $L(i,j)$ is either $L(i+1,j)$ or $L(i+1,j)+1$ (an optimal folding for $S_{i,j}$ may contain at most 1 additional base-pair with respect to an optimal folding of $S_{i+1,j}$) thus $S_{i,j}$ is a STEP if $L(i,j) = L(i+1,j)+1$. Since $L(i,j)$ is in the range $0, 1, \ldots, L$, there are at most L STEP sub-instances ending at index j (and exactly L such sub-instances for $j = n$), and $P = L \leqslant n/2$.

We next show a bottom-up algorithm that computes L according to Eqs. (2.1), (2.2), and (4.1). The presented algorithm is similar to Algorithm 1, where a forward dynamic programming technique is applied in order to efficiently compute $L^p(i,j)$.

The new algorithm also scans and computes the entries of M in decreasing row index and increasing column index. It maintains the following invariant: *upon reaching entry $M[i,j]$, the entry contains the value $L^p(i,j)$*. Before computing row i in M, the entries $M[i,i-1]$ and $M[i,i]$ are initialized explicitly with the solutions of the corresponding trivial sub-instances. All entries $M[i,j]$ for $i < j \leqslant n$ are initialized with the corresponding values $M[i,i] + M[i+1,j]$. This initialization is equivalent to examining the split-point $q = i+1$ in the computation of $L^p(i,j)$ according to Eq. (4.1) for all $j > i$. Note that at this stage the invariant is sustained for the first entry in the row which is traversed by the algorithm — $M[i,i+1]$, since $Q^{\text{step-oct}}_{i,i+1} = \emptyset$.

Based on the invariant, upon reaching $M[i,j]$, the entry contains the value $L^p(i,j)$. The value $L(i,j)$ can then be computed by resolving the maximum between the current entry value, and the value of $L^c(i,j)$ which is obtained from Eq. (2.2). If $L^c(i,j) > L^p(i,j)$, $S_{i,j}$ is classified as an OCT. Then, if $M[i,j] > M[i,i] + M[i+1,j]$, $S_{i,j}$ is classified as a STEP, and the split-point $q = j+1$ is considered and forward-reflected to the computation of $L^p(i,j')$, for all $j' > j$ such that $S_{j+1,j'}$ is an OCT. This forward computation updates the value of $M[i,j']$ to be the maximum among its current value, and that of $M[i,j] + M[j+1,j']$. Thus, the maximum expression of Eq. (4.1) is accumulated, and the maintenance of the invariant is guaranteed.

Algorithm 3 below implements the described forward dynamic programming approach, combined with the space-efficient approach described in Section 3.2. An illustration of its run is given in Fig. 7. The speedup obtained by this algorithm is due to the fact that a split-point q is examined by the algorithm only if $q = i+1$, or the prefix $S_{i,q-1}$ is a STEP and the suffix $S_{q,j}$ is an OCT. Note that, for each one of the Z OCT sub-instances $S_{q,j}$, there are at most P sub-instances $S_{i,q-1}$ which may be corresponding STEP prefixes. Thus, the total run-time contribution due to the examination of split-points inducing a STEP-prefix-OCT-suffix partition is $O(PZ)$. Since additional $O(1)$ operations are performed for each one of the $O(n^2)$ sub-instances (the examination of the split-point $i+1$ and the computation of L^c), the total running time is $O(n^2 + PZ)$. The space complexity remains $O(Z)$, as the space complexity of Algorithm 1.

Lemma 7. *Algorithm 3 computes $L(1,n)$ for a given instance S of length n, in $O(n^2 + PZ)$ time and $\Theta(Z)$ space.*

Algorithm 3: Forward RNA folding

```
input  : An RNA string $S = s_1 s_2 \cdots s_n$
output: $L(1,n)$
1  for $i \leftarrow n$ down to 1 do
2      set explicitly the values of $M[i,i-1]$ and $M[i,i]$ to the solutions for the corresponding trivial sub-instances;
3      mark $S_{i,i}$ as an OCT;
4      set $M[i,j] \leftarrow M[i,i] + M[i+1,j]$ for all $i < j \leqslant n$;
5      for $j \leftarrow i+1$ to $n$ do
6          compute $L^c(i,j) = f(M[i+1,j-1], s_i, s_j)$;
7          if $M[i,j] < L^c(i,j)$ then
8              set $M[i,j] \leftarrow L^c(i,j)$;
9              mark $S_{i,j}$ as an OCT;
10         if $S_{i,j}$ is a STEP then
11             for all $j'$ s.t. $S_{j+1,j'}$ is an OCT do
12                 set $M[i,j'] \leftarrow \max\{M[i,j'], M[i,j] + M[j+1,j']\}$;
13     discard from memory the values in all entries in row $i+1$ of $M$ that do not correspond to OCT sub-instances;
14 return $M[1,n]$.
```

4.2. An $O(LZ)$ algorithm for the base-pairing maximization variant

For the base-pairing maximization problem $P = L$, and the running time of the algorithm which was presented in the previous section is $O(n^2 + LZ)$. In this section we further reduce the running time for this variant to $O(LZ)$. While in the worst case $LZ = \Theta(n^3)$, in the sparse case LZ can get to be as low as $\Theta(n)$ (e.g. when $L = O(1)$), motivating

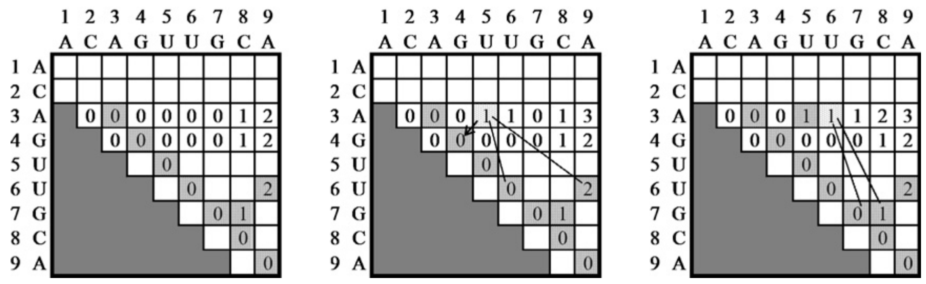

Fig. 7. An exemplification of Algorithm 3. The left figure shows the initialization of row 3 in the table M, copying the values from row 4. The figure in the middle demonstrates the computation of the entry $M[3,5]$. Upon reaching this entry, the entry value is 0, which corresponds to $L^p(3,5)$. Since s_5 and s_3 are complementary bases, $L^c(3,5) = M[4,4]+1 = 1$ is evaluated, the entry value is updated to be 1, and $S_{3,5}$ is marked as an OCT. Since $M[3,5] > M[4,5]$, $S_{3,5}$ is classified as a STEP, and the splitting at $q = 6$ is considered for all sub-instances $S_{3,j}$ such that $S_{6,j}$ is an OCT. This forward computation updates the values in $M[3,6]$ and $M[3,9]$. The right figure shows the computation of the next entry, $M[3,6]$. Here there is no need to evaluate $L^c(3,6)$ (since s_6 and s_3 are not complementary), and $S_{3,6}$ is also classified as a STEP (since $M[3,6] > M[4,6]$). Therefore, splitting at $q = 7$ is considered for all sub-instances $S_{3,j}$ such that $S_{7,j}$ is an OCT, updating the values in $M[3,7]$ and $M[3,8]$.

this improvement. The time reduction is achieved by applying a *step encoding* [20] to the dynamic programming table, representing each row in the table by its $O(L)$ steps (see Fig. 9). Hence, in what follows we give corresponding step-encoding formulation of the problem.

While in the standard formulation of the SSF problem sub-instances were defined with respect to a pair of indices i and j in the input string S, in the step-encoding formulation sub-instances are defined by an index i in S and a score x. The next definition gives the step-encoding equivalents of the entities $L(i,j)$, $L^p(i,j)$, and $L^c(i,j)$.

Definition 5. For $1 \leqslant i \leqslant n$, $0 \leqslant x$, and $\alpha \in \{\varepsilon, p, c\}$ (where ε denotes the empty word), define $\beta^\alpha(i,x)$ to be the minimum index j such that $L^\alpha(i,j) \geqslant x$, or ∞ if there is no such j.

From the definition above, we get that

$$\beta(i,x) = \min\{\beta^c(i,x), \beta^p(i,x)\}. \tag{4.2}$$

Note the relation between the step-encoding formulation and the standard formulation, where $L(i,j)$ is the maximum x such that $\beta(i,x) \leqslant j$. Also, observe that for $j > i$, $S_{i,j}$ is an OCT if and only if there is some $x \geqslant 1$ such that $j = \beta^c(i,x) > \beta^p(i,x)$. The set $Y_{i,x}$ is the step-encoding equivalent of $Q_{i,j}^{\text{step-oct}}$:

$$Y_{i,x} = \{j\colon\ \exists q \text{ s.t. } S_{i,q-1} \text{ is a STEP, } S_{q,j} \text{ is an OCT, and } L(i,q-1) + L(q,j) = x\}.$$

The following auxiliary function will be used in the computation of $\beta^c(i,x)$.

Definition 6. For $\sigma \in \{A,C,G,U\}$ and $1 \leqslant r < n$, define $next(r,\sigma)$ to be the minimum index $r' > r$ such that $s_{r'}$ is *complementary* to σ, or ∞ if there is no such index r'.

A straightforward $O(n)$ time and space preprocessing allows to obtain values of queries of the form $next(r,\sigma)$ in $O(1)$ time. The description of this preprocessing is left out of this paper, and its implementation can be found with our online code.

For $x = 0$, $\beta(i,x) = i-1$ by definition. For $x \geqslant 1$, we now convert Eqs. (2.2) and (4.1) to their equivalent forms in the step encoding formulation. Fig. 8 illustrates this recurrence.

$$\beta^c(i,x) = next\big(\beta(i+1,x-1), s_i\big), \tag{4.3}$$

$$\beta^p(i,x) = \min\Big\{\beta(i+1,x),\ \min_{j\in Y_{i,x}}\{j\}\Big\}. \tag{4.4}$$

Proof. [Eq. (4.3)] Let $j = \beta^c(i,x)$, and $j' = \beta(i+1,x-1)$ (i.e. j is the minimum index such that there is a co-terminus folding of $S_{i,j}$ of size x, and j' is the minimum index such that there is (any) folding of $S_{i+1,j'}$ of size $x-1$). Then, we need to show that $j = next(j', s_i)$.

First, we show that $j \geqslant next(j', s_i)$. Let F be a co-terminus folding of $S_{i,j}$ of size x, and consider the folding F' which is obtained by removing the base-pair (i,j) from F. Then, F' is a folding of $S_{i+1,j-1}$ of size $x-1$, and in particular $j' \leqslant j-1$. Since s_j and s_i are complementary, it follows that $j = next(j-1, s_i) \geqslant next(j', s_i)$.

Next, we show that $j \leqslant next(j', s_i)$. Any folding which is obtained by adding the base-pair $(i, next(j', s_i))$ to an optimal folding of $S_{i+1,j'}$ is a co-terminus folding of size x for the sub-instance $S_{i,next(j',s_i)}$, and since j is the minimum index such that there is a co-terminus folding of $S_{i,j}$ of size x, we have that $j \leqslant next(j', s_i)$. Thus, $j = next(j', s_i)$. □

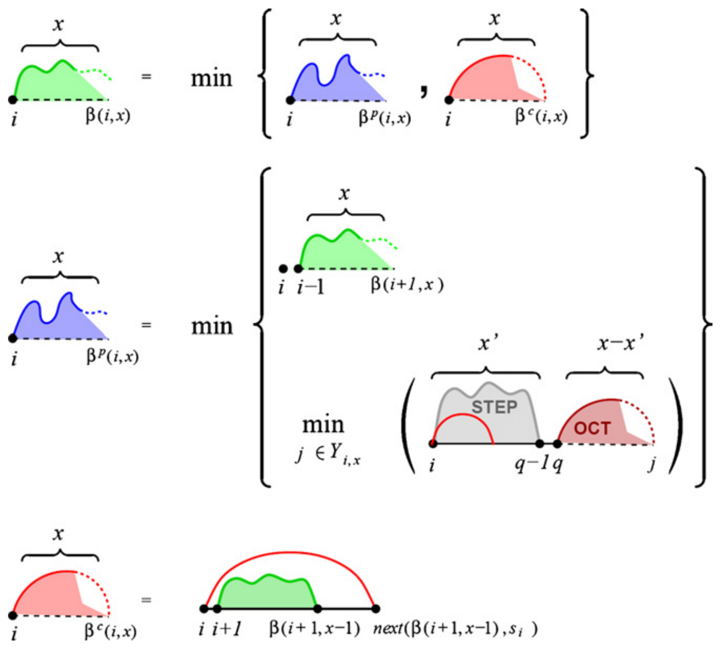

Fig. 8. A schematic illustration of the recursion of Eqs. (4.2), (4.3), and (4.4).

Proof. [Eq. (4.4)] Let $j^* = \beta^p(i,x)$, and $j' = \beta(i+1,x)$ (i.e. j^* is the minimum index such that there is a partitionable folding of S_{i,j^*} of size x, and j' is the minimum index such that there is (any) folding of $S_{i+1,j'}$ of size x). Then, we need to show that $j^* = \min\{j', \min_{j\in Y_{i,x}}\{j\}\}$.

First, we show that $j^* \geqslant \min\{j', \min_{j\in Y_{i,x}}\{j\}\}$. From Lemma 6, there is an index q such that $L(i,q-1) + L(q,j^*) = L(i,j^*) = x$, and either $q = i+1$ or $S_{i,q-1}$ is a STEP and S_{q,j^*} is an OCT. If $q = i+1$, then since $L(i,q-1) = L(i,i) = 0$, we have that $L(q,j^*) = L(i+1,j^*) = x$, and in particular $j^* \geqslant j'$. Otherwise, if $S_{i,q-1}$ is a STEP and S_{q,j^*} is an OCT, then $j^* \in Y_{i,x}$ and in particular $j^* \geqslant \min_{j\in Y_{i,x}}\{j\}$.

Next, we show that $j^* \leqslant \min\{j', \min_{j\in Y_{i,x}}\{j\}\}$. By the definition of $Y_{i,x}$, it is clear that for every $j \in Y_{i,x}$ there is a partitionable folding of $S_{i,j}$ of size x, and therefore $j^* \leqslant \min_{j\in Y_{i,x}}\{j\}$. Also, $j^* \leqslant j'$ since any folding of $S_{i+1,j'}$ of size x is also a partitionable folding of $S_{i,j'}$ of size x, admitting the split-point $i+1$.

Thus, $j = \min\{j', \min_{j\in Y_{i,x}} j\}$. $\square$

We next show an explicit bottom-up dynamic programming algorithm that computes the recurrence in Eqs. (4.2), (4.3), and (4.4). This algorithm adopts a forward dynamic programming approach, similarly to that of Algorithm 3, where the number of sub-instances, as well as the dimensions of the data structure that stores solutions for these sub-instances, is $O(Ln)$ instead of $O(n^2)$.

The algorithm fills a table B of size $O(nL)$, whose entries $B[i,x]$ store solutions $\beta(i,x)$ for sub-instances (S,i,x) (see Fig. 9). It maintains the following invariant: *upon reaching entry* $B[i,x]$, *if* $\beta^p(i,x) \leqslant \beta^c(i,x)$, *then the entry contains the value* $\beta^p(i,x)$.

Before computing row i in B, the entry $B[i,0]$ is initialized with $i-1$, and all entries $B[i,x]$ for $x \geqslant 1$ such that $B[i+1,x] < -\infty$ are initialized with the corresponding values $B[i+1,x]$. This initialization is equivalent to examining the value $\beta(i+1,x)$ in the computation of $\beta^p(i,x)$ according to Eq. (4.4).

We show that the invariant is sustained at this stage for the entry $B[i,1]$. If $\beta^p(i,1) > \beta^c(i,1)$ then the invariant is sustained by definition. If $\beta^p(i,1) \leqslant \beta^c(i,1)$, let $j = \beta^p(i,1)$ and consider any split-point q such that $L(i,q-1) + L(q,j) = 1$. Note that $L(i,q-1) = 0$, since otherwise it contradicts the definition of j. Thus $S_{i,q-1}$ is not a STEP, and in particular $Y_{i,1} = \emptyset$. In this case, $\beta^p(i,1) = \beta(i+1,1) = B[i+1,1]$, which is indeed the value of $B[i,1]$ according to the initialization.

Based on the invariant, upon reaching $B[i,x]$, the entry contains the value $\beta^p(i,x)$ if $\beta^p(i,x) \leqslant \beta^c(i,x)$. The value of $\beta^c(i,x)$ is then computed according to Eq. (4.3). If $\beta^c(i,x) < B[i,x]$, it follows that $\beta^c(i,x) < \beta^p(i,x)$, and therefore the entry $B[i,x]$ is set to $\beta^c(i,x)$, and $S_{i,B[i,x]}$ is marked as an OCT. Else, $\beta^c(i,x) \geqslant B[i,x]$, and according to the invariant the entry contains at this stage the value $\beta(i,x) = \beta^p(i,x)$.

Then, for all q such that $B[i,x] \leqslant q-1 < B[i+1,x]$, the sub-instance $S_{i,q-1}$ is a STEP with $L(i,q-1) = x$, and therefore for all j such that $S_{q,j}$ is an OCT the algorithm updates the value of $B[i,x+L(q,j)]$ to be the minimum between its current value and j, thus accumulating the minimum according to Eq. (4.4), and guaranteeing the maintenance of the invariant.

As was shown for Algorithm 3, the number of operations due to the examinations of split-points in the computation of Eq. (4.4) throughout the whole run of the algorithm is $O(LZ)$. Other than that, there are $O(1)$ operations for each of the $O(nL)$ sub-instance (S,i,x), and the total running time is $O(nL+LZ) = O(LZ)$. The space complexity remains $\Theta(Z)$, as of Algorithm 1.

Algorithm 4: Step encoded RNA folding

```
input  : An RNA string $S = s_1 s_2 \cdots s_n$
output: $L(1, n)$
1  set $B[n, 0] \leftarrow n - 1$;
2  for $i \leftarrow n - 1$ down to 1 do
3      set $B[i, 0] \leftarrow i - 1$,  $B[i, L(i + 1, n) + 1] \leftarrow \infty$;
4      set $B[i, x] \leftarrow B[i + 1, x]$ for all $x$ s.t. $B[i + 1, x] < \infty$;
5      for $x \leftarrow 1$ to $L(i + 1, n) + 1$ do
6          if $B[i, x] > next(B[i + 1, x - 1], s_i)$ then
7              set $B[i, x] \leftarrow next(B[i + 1, x - 1], s_i)$;
8              mark $S_{i,B[i,x]}$ as an OCT;
9          if $x \leqslant L(i + 1, n)$ then
10             for all $q$ s.t. $B[i, x] \leqslant q - 1 < B[i + 1, x]$ do
11                 for all $j$ s.t. $S_{q,j}$ is an OCT do
12                     set $B[i, x + L(q, j)] \leftarrow \min\{B[i, x + L(q, j)], j\}$;
13     discard from memory the values in all entries in row $i + 1$ of $B$ that do not correspond to OCT sub-instances;
14 return $L(1, n)$ - the maximum $x$ such that $B[1, x] < \infty$;
```

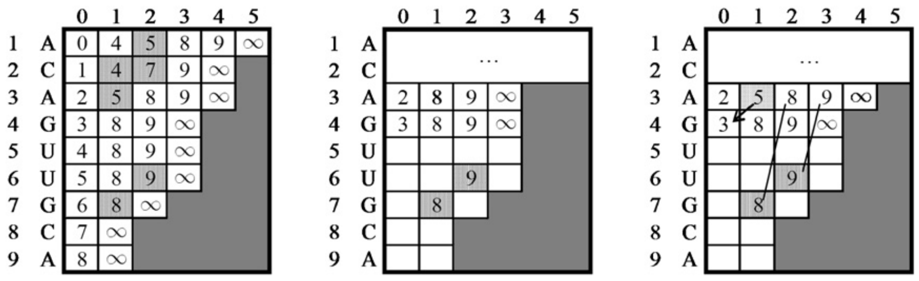

Fig. 9. An exemplification of Algorithm 4. This figure shows the table B with respect to the RNA string $S = \text{ACAGUUGCA}$, which is the *step encoding* of the table M in Fig. 5. The left plot shows the complete table, were grayed entries $B[i, x]$ correspond to OCT sub-instances $S_{i,B[i,x]}$. The plot at the middle demonstrates a snapshot of the entries maintained by the algorithm at the stage where row 3 is about to be computed, after the initialization. The right plot shows the computation of the entry $B[3, 1]$. Since $\beta^c(i, x) = next(B[4, 0], s_3) = next(3, A) = 5$, the initial entry value 8 is replaced with the value 5, and the sub-instance $S_{3,5}$ is marked as an OCT. Then, all split-points q such the $B[3, 1] = 5 \leqslant q - 1 < 8 = B[4, 1]$ are examined with respect to OCT suffixes $S_{q,j}$. This computation updates the values in the entries $B[3, 2]$ and $B[3, 3]$.

Algorithm 4 gives the pseudo-code of the algorithm described above, and Fig. 9 illustrates its run. The time and space complexities of the algorithm are summarized in the following lemma.

Lemma 8. *Algorithm 4 computes $L(1, n)$ for a given instance S of length n, in $O(LZ)$ time and $\Theta(Z)$ space.*

5. Simultaneous alignment and folding

The *RNA Simultaneous Alignment and Folding* problem (SAF) was defined by David Sankoff [28]. Similarly to the classical sequence alignment problem, the goal of the SAF problem is to find an alignment of several RNA strings, and in addition to find a common folding for the aligned segments of the strings. The score of a given alignment with folding takes into account both standard alignment elements such as character matching, substitutions and indels, as well as the folding score. For simplicity, our formulation assumes that the instance is composed of two strings, and uses a simple scoring scheme. It is possible to extend this formulation to handle an arbitrary number of strings in the instance, as well as to incorporate more complex scoring schemes (e.g. energy minimization), however this is beyond the scope of this paper.

Similarly to single RNA strand folding algorithms, the basic dynamic programming algorithm for the SAF problem [28] computes the scores of all sub-instances of its input instance, and then combines these values to resolve the score of the full input instance. Ziv-Ukelson et al. [38] have extended the results from [34] to the SAF problem, taking advantage of the triangle inequality property to constrain the split-points that need to be examined to only those for which the suffix of the partitioned instance is an OCT. Here, we further reduce the number of examined split-points by examining only split-points that partition the instance according to the STEP-prefix-OCT-suffix pattern (in addition to a constant number of base-case split-points), and apply a sparsifying technique similar to the one described in Section 3.2 in order to reduce the space complexity of the algorithm.

The rest of this section proceeds as follows. In Section 5.1 the problem is formally defined, Section 5.2 shows the recursive score computation by adjusting the techniques described above in the context of the SAF problem, and Section 5.3 shows an explicit score computation algorithm. Though the formulation of the SAF problem is a little more complex than that of the SSF problem, all computational techniques and results here are natural extensions of those in the previous sections, thus detailed proofs are omitted.

5.1. SAF Formal problem definition

An input instance of the SAF problem is a pair of RNA strings $\tilde{S} = (S, S')$. We use n and m to denote the lengths of S and S', respectively. Call $\tilde{S}$ an *empty instance* if $n = m = 0$. Let $\tilde{S}_{i,j;i',j'}$ denote the sub-instance $\tilde{S}_{i,j;i',j'} = (S_{i,j}, S'_{i',j'})$ of $\tilde{S}$. For two sub-instances $\tilde{S}_{i,j;i',j'}$ and $\tilde{S}_{k,l;k',l'}$, say that $\tilde{S}_{k,l;k',l'}$ is a sub instance of $\tilde{S}_{i,j;i',j'}$ if $i \leqslant k \leqslant l \leqslant j$ and $i' \leqslant k' \leqslant l' \leqslant j'$, where it is a *strict* sub-instance if $\tilde{S}_{k,l;k',l'} \neq \tilde{S}_{i,j;i',j'}$.

Definition 7. An *alignment* of the sub-instance $\tilde{S}_{i,j;i',j'}$ is a pair $\tilde{A} = (A, A')$, where $A = a_1a_2\cdots a_r$ and $A' = a'_1a'_2\cdots a'_r$ are strings of the same length r over the alphabet $\{A, C, G, U, -\}$, obtained by inserting *gap* characters "–" into $S_{i,j}$ and $S'_{i',j'}$, respectively. Denote by $|\tilde{A}|$ the *length* r of $\tilde{A}$.

Let $\tilde{A}_k = (a_k; a'_k)$ denote the kth *column* of an alignment $\tilde{A}$. We assume that no column $\tilde{A}_k$ is composed of gap characters only. A *folding* of an alignment is defined similarly to a folding of a single string, except for the fact that now each pair (k, l) in a folding F represents a pair of indices of columns in the alignment. Call a pair $(\tilde{A}, F)$, where $\tilde{A}$ is an alignment of a sub-instance $\tilde{S}_{i,j;i',j'}$ and F is a folding of $\tilde{A}$, an *alignment with folding* of $\tilde{S}_{i,j;i',j'}$ (see Fig. 10).

An *SAF scoring scheme* $\textsc{score}$ is a function which evaluates the qualities of alignments with foldings, where $\textsc{score}(\tilde{A}, F) > \textsc{score}(\tilde{A}', F')$ implies that the alignment with folding $(\tilde{A}, F)$ is of a better quality than $(\tilde{A}', F')$. Typically, a scoring scheme takes into account standard alignment-related expressions such as indels, base-matchings, and base-substitutions costs, as well as folding-related expressions such as the number of base-pairs or folding energy. In addition, a scoring scheme may also promote compensatory mutations by highly scoring pairs of the form (k, l) in the folding, such that the paired columns $\tilde{A}_k$ and $\tilde{A}_l$ exhibit nucleotide complementarity between both pairs (a_k, a_l) and (a'_k, a'_l), while $a_k \neq a'_k$ and $a_l \neq a'_l$. The *RNA Simultaneous Alignment and Folding* problem (SAF) is defined as follows:

Problem 2 *(SAF).* Let $\textsc{score}$ be some SAF scoring scheme, and $\tilde{S} = (S, S')$ an SAF instance with $|S| = n$ and $|S'| = m$. Let $\tilde{D}_{i,j;i',j'}$ denote the set of all possible alignments with foldings of a sub-instance $\tilde{S}_{i,j;i',j'}$.

- The SAF-*optimization* problem is to calculate $\tilde{L}(1, n; 1, m)$, where
$$\tilde{L}(i, j; i', j') = \max_{(\tilde{A}, F) \in \tilde{D}_{i,j;i',j'}} \{\textsc{score}(\tilde{A}, F)\}.$$
- The SAF-*search* problem is to find an alignment with folding $(\tilde{A}^*, F^*)$ for $\tilde{S}$ that satisfies $\textsc{score}(\tilde{A}^*, F^*) = \tilde{L}(1, n; 1, m)$ (call such an alignment with folding *optimal* under the scoring scheme $\textsc{score}$).

In this work, we assume that the scoring scheme is of the following form:

$$\textsc{score}(\tilde{A}, F) = \sum_{1 \leqslant k \leqslant |\tilde{A}|} \delta(\tilde{A}_k) + \sum_{(k,l) \in F} \tau(\tilde{A}_k, \tilde{A}_l).$$

Here, δ is a *column aligning cost function*, and τ is a *column-pair aligning cost function*. Typically, $\delta(\tilde{A}_k)$ reflects the alignment quality of the kth column in $\tilde{A}$, giving high scores for aligning nucleotides of the same type and penalizing alignment of nucleotides of different types or aligning a nucleotide against a gap. $\tau(\tilde{A}_k, \tilde{A}_l)$ reflects the benefit from forming a base-pair in both of the input strings S and S' between the bases corresponding to columns $\tilde{A}_k$ and $\tilde{A}_l$ of the alignment (if gaps or non-complementary bases are present in these columns, it may induce a score penalty). In addition, compensatory mutations in these columns may also increase the value of $\tau(\tilde{A}_k, \tilde{A}_l)$ (thus it may compensate for some penalties taken into account in the computation of $\delta(\tilde{A}_k)$ and $\delta(\tilde{A}_l)$).

The set of *split-points* $Q_{i,j;i',j'}$ with respect to a sub-instance $\tilde{S}_{i,j;i',j'}$ is the set of pairs

$$Q_{i,j;i'j'} = \{(q, q'): i \leqslant q \leqslant j + 1,\ i' \leqslant q' \leqslant j' + 1\} \backslash \{(i, i'), (j + 1, j' + 1)\}.$$

A split-point $(q, q') \in Q_{i,j;i',j'}$ implies a partition of $\tilde{S}_{i,j;i',j'}$ into two sub-instances: a prefix sub-instance $\tilde{S}_{i,q-1,i',q'-1}$ and a suffix sub-instance $\tilde{S}_{q,j,q',j'}$. Excluding the points (i, i') and $(j + 1, j' + 1)$ from $Q_{i,j;i',j'}$ guarantees that each one of the sub-instances $\tilde{S}_{i,q-1,i',q'-1}$ and $\tilde{S}_{q,j,q',j'}$ is a non-empty, strict sub-instance of $\tilde{S}_{i,j;i',j'}$. Say that an alignment with folding $(\tilde{A}, F)$ of $\tilde{S}_{i,j;i',j'}$ *admits* the split-point $(q, q') \in Q_{i,j;i',j'}$ if there is some $1 \leqslant p \leqslant |\tilde{A}|$ such that:

- The suffixes $a_pa_{p+1}\cdots a_{|\tilde{A}|}$ and $a'_pa'_{p+1}\cdots a'_{|\tilde{A}|}$, excluding the gaps, are identical to the strings $S_{q,j}$ and $S'_{q',j'}$, respectively.
- For every $(k, l) \in F$, either $l < p$ or $k \geqslant p$.

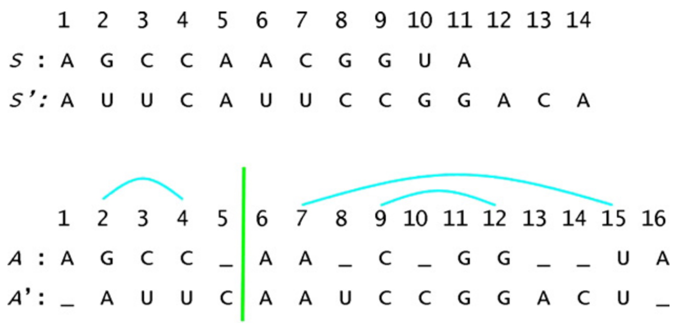
1 2 3 4 5 6 7 8 9 10 11 12 13 14
S : A G C C A A C G G U A
S': A U U C A U U C C G G A C A
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
A : A G C C _ A A _ C _ G G _ _ U A
A': _ A U U C A A U C C G G A C U _

Fig. 10. An SAF-instance, and a corresponding alignment with folding. The folding contains the column pairs $(2,4)$, $(7,15)$, and $(9,12)$. Column 6 in the alignment sustains the condition that for every (k,l) in the folding, either $l<6$ or $k \geqslant 6$, and shows that the alignment with folding admits the split-point $(5,5)$. Columns 2, 5, 7, and 16 correspond to the split-points $(1,0)$, $(5,4)$, $(6,6)$ and $(11,15)$ respectively, which are also admitted by the alignment with folding.

In other words, an alignment with folding admits a split-point (q,q') if (q,q') implies a partition of the alignment with folding into two independent alignments with folding, one for the prefix sub-instance $\tilde{S}_{i,q-1,i',q'-1}$ and one for the suffix sub-instance $\tilde{S}_{q,j,q',j'}$ (see Fig. 10).

5.2. *A recursive score computation*

Again, we distinguish between two kinds of alignments with folding of a sub instance $\tilde{S}_{i,j;i',j'}$. An alignment with folding $(\tilde{A},F)$ is called *partitionable* if it admits at least one split-point, otherwise $(\tilde{A},F)$ is called *co-terminus*. Observe that a co-terminus alignment with folding $(\tilde{A},F)$ either sustains that $|\tilde{A}| \leqslant 1$ and $F=\emptyset$ (in the cases where both input strings are empty, one of the strings is empty and the other is of length 1, or both strings are of length 1 and the alignment contains a single column), or $|\tilde{A}|>1$ and $(1,|\tilde{A}|) \in F$.

Let $\tilde{L}^c(i,j;i',j')$ and $\tilde{L}^p(i,j;i',j')$ denote the maximum scores (or the solutions) of a co-terminus and a partitionable alignment with folding of $\tilde{S}_{i,j;i',j'}$, respectively (if $Q_{i,j;i',j'}=\emptyset$, define $\tilde{L}^p(i,j;i',j')=-\infty$). Then,

$$\tilde{L}\big(i,j;i',j'\big)=\max\big\{\tilde{L}^c\big(i,j;i',j'\big),\tilde{L}^p\big(i,j;i',j'\big)\big\}. \tag{5.1}$$

For sub-instances of the form $\tilde{S}_{i,i-1;i',i'-1}$ (empty sub-instances), $\tilde{S}_{i,i;i',i'-1}$, and $\tilde{S}_{i,i-1;i',i'}$, there is no split-point by definition, and $\tilde{L}^p$ for such instances is $-\infty$. Every such sub-instance has a single (co-terminus) alignment with folding, yielding the following scores: $\tilde{L}(i,i-1;i',i'-1)=\textsc{score}((\varepsilon;\varepsilon),\emptyset)=0$ (where ε denotes the empty string), $\tilde{L}(i,i;i',i'-1)=\textsc{score}((s_i;-),\emptyset)=\delta(s_i;-)$, and $\tilde{L}(i,i-1;i',i')=\textsc{score}((-;s'_{i'}),\emptyset)=\delta(-;s'_{i'})$. For sub-instances of the form $\tilde{S}_{i,i;i',i'}$ there is a single co-terminus alignment with folding $((s_i;s'_{i'}),\emptyset)$, yielding the score $\tilde{L}^c(i,i;i',i')=\textsc{score}((s_i;s'_{i'}),\emptyset)=\delta(s_i;s'_{i'})$, and two partitionable alignments with foldings $((s_i-;-s'_{i'}),\emptyset)$ and $((-s_i;s'_{i'}-),\emptyset)$, both giving the same score $\tilde{L}^p(i,i;i',i')=\textsc{score}((s_i-;-s'_{i'}),\emptyset)=\textsc{score}((-s_i;s'_{i'}-),\emptyset)=\delta(s_i;-)+\delta(-;s'_{i'})$. Call sub-instances as described above *trivial*.

A straightforward adaptation of Eqs. (2.2) and (2.3) gives the recursive computation of $\tilde{L}^c$ and $\tilde{L}^p$ for non-trivial instances:

$$\tilde{L}^c\big(i,j;i',j'\big)=\max\left\{\begin{array}{l}\tilde{L}\big(i+1,j-1;i'+1,j'-1\big)+\tau\big(\big(s_i;s'_{i'}\big),\big(s_j;s'_{j'}\big)\big),\\ \tilde{L}\big(i+1,j-1;i',j'\big)+\tau\big((s_i;-),(s_j;-)\big),\\ \tilde{L}\big(i,j;i'+1,j'-1\big)+\tau\big(\big(-;s'_{i'}\big),\big(-;s'_{j'}\big)\big)\end{array}\right\}, \tag{5.2}$$

$$\tilde{L}^p\big(i,j;i',j'\big)=\max_{(q,q')\in Q_{i,j;i',j'}}\big\{\tilde{L}\big(i,q-1;i',q'-1\big)+\tilde{L}\big(q,j;q',j'\big)\big\}. \tag{5.3}$$

Next, we adopt the concept of OCT and STEP instances described in the previous sections to the context of the SAF problem. Call a sub-instance $\tilde{S}_{i,j;i',j'}$ an OCT if every optimal alignment with folding of $\tilde{S}_{i,j;i',j'}$ is co-terminus. Note that trivial sub-instances $\tilde{S}_{i,j;i',j'}$ with $|S_{i,j}|+|S'_{i',j'}| \leqslant 1$ have no partitionable alignments with foldings, an thus are OCTs by definition. A trivial sub-instance $\tilde{S}_{i,i;i',i'}$ is an OCT if its co-terminus solution $\tilde{L}^c(i,i;i',i')=\delta(s_i;s'_{i'})$ is strictly greater than its partitionable solution $\tilde{L}^c(i,i;i',i')=\delta(s_i;-)+\delta(-;s'_{i'})$.

Call a sub-instance $\tilde{S}_{i,j;i',j'}$ a STEP if $\tilde{L}(i,j;i',j')>\tilde{L}(i,q-1;i',q'-1)+\tilde{L}(q,j;q',j')$, for $(q,q')\in\{(i+1,i'+1),(i+1,i'),(i,i'+1)\}$. In other words, $\tilde{S}_{i,j;i',j'}$ is a STEP if it has no optimal alignment with folding which admits a split-point (q,q'), such that the prefix $\tilde{S}_{i,q-1,i',q'-1}$ is trivial.

Define the subset of split-points $Q^{\text{step-oct}}_{i,j;i'j'} \subseteq Q_{i,j;i',j'}$ with respect to $\tilde{S}_{i,j;i',j'}$:

$$Q^{\text{step-oct}}_{i,j;i'j'}=\big\{\big(q,q'\big)\in Q_{i,j;i',j'}\colon\ \tilde{S}_{i,q-1;i',q'-1} \text{ is a STEP and } \tilde{S}_{q,j;q',j'} \text{ is an OCT}\big\}.$$

Similarly to the single strand variant, we show how to refine the computation of $\tilde{L}^p(i, j; i', j')$:

$$\tilde{L}^p(i, j; i', j') = \max_{(q,q') \in Q^{\text{step-oct}}_{i,j;i'j'} \cup \{(i+1,i'+1),(i+1,i'),(i+1,i')\}} \left\{ \tilde{L}(i, q-1; i', q'-1) + \tilde{L}(q, j; q', j') \right\}. \tag{5.4}$$

Proof. The proof is a straightforward adaptation of the proof of Lemma 6 to the SSF variant. □

5.3. A space efficient algorithm for the SAF problem

We next apply the same space reduction technique presented in Section 3.2 to the SAF problem.

Lemma 9. *For a sub-instance $\tilde{S}_{i,j;i',j'}$, it is possible to compute $\tilde{L}(i, j; i', j')$ by examining only those values $\tilde{L}(a, b; a', b')$ corresponding to strict sub-instances of $\tilde{S}_{i,j;i',j'}$ which are either OCTs, trivial, or sustain that $a = i$ or $a = i + 1$.*

Proof. Immediate from Eqs. (5.1), (5.2) and (5.4), and the definition of $Q^{\text{step-oct}}_{i,j;i'j'}$. □

Definition 8. For an SAF instance $\tilde{S}$, $\tilde{Z}(\tilde{S})$ is the number of sub-instances of $\tilde{S}$ which are OCTs.

When context is clear, we write $\tilde{Z}$ instead of $\tilde{Z}(\tilde{S})$. A dynamic programming algorithm similar to the algorithm presented in [38] can now be designed. The algorithm maintains a "super-table" N of size $n \times n$, for which each entry $N_{i,j}$ contains an "internal table" of size $m \times m$. We denote by $N_{i,j}[i', j']$ the (i', j')-entry in the internal table $N_{i,j}$, where the value of this entry corresponds to the solution for the sub-instance $\tilde{S}_{i,j;i',j'}$. According to Lemma 9, iterating over the rows of the super-table N starting from row n and decreasing the row index i in each iteration, the algorithm needs to maintain in memory only $\Theta(\tilde{Z})$ scores of OCT sub-instances, in addition to scores in the most recently computed two rows i and $i + 1$ of the super-table. Note that each internal table requires $O(m^2)$ space, thus the space required for maintaining two rows in the super-table is $O(m^2n)$ (the asymmetry rises from the manner in which the entries in N are indexed, and the order of the loops in lines 1 and 2 of Algorithm 5). Therefore, the total space complexity of the algorithm is $O(m^2n + \tilde{Z})$ (we can assume w.l.o.g. that $m \leqslant n$). The time complexity is dictated by the number of examined split-points along the algorithm's run. For every one of the $\Theta(n^2m^2)$ sub-instances $\tilde{S}_{i,j;i',j'}$, the algorithm examines explicitly the three split-points $(i + 1, i' + 1)$, $(i, i' + 1)$, and $(i + 1, i')$, in addition to split-points in the set $Q^{\text{step-oct}}_{i,j;i'j'}$. Here, each one of the $\tilde{Z}$ OCT suffix sub-instances is examined against at most $\tilde{P}$ STEP prefix sub-instances, where $\tilde{P}$ denotes the maximum number of STEP sub-instances $\tilde{S}_{i,j;i',j'}$, taken over all pairs j, j' for $1 \leqslant j \leqslant n$ and $1 \leqslant j' \leqslant m$ (note that $\tilde{P} \leqslant nm$ and $\tilde{Z} \leqslant nm(P + 3)$). Thus, the time complexity of the algorithm is $O(n^2m^2 + \tilde{P}\tilde{Z})$.

Algorithm 5 below gives the pseudo-code for the algorithm described above. Fig. 11 illustrates its run, and Lemma 10 states its time and space complexities.

Algorithm 5: RNA alignment and folding

```
input  : An SAF instance $\tilde{S} = (S, S')$, where $|S| = n$ and $|S'| = m$
output: $\tilde{L}(1, n; 1, m)$
1  for $i \leftarrow n$ down to 1 do
2      for $i' \leftarrow m$ down to 1 do
3          set $N_{i,i-1}[i', i'-1]$, $N_{i,i-1}[i', i']$, $N_{i,i}[i', i'-1]$, and $N_{i,i}[i', i']$ explicitly to the solutions for the corresponding trivial sub-instances;
4          Mark $\tilde{S}_{i,i-1;i',i'}$ and $\tilde{S}_{i,i;i',i'-1}$ as OCTs. If $\tilde{L}^c(i, i; i', i') > \tilde{L}^p(i, i; i', i')$, mark $\tilde{S}_{i,i;i',i'}$ as an OCT;
5          for $j \leftarrow i$ to $n$ do
6              for $j' \leftarrow i'$ to $m$ do
7                  for all $(q, q') \in \{(i+1, i'+1), (i+1, i'), (i, i'+1)\}$ do
8                      set $N_{i,j}[i', j'] \leftarrow \max\{N_{i,j}[i', j'], N_{i,q-1}[i', q'-1] + N_{q,j}[q', j']\}$;
9                  compute $\tilde{L}^c(i, j; i', j')$ according to Eq. (5.2);
10                 if $N_{i,j}[i', j'] < \tilde{L}^c(i, j; i', j')$ then
11                     set $N_{i,j}[i', j'] \leftarrow \tilde{L}^c(i, j; i', j')$;
12                     mark $\tilde{S}_{i,j;i',j'}$ as an OCT;
13                 if $\tilde{S}_{i,j;i',j'}$ is a STEP then
14                     for all $(p, p')$ s.t. $\tilde{S}_{j+1,p;j'+1,p'}$ is an OCT do
15                         set $N_{i,p}[i', p'] \leftarrow \max\{N_{i,p}[i', p'], N_{i,j}[i', j'] + N_{j+1,p}[j'+1, p']\}$;
16         discard from memory the values in all entries $N_{i+1,j}[i', j']$ for all $i \leqslant j \leqslant n$ and $i' \leqslant j' \leqslant m$ which do not correspond to OCT instances;
17 return $N_{1,n}[1, m]$;
```

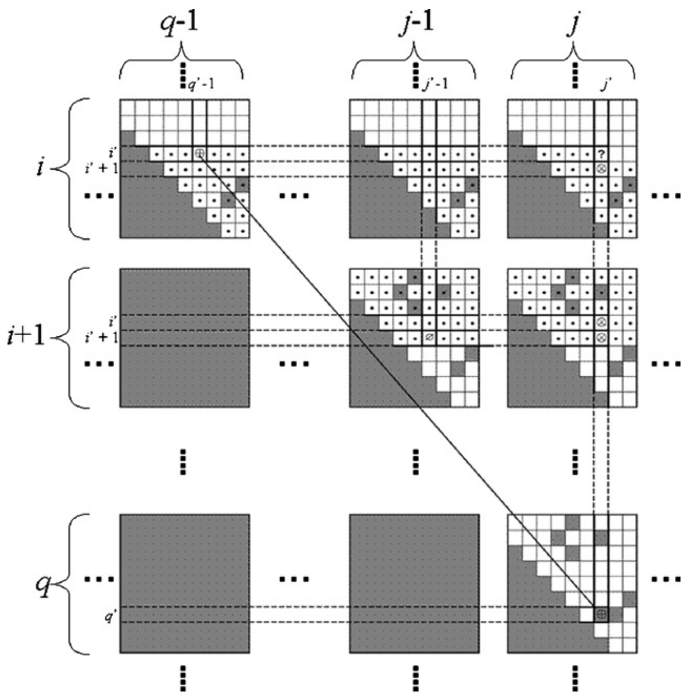

Fig. 11. A schematic illustration of the data structure maintained by Algorithm 5. In order to compute the entry $N_{i,j}[i', j']$, the algorithm needs to examine the value in $N_{i+1,j-1}[i'+1, j'-1]$ for the computation of $\tilde{L}^c(i, j; i', j')$ (marked with a $\oslash$ symbol), and the values in $N_{i,j}[i'+1, j']$, $N_{i+1,j}[i', j']$, and $N_{i+1,j}[i'+1, j']$ (marked with $\otimes$ symbols) for the computation of $\tilde{L}^p(i, j; i', j')$. In addition, the algorithm examines entry-pairs $N_{i,q-1}[i', q'-1]$ and $N_{q,j}[q', j']$, such that the prefix $\tilde{S}_{i,q-1;i',q'-1}$ is a STEP sub-instance, and the suffix $\tilde{S}_{q,j;q',j'}$ is an OCT sub-instance (marked with $\oplus$ symbols). The algorithm maintains only entries in rows i and $(i+1)$ of the super-table N (marked with "dot" symbols) and entries which correspond to OCT sub-instances (illustrated by grayed entries in the upper triangle). All entries starting from the $(i+2)$th row in the super-table that do not correspond to OCT sub-instances are discarded from memory.

Lemma 10. *Algorithm 5 computes $\tilde{L}(1, n; 1, m)$ of a given SAF instance $\tilde{S}$ with strings of lengths n and m, in time and space complexities of $O(n^2m^2 + \tilde{P}\tilde{Z})$ and $O(m^2n + \tilde{Z})$, respectively.*

6. Experimental results

The sparsification techniques presented in this paper were tested for the SSF base-pairing maximization problem. Four algorithm variants were implemented: the *naive folder* which applies no sparsification, the *OCT folder* which applies the sparsification technique of [34] (presented in Section 3), the *STEP-OCT folder* which applies the sparsification technique presented in Section 4.1, and the *STEP-ENC folder* (for "step-encoding") which applies the sparsification technique presented in Section 4.2. All algorithms except for the naive algorithm maintain sparse dynamic programming tables. Java source code for all four variants is available at http://www.cs.bgu.ac.il/~zakovs/RNAfold/SparseFold.zip.

Two experiments were conducted. In the first experiment, the performance of the sparse algorithms was measured for randomly chosen sequences of different lengths. For each length $l = 50, 100, 150, \ldots, 1000$, a set of 100 random sequences was generated (assuming equal nucleotide distributions) and folded by all four algorithms. Fig. 12 illustrates the average improvements of the sparse algorithms over the naive algorithm. The *OCT*, *STEP-OCT*, and *STEP-ENC* curves show the average proportions of split-points examined by each one of the sparse algorithms, with respect to all $\Theta(n^3)$ split-points examined by the naive algorithm. The Z curve demonstrates the space reduction obtained by the sparse algorithms, by showing the average proportions of maintained entries in the sparse tables with respect to all $\Theta(n^2)$ entries maintained in the tables which are used by the naive algorithm.

As can be seen, all sparse algorithms perform significantly better than the naive algorithm in terms of both time and space complexities, where the *STEP-ENC* variant is consistently the fastest variant. For sufficiently long sequences, the proportion of maintained entries is about 2%, where the proportion of split-points examined by the *STEP-ENC* algorithm is about 1%.

The second experiment examines the behavior of the sparse algorithms for different values of L. A set of 10 000 random sequences, each of length 500 bases, was folded by all four algorithms. The sequences where then divided into bins according to their folding scores, where the ith bin corresponds to sequences for which the folding score is between $10i + 1$ and $10(i + 1)$. The average performance of each one of the algorithms on all sequences in each bin was measured. In order to increase the likelihood of obtaining sequences with low scores, the proportion of nucleotides of type A was biased and set to $\frac{j}{10\,000}$ for the jth sequence. Other than that, the sequences where randomly chosen.

Fig. 13 presents the average improvements of the sparse algorithms over the naive algorithm in this experiment, showing the same entities measured in Fig. 12. The experiment demonstrates that, as expected, for lower values of L the computation

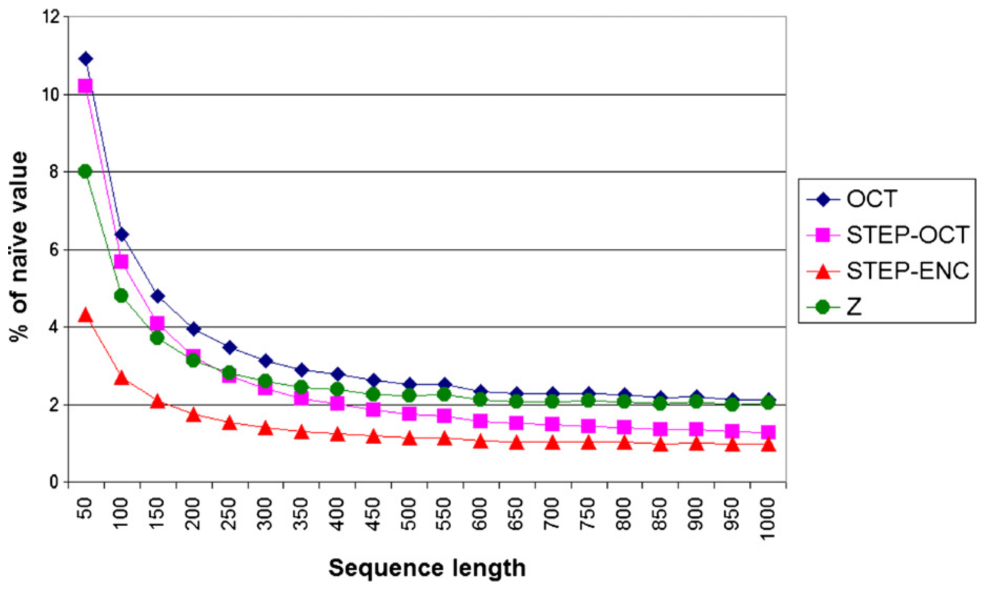

Fig. 12. The performance of the sparse algorithms over sequences of different lengths. The *OCT*, *STEP-OCT*, and *STEP-ENC* curves illustrate the average proportions of split-points examined by each one of the sparse algorithms with respect to all $\Theta(n^3)$ split-points examined by the naive algorithm. The *Z* curve illustrate the average proportions of entries maintained in the sparse tables with respect to all $\Theta(n^2)$ entries maintained by the naive algorithm.

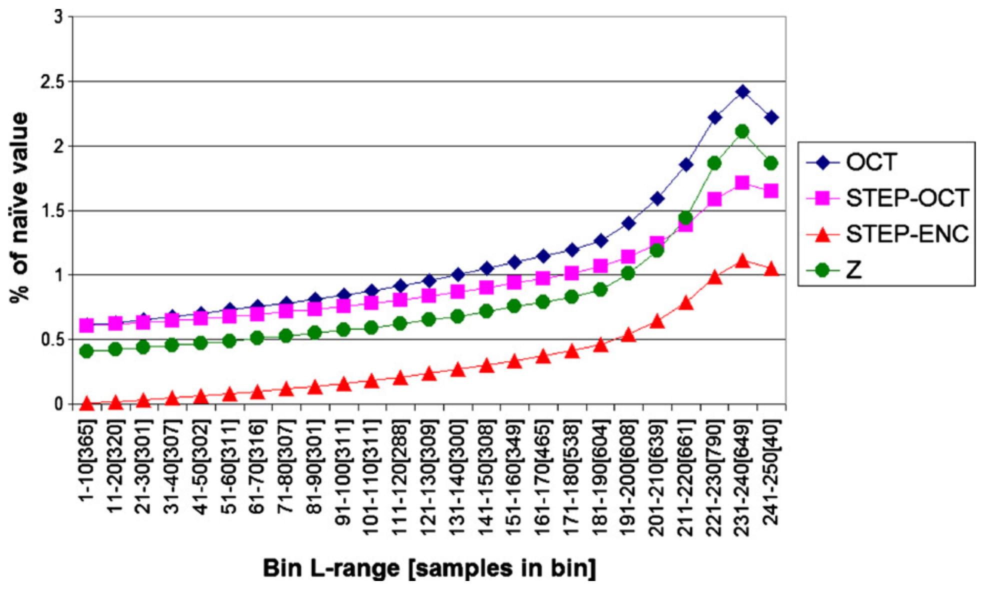

Fig. 13. The performance of the sparse algorithms over sequences of different scores. 10 000 random sequences (with biased '*A*' proportion), each of length 500 bases, were divided into bins according to their folding scores (L). The *OCT*, *STEP-OCT*, and *STEP-ENC* curves illustrate the average proportions of split-points examined by each one of the sparse algorithms with respect to all $\Theta(n^3)$ split-points examined by the naive algorithm. The *Z* curve illustrates the average proportions of entries maintained in the sparse tables with respect to all $\Theta(n^2)$ entries maintained by the naive algorithm.

is more "sparse" and requires less split-point examinations and less memory usage. Notably, the *STEP-ENC* algorithm is extremely efficient for most values of L, examining less than 0.5% of all split-points.

7. Concluding discussion

This paper presents several new techniques for sparsifying RNA folding algorithms, in order to reduce their time and space complexities in practice. The space complexity improvement is obtained by a simple modification of an algorithm by Wexler et al. [34], which used sparsification for improving the time complexity of RNA folding. This modification allows the algorithm to maintain in memory only a small subset of solutions computed for sub-instances. The time complexity improvement is obtained by using stronger sparsification criteria than that presented in [34]. Forward dynamic programming is applied in order to avoid redundant computations. These sparsification techniques are described for both the SSF and SAF problems. We show experimental results that demonstrate the applicability of these techniques with respect to the basic SSF base-pairing maximization variant.

The presented techniques may also be applied to more realistic scoring schemes than those presented here. For example, the *minimum free energy* scheme (mfe) [41,24] scores differently stacked base-pairs and loop-closing base-pairs, thus the computation of $L^c(i, j)$ needs as an input both $L^c(i+1, j-1)$ and $L^p(i+1, j-1)$. As a matter of fact, it even further distinguishes between different types of loops and maintains several types of structure-constrained scores, in addition to co-terminus and partitionable scores. Since the mfe scoring scheme still allows for the computation of $L^c(i, j)$ to be conducted in a constant time, the presented algorithmic framework for improving the time complexity still holds for this scheme. In addition, since usually this computation examines only scores of sub-instances $S_{i',j'}$ of $S_{i,j}$, where $i' < i + D$ for some constant D, the space complexity improvement also scales up to the more realistic scoring schemes, where the algorithm needs to maintain in memory scores of OCT sub-instances, and D rows of the full table. In the computation of $L^p(i, j)$, it is possible that some energetic term E_q, reflecting *coaxial stacking energy*, is added when computing a score which corresponds to a split-point q (i.e. $L(i, q-1) + L(q, j) + E_q$). In such cases, the concept of OCT instances can be relaxed, where a sub-instance $S_{i,j}$ is considered an OCT if $L^c(i, j) > L(i, q-1) + L(q, j) + E_q$ for every $q \in Q_{i,j}$. Such a relaxation ensures that the computation of $L^p(i, j)$ as defined in Eq. (4.1) would still yield a correct value, while avoiding many redundant computations.

The notion of *forward computation* for sparsifying dynamic programming was previously applied to several related problems. In [17], Graham et al. used forward computation in a sparse *context free grammar recognition* algorithm. There, for every sub-sentence $S_{i,q-1}$ that can be derived from a non-terminal A in the grammar, and for every CFG rule of the form $C \to AB$, the algorithm identifies all sub-sentences $S_{q,j}$ which can be derived from the non-terminal B, and concludes that $S_{i,j}$ can be derived from C. Jansson et al. [22] used forward computation for the problem of aligning an RNA sequence to a structured sequence. Their computation also involves the examinations of different splittings of every sub-instance, where the criterion for prefixes induced by such splittings is similar to the so called STEP criterion here, and the suffix criterion was dictated from the input.

Other heuristic sparsification techniques were previously applied, mainly to the SAF problem. The *FOLDALIGN* software package [18,32] is an example of one of several tools which apply heuristic sparsifications to the SAF problem, which are somewhat (though weakly) related to those presented here. The FOLDALIGN algorithm allows the user to limit the solution computations to only those sub-instances for which the length difference between the two strings is bounded, and in addition, to restrict the computation to examine only split-points which are of bounded distance from the sub-instance's end-points. These two sparsification techniques considerably reduce both the time and space requirements of the algorithm, but at the cost of potentially missing optimal solutions. In addition, this implementation also applies forward computation, which sparsifies split-point examinations according to weaker criteria than those presented here. A split-point (q, q') is examined with respect to a sub-instance $\tilde{S}_{i,j;i',j}$ if the following two conditions hold. First, the suffix $\tilde{S}_{q,j;q',j}$ has *at least one* optimal solution (where optimality is conditioned by the assumption that no other heuristic sparsification is applied) in which the right endpoint of the alignment is paired (to some other column of the alignment). This is a weaker criterion than a symmetric definition of the STEP criterion, which requires that in *all optimal* solutions the right alignment endpoint is paired. The second condition is that the prefix $\tilde{S}_{i,q-1;i',q'-1}$ is required to have *at least one* optimal solution (again, optimality is conditioned by the assumption that no other heuristic sparsification is applied), in which the left and right alignment endpoints are paired to each other. This is a weaker criterion than the OCT criterion, which requires that in *all optimal* solutions the two endpoints are paired to each other.

As a next step, we intend to extend our implementation of the presented techniques to additional RNA folding variants (such as SSF variants with other scoring schemes, as well as to the SAF problem and to the RNA–RNA interaction problem [2, 9]). We also plan to further study the adaptation of these approaches to several related problems with similar combinatorial properties. One family of problems which falls into this category is the family of context free grammar (CFG) recognition and parsing problems [10,23,36,6]. It is well known that SSF can be viewed as a special case of the probabilistic CFG parsing problem (see e.g. [27,13]), where the classical algorithms for these problems are of similar structure. Another example of a problem with a structure which is similar to that of SSF is the problem of finding the maximum independent set in a circle graph [29,4].

Acknowledgements

We thank the anonymous JDA referees for their many helpful comments. The work of Shay Zakov and Michal Ziv-Ukelson was partially supported by the Frankel Center for Computer Science at Ben Gurion University of the Negev. Rolf Backofen received funding from the German Research Foundation (DFG grant BA 2168/2-1 SPP 1258), and from the German Federal Ministry of Education and Research (BMBF grant 0313921 FRISYS).

References

[1] Tatsuya Akutsu, Approximation and exact algorithms for RNA secondary structure prediction and recognition of stochastic context-free languages, Journal of Combinatorial Optimization 3 (1999) 321–336.
[2] Can Alkan, Emre Karakoç, Joseph H. Nadeau, Süleyman Cenk Sahinalp, Kaizhong Zhang, RNA–RNA interaction prediction and antisense RNA target search, Journal of Computational Biology 13 (2) (2006) 267–282.
[3] Mirela Andronescu, Anne Condon, Holger H. Hoos, David H. Mathews, Kevin P. Murphy, Efficient parameter estimation for RNA secondary structure prediction, Bioinformatics 23 (13) (2007) i19–i28.
[4] Alberto Apostolico, Mikhail J. Atallah, Susanne E. Hambrusch, New clique and independent set algorithms for circle graphs, Discrete Applied Mathematics 36 (1) (March 1992) 1–24.

[5] Rolf Backofen, Dekel Tsur, Shay Zakov, Michal Ziv-Ukelson, Sparse RNA folding: Time and space efficient algorithms, in: Gregory Kucherov, Esko Ukkonen (Eds.), Combinatorial Pattern Matching, 20th Annual Symposium, Proceedings, CPM 2009, Lille, France, June 22–24, 2009, in: Lecture Notes in Computer Science, vol. 5577, Springer, 2009, pp. 249–262.

[6] James K. Baker, Trainable grammars for speech recognition, The Journal of the Acoustical Society of America 65 (S1) (1979) S132.

[7] Athanasius F. Bompfünewerer Consortium, Rolf Backofen, Stephan H. Bernhart, Christoph Flamm, Claudia Fried, Guido Fritzsch, Jorg Hackermuller, Jana Hertel, Ivo L. Hofacker, Kristin Missal, Axel Mosig, Sonja J. Prohaska, Dominic Rose, Peter F. Stadler, Andrea Tanzer, Stefan Washietl, Sebastian Will, RNAs everywhere: genome-wide annotation of structured RNAs, Journal of Experimental Zoology Part B: Molecular and Developmental Evolution 308 (1) (2007) 1–25.

[8] Timothy M. Chan, More algorithms for all-pairs shortest paths in weighted graphs, SIAM Journal of Computing 39 (5) (2010) 2075–2089.

[9] Hamidreza Chitsaz, Raheleh Salari, S. Cenk Sahinalp, Rolf Backofen, A partition function algorithm for interacting nucleic acid strands, Bioinformatics 25 (12) (2009) i365–i373.

[10] John Cocke, Jacob T. Schwartz, Programming Languages and Their Compilers, Courant Institute of Mathematical Sciences, New York, 1970.

[11] Chuong B. Do, Daniel A. Woods, Serafim Batzoglou, CONTRAfold: RNA secondary structure prediction without physics-based models, Bioinformatics 22 (14) (2006) e90-8.

[12] Robin Dowell, Sean Eddy, Evaluation of several lightweight stochastic context-free grammars for RNA secondary structure prediction, BMC Bioinformatics 5 (1) (2004) 71.

[13] Richard Durbin, Sean R. Eddy, Anders Krogh, Graeme J. Mitchison, Biological Sequence Analysis: Probabilistic Models of Proteins and Nucleic Acids, Cambridge University Press, 1998.

[14] Yelena Frid, Dan Gusfield, A simple practical and complete $O(\frac{n^3}{\log n})$-time algorithm for RNA folding using the *four-russians* speedup, Algorithms for Molecular Biology 5 (1) (2010) 5–13.

[15] Yelena Frid, Dan Gusfield, A worst-case and practical speedup for the RNA co-folding problem using the *four-russians* idea, in: Vincent Moulton, Mona Singh (Eds.), Algorithms in Bioinformatics, 10th International Workshop, WABI 2010, Proceedings, Liverpool, UK, September 6–8, 2010, in: Lecture Notes in Computer Science, vol. 6293, Springer, 2010, pp. 1–12.

[16] Paul P. Gardner, Robert Giegerich, A comprehensive comparison of comparative RNA structure prediction approaches, BMC Bioinformatics 5 (2004) 140.

[17] Susan L. Graham, Michael A. Harrison, Walter L. Ruzzo, An improved context-free recognizer, ACM Transactions on Programming Languages and Systems 2 (3) (1980) 415–462.

[18] Jacob Hull Havgaard, Rune B. Lyngso, Gary D. Stormo, Jan Gorodkin, Pairwise local structural alignment of RNA sequences with sequence similarity less than 40%, Bioinformatics 21 (9) (2005) 1815–1824.

[19] Daniel S. Hirschberg, A linear space algorithm for computing maximal common subsequences, Communications of the ACM 18 (6) (1975) 341–343.

[20] Daniel S. Hirschberg, Algorithms for the longest common subsequence problem, JACM 24 (1977) 664–675.

[21] Ivo L. Hofacker, Vienna RNA secondary structure server, Nucleic Acids Research 13 (2003) 3429–3431.

[22] Jesper Jansson, See-Kiong Ng, Wing-Kin Sung, Hugo Willy, A faster and more space-efficient algorithm for inferring arc-annotations of RNA sequences through alignment, Algorithmica 46 (2) (2006) 223–245.

[23] Tadao Kasami, An efficient recognition and syntax analysis algorithm for context-free languages, Technical Report AFCRL-65-758, Air Force Cambridge Res. Lab., Bedford, Mass., 1965.

[24] David H. Mathews, Jeffrey Sabina, Michael Zuker, Douglas H. Turner, Expanded sequence dependence of thermodynamic parameters improves prediction of RNA secondary structure, J. Mol. Biol. 288 (1999) 911–940.

[25] David H. Mathews, Douglas H. Turner, Dynalign: an algorithm for finding the secondary structure common to two RNA sequences, Journal of Molecular Biology 317 (2) (2002) 191–203.

[26] Ruth Nussinov, Ann B. Jacobson, Fast algorithm for predicting the secondary structure of single-stranded RNA, PNAS 77 (11) (1980) 6309–6313.

[27] Yasubumi Sakakibara, Michael Brown, Richard Hughey, I. Saira Mian, Kimmen Sjolander, Rebecca C. Underwood, David Haussler, Stochastic context-free grammars for tRNA modeling, Nucleic Acids Research 22 (23) (1994) 5112–5120.

[28] David Sankoff, Simultaneous solution of the RNA folding, alignment and protosequence problems, SIAM Journal on Applied Mathematics 45 (5) (1985) 810–825.

[29] Kenneth J. Supowit, Finding a maximum planar subset of a set of nets in a channel, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 6 (1) (1987) 93–94.

[30] Ignacio Tinoco, Olke C. Uhlenbeck, Mark D. Levine, Estimation of secondary structure in ribonucleic acids, Nature 230 (1971) 362–367.

[31] Ignacio Tinoco, Philip N. Borer, Barbara Dengler, Mark D. Levine, Olke C. Uhlenbeck, Donald M. Crothers, Jay Gralla, Improved estimation of secondary structure in ribonucleic acids, Nature New Biology 246 (1973) 40–41.

[32] Elfar Torarinsson, Jacob Havgaard, Jan Gorodkin, Multiple structural alignment and clustering of RNA sequences, Bioinformatics 23 (8) (2007) 926–932.

[33] Michael S. Waterman, Temple F. Smith, RNA secondary structure: a complete mathematical analysis, Mathematical Biosciences 42 (1978) 257–266.

[34] Ydo Wexler, Chaya Zilberstein, Michal Ziv-Ukelson, A study of accessible motifs and RNA folding complexity, Journal of Computational Biology 14 (6) (2007) 856–872.

[35] Sebastian Will, Kristin Reiche, Ivo L. Hofacker, Peter F. Stadler, Rolf Backofen, Inferring non-coding RNA families and classes by means of genome-scale structure-based clustering, PLOS Computational Biology 3 (4) (2007) e65.

[36] Daniel H. Younger, Recognition and parsing of context-free languages in time n^3, Information and Control 10 (2) (1967) 189–208.

[37] Shay Zakov, Dekel Tsur, Michal Ziv-Ukelson, Reducing the worst case running times of a family of RNA and CFG problems, using Valiant's approach, in: Vincent Moulton, Mona Singh (Eds.), Algorithms in Bioinformatics, 10th International Workshop, WABI 2010, Proceedings, Liverpool, UK, September 6–8, 2010, in: Lecture Notes in Computer Science, vol. 6293, Springer, 2010, pp. 65–77.

[38] Michal Ziv-Ukelson, Irit Gat-Viks, Ydo Wexler, Ron Shamir, A faster algorithm for simultaneous alignment and folding of RNA, Journal of Computational Biology 17 (8) (2010) 1051–1065.

[39] Michael Zuker, Computer prediction of RNA structure, Methods Enzymol. 180 (1989) 262–288.

[40] Michael Zuker, Mfold web server for nucleic acid folding and hybridization prediction, Nucleic Acids Research 13 (2003) 3406–3415.

[41] Michael Zuker, P. Stiegler, Optimal computer folding of large RNA sequences using thermodynamics and auxiliary information, Nucleic Acids Research 9 (1) (1981) 133–148.