Local Exact Pattern Matching for Non-Fixed RNA Structures

Mika Amit, Rolf Backofen, Steffen Heyne, Gad M. Landau, Mathias Möhl, Christina Otto, and Sebastian Will

Abstract—Detecting local common sequence-structure regions of RNAs is a biologically important problem. Detecting such regions allows biologists to identify functionally relevant similarities between the inspected molecules. We developed dynamic programming algorithms for finding common structure-sequence patterns between two RNAs. The RNAs are given by their sequence and a set of potential base pairs with associated probabilities. In contrast to prior work on local pattern matching of RNAs, we support the breaking of arcs. This allows us to add flexibility over matching only fixed structures; potentially matching only a similar subset of specified base pairs. We present an $O(n^3)$ algorithm for local exact pattern matching between two nested RNAs, and an $O(n^3 \log n)$ algorithm for one nested RNA and one bounded-unlimited RNA. In addition, an algorithm for approximate pattern matching is introduced that for two given nested RNAs and a number k, finds the maximal local pattern matching score between the two RNAs with at most k mismatches in $O(n^3k^2)$ time. Finally, we present an $O(n^3)$ algorithm for finding the most similar subforest between two nested RNAs.

Index Terms—Pattern matching, RNA local similarity, tree local similarity, sequence-structure matching

1 Introduction

Ribonucleic acid (RNA) is a chain of nucleotides present in the cells of all living organisms. Most RNAs are single-stranded. RNA strands have a backbone made from groups of phosphates and ribose sugar, to which one of four bases can attach (Adenine, Cytosine, Guanine, and Uracil). The bases are linked together by their phosphodiester bonds (usually referred to as *backbone connection*), and interact with each other using hydrogen bonds (usually referred to as *bond connections*), forming the RNA structure. We further denote two bases that are connected by bond connection as *base pairs* and a base that has only backbone connections as a *single base*. RNA performs important functions for living organisms, ranging from the regulation of gene expression to assistance with copying genes. The important role that small RNA take in operating the cell's control has been discovered recently and it was referred to as the breakthrough of the year 2002 in Science magazine [6].

Finding similarity between sequences and structures of RNAs is an important and well studied task. The reason is that the activity and functionality of RNA is determined by its sequence and mainly by its secondary and tertiary structure [17]. Furthermore, the structure of a molecule is usually much more preserved during evolution than its sequence alone. Thus, analyzing and comparing the secondary (and tertiary) structures of given RNAs plays a very important role in the RNA research.

The complexity of RNA secondary structure is defined by the amount and order of the *base pairs* that it contains. It is commonly categorized as follows:

- *Plain*: no base pairs at all (this is the primary structure of the RNA)
- *Nested*: each base can be connected by a bond connection to at most one other base, and there are no crossing base pairs
- *Crossing*: each base can be maximally connected by a bond connection to one other base
- *Bounded-Unlimited*: each base can be maximally connected by a bond connection to a constant number of other bases
- *Unlimited*: no restrictions on the base pairs

Fig. 1 demonstrates three ways of visualizing RNA nested structure. Throughout this work we use the arc-annotated sequence, that represents both the sequence and the structure of the RNA by adding an arc between each two bases that have a bond connection. This representation can describe both nested and bounded-unlimited RNA structures (see Fig. 1).

There are several approaches to compute the similarity between two given RNAs, among them are tree similarity algorithms such as edit distance [5], [7], [8], [10], [14], [23], [24], alignment [2], [13], [18], [20], and LAPCS [9], [11], [15]. An edit distance between two ordered trees, T_1 and T_2, is a set of edit operations applied on T_1 in order to turn it into T_2. The optimal edit distance between two trees is such set of edit operations with minimum cost. Tree alignment restricts the edit operations such that insertions are made for both T_1 and T_2 to make them isomorphic, and then relabeling of the nodes is done (see [3], [4] for thorough surveys). Zhang and Shasha [24] present an edit distance algorithm

- M. Amit is with the Department of Computer Science, University of Haifa, Mount Carmel, Haifa 3498838, Israel. E-mail: mika.amit2@gmail.com.
- R. Backofen is with the Bioinformatics, Institute of Computer Science, Albert-Ludwigs-Universität, Freiburg, Germany and Center for Biological Signaling Studies, Albert-Ludwigs-Universität, Freiburg 79110, Germany.
- S. Heyne, M. Möhl and C. Otto are with the Bioinformatics, Institute of Computer Science, Albert-Ludwigs-Universität, Freiburg 79110, Germany. E-mail: heyne@ie-freiburg.mpg.de, info@mamoworld.com.
- G.M. Landau is with the Department of Computer Science, University of Haifa, Mount Carmel, Haifa, Israel and Department of Computer Science and Engineering, NYU Polytechnic School of Engineering, New York University, Brooklyn, NY 11201. E-mail: landau@univ.haifa.ac.il.
- S. Will is with the Bioinformatics, Institute of Computer Science, Albert-Ludwigs-Universität, Freiburg, Germany and CSAIL and Mathematics Department, MIT 02139, Cambridge, MA.

Manuscript received 7 Nov. 2012; revised 27 Nov. 2013; accepted 18 Dec. 2013; date of publication 1 Jan. 2014; date of current version 7 May 2014.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TCBB.2013.2297113

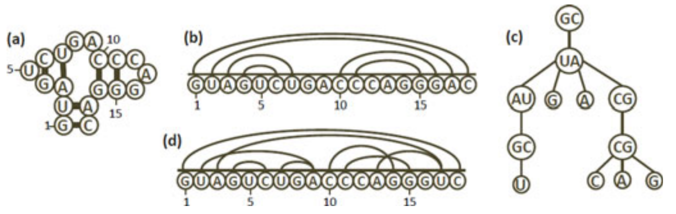
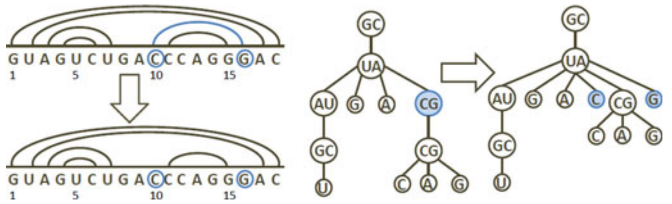


Fig. 1. RNA secondary structure representations: figures (a-c) represent the same RNA sample of length 18 with depth 5. (a) schematic two dimensional description of RNA folding (b) arc annotated sequence (c) an ordered tree: a single base is represented as a leaf and a base pair is represented as either a leaf (if the base pair's span is 2) or as an internal node with child nodes (of the base pairs and single bases that it contains). Figure (d) represents a bounded-unlimited RNA structure with an arc-annotated sequence.


G U A G U C U G A C C C A G G G A C
1 5 10 15
G U A G U C U G A C C C A G G G A C
1 5 10 15
Fig. 2. Arc breaking operation: both representations show the result of the arc breaking operation for base pair CG in positions (10,16).

that works in $O(nm \times min\{D_1, L_1\} \times min\{D_2, L_2\})$ where n and m are the sizes of T_1 and T_2, respectively, and are defined by the number nodes in the tree ($n > m$). D_i is the depth of tree i and L_i is the number of leaves in tree i. Klein [14] presents an $O(m^2 n \log n)$ algorithm, which in some cases performs better than the previous algorithm. An optimal $O(n^3)$ decomposition algorithm for tree edit distance was given by Demaine et al. [7]. Ma et al. [25] compute the edit distance between two RNAs where at least one is of nested structure. This algorithm runs in $O(n^2 D_1 D_2)$, and an explanation of how to modify it to run in $O(n^3 \log n)$ is given.

Jiang et al. [10] present an algorithm for global edit distance between nested and crossing structures, this algorithm allows arc edit operations, such as arc breaking, arc altering and arc removing. The algorithm runs in $O(n^2 m^2)$ and can be modified to work in $O(n^2 m \log m)$ time using the technique of Klein [14]. In a similar way to our explanation in Section 5, we believe that the algorithm can be modified to run in $O(n^3)$ time for nested structures using the technique of Demaine et al. [7].

Another approach for similarity checking is finding common motifs between two RNAs. In this problem, local maximal exact sequence-structure patterns are computed. Backofen and Siebert [19] solve this problem for two fixed nested RNAs in $O(n^2)$ time. In Schmiedl et al. [21] we present a heuristic solution for bounded-unlimited structures.

In this work, we solve the problem of finding exact and approximate local common motifs between nested and bounded-unlimited structures. We are the first to present deterministic algorithms for these problems when the arc breaking operation is allowed. The basic edit operations that are allowed in our algorithms are similar to the ones of [10], and we also use the ideas of [14] and [7] in order to improve the time complexity. The problem of local pattern matching is known to be more complex than the global one, in this work we developed new techniques for finding such patterns.

Jansson and Peng [12] describe $O(n^4)$ algorithms for finding a subforest F of T_1 such that F has a minimal edit distance from T_2. The structure of F is restricted to being a *simple*, *sibling* or *closed* subforest, where a *simple* subforest is a subtree, a *sibling* subforest is a set of *simple* subforests whose roots are siblings in T_1, and *closed* is a complete subtree of T_1.

1.1 Our Results

In this work, we are looking for local exact pattern matching between two RNA molecules. We use the definitions from [19], and add an additional edit operation: *arc breaking*, which breaks a base pair into two single bases. Adding the *arc breaking* operation means that the bonds are not necessarily preserved in the common substructure. This enhancement to the pattern matching algorithm allows greater flexibility in both the input and the output. Instead of representing a fixed structure, the input can be interpreted as a set of weighted secondary structures. This is encoded by base pairs with probabilities. For this purpose we score the match of two base pairs according to their probabilities. The arc breaking operation is demonstrated in Fig. 2. In addition, the scoring functions used in our work can be modified in order to support various matching schemes. For instance, compensatory mutations, or mismatches between single bases can be treated. The formal definitions of the problems are given in Section 2.

We present a simple $O(n^4)$ algorithm for computing the local exact pattern matching between two nested RNAs (Section 3). In Section 4, we continue with an $O(n^3 \log n)$ algorithm, and in Section 6 we show how to modify the algorithm to support one nested and one bounded-unlimited input structure ($(Nested, Bounded - Unlimited)$, in short). In Section 5 we show how to improve the algorithm for $(Nested, Nested)$ RNAs to $O(n^3)$. These algorithms use $O(n^2)$ space.

The approximate matching problem is presented in Section 7. In this problem we look for pattern matching having at most k mismatch bases. In Section 7 we present an $O(n^3 k^2)$ algorithm for computing the local approximate matching between two nested RNAs with at most k mismatches. This algorithm can be also modified to work in $O(n^3 k^2 \log n)$ for $(Nested, Bounded - Unlimited)$ RNAs. The space complexity of these algorithms is $O(n^2 k)$.

In Section 8 we describe an $O(n^3)$-time and $O(n^2)$-space algorithm for computing the most similar sibling substructure between two $(Nested, Nested)$ RNAs, as defined in [12].

2 Notations and Definitions

An *RNA Sequence* (in short, *RNA*) is an ordered pair $R = (S, B)$, where $S = s_1, \ldots, s_{|S|}$, and s_i is defined over the alphabet $\Sigma = \{A, C, G, U\}$ and represents the RNA primary structure. B, the optional secondary structure, is a set of tuples $\{(a, b, p) | 1 \leq a < b \leq |S|,\ 0 < p \leq 1\}$, such that a tuple $bp = (a, b, p) \in B$ represents a hydrogen bond (a base pair) between bases a and b that exists with probability p in R. We denote a and b as the *left* and *right* endpoints of bp,

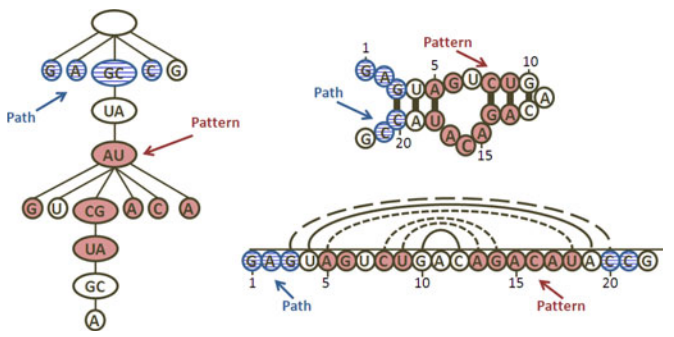


Fig. 3. Path and pattern examples in three representations of the same RNA example. A path is marked with horizontal lines and contains the bases $\{1, 2, 3, 20, 21\}$, a pattern is shadowed and contains the bases $\{5, 6, 8, 9, 13, 14, 15, 16, 17, 18\}$. Note that the pattern contains the base pairs $(5, 18)$, $(8, 14)$ and $(9, 13)$, whereas the base pairs (3, 20) and (10,12) are not included.

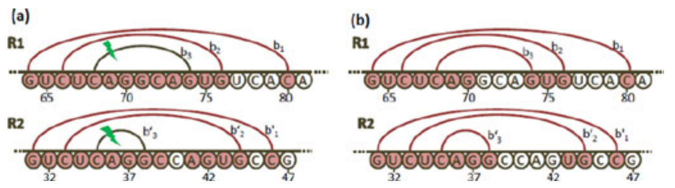


Fig. 4. Two matchings example. The figure presents two matching examples that can be defined between R_1 and R_2. In both cases the matchings are maximally extended. Note that matching (a) contains the base pairs (b_1, b'_1) and (b_2, b'_2), and matching (b) contains (b_1, b'_1), (b_2, b'_2) and (b_3, b'_3). The matching scores depend on the definition of α and β functions. Given $b_1 = (64, 80, 0.9)$, $b_2 = (66, 76, 0.6)$, $b_3 = (68, 74, 0.1)$, $b'_1 = (31, 46, 0.8)$, $b'_2 = (33, 44, 0.5)$, and $b'_3 = (35, 38, 0.3)$ and using our function definitions, $score(a) = 14 + (1.9 \cdot 1.8) + (1.6 \cdot 1.5) = 19.82$ and $score(b) = 11 + (1.9 \cdot 1.8) + (1.6 \cdot 1.5) + (1.1 \cdot 1.3) = 18.25$, thus the matching with the maximal score is (a).

respectively. A base that is neither left nor right endpoint is denoted as a *single base*. We further distinguish between two connection types of bases in R: the connection between a base i and its subsequent base $i + 1$ is denoted as a *backbone connection*, and a base pair connection is denoted as a *bond connection*. The *span* of a base pair $bp = (a, b, p) \in B$ is the number of bases that it contains. i.e., $|bp| = (b - a + 1)$. We assume that the number of base pairs in R is $O(n)$, which holds for nested and bounded-unlimited structures by definition. For simplicity, we assume that R contains a base pair between positions 1 and n, and add such base pair if it does not exist.

Definition 1 (Parent-child relation between bases). *A parent of base pair $bp = (a, b, p) \in B$ (resp. single base i) is the smallest span base pair $pbp = (c, d, q) \in B$ that contains bp (resp. i) in it. That is, c, d are the closest endpoints of a base pair such that $c < a < b < d$ (resp. $c < i < d$). We denote bp (resp. i) as the* child *of pbp.*

Note that in *Nested* structures, every base (or base pair) has a unique parent base pair, except for $bp = (1, n, p)$. In more complex structures, a base (or base pair) can have several parent base pairs. In this paper, we are interested in the parent-child relation between bases (and base pairs) in *Nested* structures.

We proceed with definitions of substructures of R (see Fig. 3 for examples):

Definition 2 (Path). *A path in RNA R is a sequence of unique positions $(i_1, \ldots, i_y)$ such that $\forall 1 \le k < y$, i_k is connected to i_{k+1} with either a backbone or a bond connection. If i_k is connected to i_{k+1} with a bond connection we say that the base pair $bp = (i_k, i_{k+1}, p)$ is contained in the path.*

Definition 3 (Pattern). *A pattern in RNA R is a subset of R positions $P = \{i_1, \ldots, i_y\}$ such that $y \le n$ and $\forall 1 \le k < l \le y$ there exists a path in P that connects i_k and i_l.*

Definition 4 (Exact Pattern Matching). *Given two RNAs $R_1 = (S_1, B_1)$ and $R_2 = (S_2, B_2)$, with spans n and m respectively, an* exact pattern matching *(in short, matching) M, over R_1 and R_2 is a set of pairs $M = \{(i_1, j_1), \ldots, (i_k, j_k)| \forall 1 \le \ell \le k, 1 \le i_\ell \le n, 1 \le j_\ell \le m\}$ that satisfies the following conditions:*

1. $S_1(i_\ell) = S_2(j_\ell)\ \forall 1 \le \ell \le k$.
2. $P_1 = \{i_1, \ldots, i_k\}$ is a pattern in R_1.
3. $P_2 = \{j_1, \ldots, j_k\}$ is a pattern in R_2.
4. For each $1 \le x, y \le k$, a base pair $bp_1 = (i_x, i_y, p)$ is contained in P_1 if and only if a base pair $bp_2 = (j_x, j_y, q)$ is contained in P_2.
5. M is maximally extended.

The first condition applies to the sequence equivalence requirement, whereas the rest of the conditions apply to the structural equivalence requirement. The last condition refers to the maximality of the matching, meaning that it cannot be extended sequence- or structure- wise. For two base pairs in the matching, $bp_1 = (a, b, p) \in B_1$ and $bp_2 = (c, d, q) \in B_2$, we say that $(bp_1, bp_2) \in M$.

Each matching M has an associated *score* that can be described as:

$$score(M) = \sum_{(i,j)\in M} \alpha(i, j) + \sum_{(bp_1,bp_2)\in M} \beta(bp_1, bp_2),$$

where $\alpha : [1, |\Sigma|] \times [1, |\Sigma|] \rightarrow \mathbb{R}$ returns the score of matching two single bases, and $\beta : ([1, |B_1|]) \times ([1, |B_2|]) \rightarrow \mathbb{R}$ returns the score of matching two base pairs $bp_1 = (a, b, p)$, $bp_2 = (c, d, q)$.

In our implementation, where exact matching is concerned, we set the score to $-\infty$ when the compared bases are different in order to avoid mismatches. The functions are defined as follows:

$$\alpha(i, j) = 1 \text{ if } S_1(i) = S_2(j) \text{ or } -\infty \text{ otherwise},$$

$$\beta(bp_1, bp_2) = ((1 + p) \cdot (1 + q)) \text{ if } S_1(a) = S_2(c) \text{ and } S_1(b) = S_2(d), \text{ or } -\infty, \text{otherwise}.$$

The definition of the scoring functions enables finding biologically meaningful structures via the scoring. In the general case the scoring functions can be defined to return scores other than 1 or $(1 + p) \times (1 + q)$ when the bases match. The optimal sequence-structure matching depends on both the matching of single bases and base pairs. This enables us to sometimes prefer a matching of a base pair with a high probability over matching a single base, or prefer matching large sequence of single bases over low probability base pair (see Fig. 4).

2.1 Compensatory Mutations

In order to consider compensatory mutations instead of exact matching, one can modify $\beta(bp_1, bp_2)$ function definition: instead of scoring two base pairs with different endpoints with $-\infty$, the score can be given depending on the endpoints bases. This way, two base pairs with different endpoints get higher score in the matching than the score of matching their endpoints as single bases.

2.2 Local Exact Pattern Matching Problem Definition

Given two RNAs, $R_1 = (S_1, B_1)$ and $R_2 = (S_2, B_2)$ with spans n and m, resp. ($n \geq m$), scoring functions $\alpha()$ and $\beta()$, and a number c, we want to find the set $\mathcal{M}$ containing all matchings with a score greater than c. i.e,

$$\mathcal{M} = \{M | M \ is \ a \ matching \ and \ score(M) \geq c)\}.$$

Note that the definition of the problem does not restrict the structure of the given RNA sequences. In addition, observe that since the matching set $\mathcal{M}$ contains only common base pairs between R_1 and R_2, the structure of the input RNAs actually defines the structure of $\mathcal{M}$. We will explore two different settings of RNA structures: $(Nested, Nested)$ and $(Nested, Bounded - Unlimited)$. Hence, the output matching set structure for both settings is $Nested$.

3 A Simple $O(n^4)$ Algorithm for Local Exact Pattern Matching

In this section we solve the local exact pattern matching problem following its definition in Section 2.2. We use similar ideas to those in Zhang and Shasha's tree edit distance algorithm [24], and use the edit operations presented in [10]. The algorithm distinguishes between two cases of matchings: those that don't contain any base pair matching and those that contain at least one. In the first case, no base pair from B_1 is matched with a base pair from B_2. The problem is, therefore, finding common substrings using suffix trees in time and space $O(n + m)$ [16]. The second case is the more interesting one, and we will explore its implementation in the following sections. The key idea is that we find the matchings between each combination of a *base pair* from B_1 and a *base pair* from B_2. For convenience reasons, we refer to arc-annotated substrings as *substrings*.

3.1 Finding the Maximal Matching between Two Base Pairs

The algorithm divides the process of finding the matching into two stages: finding the maximal matching in between the two endpoints of both base pairs (discussed in Section 3.2), and extending the match "outside" of the base pairs (discussed in Section 3.3). On each of these stages, the maximal score is saved in table M, of size $O(|B_1||B_2|)$, in which an entry M_{bp_1,bp_2} contains the scores of comparing the two base pairs $bp_1 \in B_1$ and $bp_2 \in B_2$: inside the base pairs, their maximal extensions and the total score. We denote these scores as $M^{in}_{bp_1,bp_2}$, $M^{out}_{bp_1,bp_2}$, and $M^{total}_{bp_1,bp_2}$ respectively.

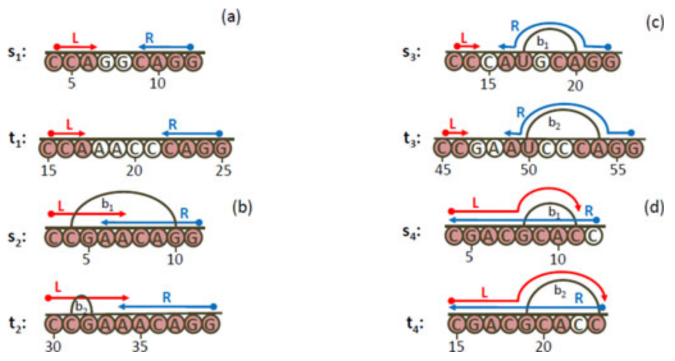


Fig. 5. $Lmatch$, $Rmatch$, $Full$ and $Score$ matchings between substrings s_i and t_i: $Lmatch$ is marked with 'L' right arrows, and $Rmatch$ is marked with 'R' left arrows. The probabilities of b_1 and b_2 to exist in s_i and t_i are 0.2 and 0.2, respectively. (a) Non-overlapping: $Lmatch(s_1, t_1)$ contains 'CCA' and $Rmatch(s_1, t_1)$ contains 'GGAC', $Full = -\infty$ and $Score(s_1, t_1) = 7$ (both 'CCA' and 'GGAC') . (b) Overlapping: $Lmatch(s_2, t_2)$ contains positions (3, 30), (4, 31), (5, 32), (6, 33) and (7, 34), $Rmatch(s_2, t_2)$ contains positions (6, 34), (7, 35), (8, 36), (9, 37), (10, 38) and (11, 39). Note that using both (7, 34) and (7, 35) (or both (6, 33) and (6, 34)) would have created an overlapping matching. Therefore, $Score(s_2, t_2) = 9$ (all single bases of s_2 and t_2 excluding base 35), and $Full(s_2, t_2) = -\infty$ since there is no matching that contains both (3, 30) and (11,39). Note that in this case, the maximal score is the one that uses arc breaking operation: $Score(s_2, t_2)$ does not include "jumping over" b_1 and b_2. (c) "Jumping over" base pairs: $Lmatch(s_3, t_3)$ contains 'CC', $Rmatch(s_3, t_3)$ contains 'GG', (b_1, b_2), and 'A'. Note that since b_1 and b_2 are contained in $Rmatch(s_3, t_3)$, the matching bases inside of them ('AC' and 'U') are also contained in $Rmatch(s_3, t_3)$. Also note that the matching between b_1 and b_2 is a $Full$ matching: the matching bases $\{(17, 50), (19, 53), (20, 54)\}$ contains both endpoints of b_1 and b_2. $Full(s_3, t_3) = -\infty$ and $Score(s_3, t_3) = 9.43$ (contains 'CC' from left and 'GG', (b_1, b_2), and 'A' from right). (d) $Full < Lmatch$: $Full(s_4, t_4) = 9$ by matching all bases of both s_4 and t_4 and arc-breaking b_1 and b_2. $Lmatch(s_4, t_4) = 9.44$ by "jumping over" the base pairs b_1 and b_2, $Rmatch(s_4, t_4) = 9$. Hence, $Score(s_4, t_4) = Lmatch(s_4, t_4)$.

3.2 Finding the Maximal Score Matching Inside the Base Pairs

The input of the algorithm is two RNAs $R_1 = (S_1, B_1)$ and $R_2 = (S_2, B_2)$ and the output is M^{in} table, in which an entry $M^{in}_{bp_1,bp_2}$ contains the maximal matching score between the base pairs $bp_1 \in B_1$ and $bp_2 \in B_2$ and their inner parts. The values of M^{in} table are computed in increasing order of the base pairs' spans in order to enable reuse of calculations: if two base pairs are contained in two other base pairs, then the calculation of the smaller base pairs' maximal matching is already calculated and there is no need to recalculate it (see Fig. 5 case (c) for an example).

The main procedure of the algorithm computes for every combination of a base pair $bp_1 = (a, b, p) \in B_1$ and a base pair $bp_2 = (c, d, q) \in B_2$, their maximal matching score by comparing the two substrings $s = (s_a, \dots, s_b)$ and $t = (t_c, \dots, t_d)$ that are defined over bp_1 and bp_2, respectively. It is a dynamic programming algorithm that computes matchings between prefixes of the substrings s and t, in increasing order of their sizes.

We next describe the $patternMatch()$ function that computes the maximal matching score between two substring s and t.

The pattern matching function. For every two substrings $s = (s_a, \dots, s_i)$ and $t = (t_c, \dots, t_j)$ the function computes four different matchings:

- *Lmatch*: The maximal left-to-right matching that starts at positions (a, c) and continues going from left to right using a backbone or bond connections until either a mismatch occurs or the rightmost bases of s or t are reached.
- *Rmatch*: The maximal right-to-left matching that starts at (i, j) and continues going from right to left until either a mismatch occurs or the leftmost bases of s or t are reached.
- *Full*: The maximal matching that contains both (a, c) and (i, j) indices, if such matching exists.
- *Score*: The maximal left to right and right to left matchings between the two substrings, such that they do not overlap and are maximally extended.

Note that the maximal matching score does not necessarily include both *Rmatch* and *Lmatch*, since the bases they contain may overlap. Another observation is that the score of a *Full* matching may be smaller than *Score* (see Fig. 5 for examples).

We use $Score(a \dots i, c \dots j)$ to refer to the *Score* between substrings $s = (s_a, \dots, s_i)$ and $t = (t_c, \dots, t_j)$. We refer to *Lmatch*, *Rmatch* and *Full* properties in a similar way. The initialization values of invalid substrings (either $i = a - 1$ or $j = c - 1$) are set to 0. The values are computed according to the following equations for every $i \geq a$ and $j \geq c$ indices (in the same order):

$$Full(a \dots i, c \dots j) = \max \begin{cases} Full(a \dots i-1, c \dots j-1) + \alpha(i,j) \\ Full(a \dots e-1, c \dots f-1) + M^{in}_{b_1,b_2}, \end{cases} \quad (1)$$

$$Lmatch(a \dots i, c \dots j) = \max \begin{cases} Lmatch(a \dots i-1, c \dots j) \\ Lmatch(a \dots i, c \dots j-1) \\ Full(a \dots i, c \dots j), \end{cases} \quad (2)$$

$$Rmatch(a \dots i, c \dots j) = \max \begin{cases} Rmatch(a \dots i-1, c \dots j-1) + \alpha(i,j) \\ Rmatch(a \dots e-1, c \dots f-1) + M^{in}_{b_1,b_2} \\ 0, \end{cases} \quad (3)$$

$$Score(a \dots i, c \dots j) = \max \begin{cases} Lmatch(a \dots i, c \dots j) \\ Score(a \dots i-1, c \dots j-1) + \alpha(i,j) \\ Score(a \dots e-1, c \dots f-1) + M^{in}_{b_1,b_2}, \end{cases} \quad (4)$$

where $b_1 = (e, i, r) \in B_1$ and $b_2 = (f, j, w) \in B_2$ (if such base pairs do not exist the value of $M^{in}_{b_1,b_2}$ is $-\infty$).

Finally, the score of $M^{in}_{bp_1,bp_2}$, for $bp_1 = (a, b, p) \in B_1$ and $bp_2 = (c, d, q) \in B_2$ is set as follows: $M^{in}_{bp_1,bp_2} = \beta(bp_1, bp_2) + Score(a \dots b, c \dots d)$.

The computation of *Full* values is straight-forward: either the matching is extended to include the rightmost bases, or it is extended to include the rightmost base pairs and their inner parts. If the matching cannot be extended, the value is set to $-\infty$. *Lmatch* value is the maximum between previously computed *Lmatch* scores and the current computed *Full* value. *Rmatch* contains the maximal score that includes i, j, therefore, if the bases mismatch, it is set to 0. Otherwise, it is the maximum between extending the matching with the rightmost bases or base pairs. The value of *Score* is the maximum between extending the maximal score with either single base or base pairs matching, or the maximal left to right matching, *Lmatch*, that was computed between the substrings. The reason for that is that each one of the allowed operations can set *Rmatch* score to 0. *Lmatch*, on the other hand, cannot be decreased and it can only be increased to contain the *Full* matching score (if it is bigger).

Note that in any of the computations the structure of the rightmost bases is not checked, which can lead to *arc-breaking* - the case when a base pair is treated as two single bases with no bond connection between them.

The value of *Rmatch* is not used for the total score in this algorithm, but in the improved algorithm it will be used and for clarity we define it here.

Time Complexity: patternMatch() function computes the matching scores between all *prefixes* of the substrings s and t. Computing the entries of *Full*, *Lmatch*, *Rmatch*, and *Score* is done in constant time, since it is the maximum over a constant number of expressions. These expressions are either scores of matching between prefixes of s and t or scores of matching base pairs that are contained in s and t. Therefore, computing the main procedure in increasing order of the base pairs' spans and comparing the prefixes in increasing order of their sizes yield constant time work in each of the expressions. We therefore count the number of substrings that are being compared as part of the algorithm. There are at most $O(n)$ base pairs in R_1 and at most $O(n)$ base pairs in R_2, therefore, there are at most $O(n^2)$ base pair comparisons. Each base pair can have at most $O(n)$ prefixes (depending on its span), which gives a rough upper bound of $O(n^2)$ for each two base pairs comparison. This intuitive analysis gives an upper bound of $O(n^4)$-time for the entire algorithm.

A more careful analysis of the number of compared substrings is the following analysis: for a base $i \in R$ define the *depth* of i as the number of base pairs that contain i in them. For example, $depth(i) = |\{(x, y, p) \in B | x \leq i \leq y\}|$. Define *MaxDepth* as the maximal depth over all bases i in R. For a base pair $bp \in B$, and for a base i that is contained in bp, there exists one prefix of bp such that i is its rightmost base. Therefore, a base i is the rightmost base of $depth(i)$ substrings. The total number of substrings in R is bounded by $|R| \times MaxDepth = O(n^2)$. We get that the number of compared substrings in both RNA molecules is bounded by $|R_1| \times MaxDepth_1 \times |R_2| \times MaxDepth_2 = O(n^4)$. It immediately follows that the number of recursive calls is bounded by $O(n^4)$.

3.3 Extending the Match Outside the Base Pairs

This section describes the algorithm for computing the maximal extension of the matching outside the endpoints of base pairs. The input of the algorithm is two RNAs, $R_1 = (S_1, B_1)$ and $R_2 = (S_2, B_2)$, and the table M^{in}. The output is the M^{out} table. Each base pairs comparison can be extended to both left and right, in this section we describe the algorithm for the extension to the right; the extension to the left is similar.

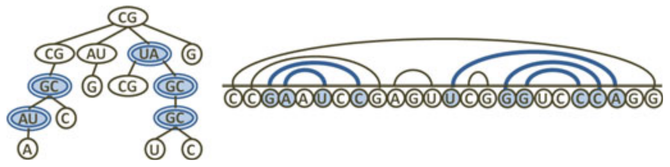


Fig. 6. Heavy path decomposition: in this RNA structure, we have three heavy path routes. They are presented in both tree and arc-annotated structures.

The algorithm computes the maximal extensions scores for every position $i \in R_1$ and $j \in R_2$, in decreasing order of i and j. The values are kept in $Rextend$ table (of size $O(n^2)$), in which an entry $Rextend(i, j)$ contains the maximal extension starting at positions i, j going right. If a mismatch occurs between s_i and t_j, the value is set to 0. Otherwise, the value is the maximum between matching single bases and matching base pairs, as follows:

$$Rextend(i,j) = \max\begin{cases} Rextend(i+1, j+1) + \alpha(i,j) \\ Rextend(b+1, d+1) + M^{in}_{b_1,b_2} \\ 0, \end{cases} \quad (5)$$

where $b_1 = (i, b, r) \in B_1$ and $b_2 = (j, d, w) \in B_2$, and $Rextend(n+1, j) = Rextend(i, m+1) = 0$.

Eventually, for every two base pairs, $bp_1 = (a, b, p) \in B_1$ and $bp_2 = (c, d, q) \in B_2$, the values in $M^{out}_{bp_1,bp_2}$ table are set as follows:

$$M^{out}_{bp_1,bp_2} = Rextend(b+1, d+1) + Lextend(a-1, c-1).$$

Time Complexity: computing each entry of tables $Rextend$ and $Lextend$ takes $O(1)$ time, since it is the maximum over three expressions. This gives a total of $O(n^2)$ time for computing the tables. The calculation of table M^{out} for two base pairs $bp_1 \in B_1$ and $bp_2 \in B_2$ is done in constant time, which gives a total time of $O(n^2)$ for all combinations of bp_1 and bp_2. Therefore, the time complexity of the algorithm is $O(n^2)$.

3.4 Complete $\mathbf{O(n^4)}$ Algorithm

The algorithm for computing the local exact pattern matching between two given RNA molecules is as follows:

1. Compute the pattern matching inside all base pairs into M^{in}.
2. Compute the extension tables $Rextend$ and $Lextend$ and the table M^{out} accordingly.
3. For each base pair $bp_1 \in B_1$ and each base pair $bp_2 \in B_2$: $M^{total}_{bp_1,bp_2} = M^{in}_{bp_1,bp_2} + M^{out}_{bp_1,bp_2}$.

Time Complexity: the time complexity of step (a) is equal to $O(n^4)$, as described in Section 3.2. In step (b), the computation of tables $Rextend$, $Lextend$, and M^{out} tables is done in $O(n^2)$ time (see Section 3.3). Finally, the last step runs in $O(n^2)$ time: for each combination of a base pair from B_1 and a base pair from B_2, the computation of $M^{total}_{bp_1,bp_2}$ entry is done in constant time.

Therefore, the time complexity of the complete algorithm is $O(n^4 + n^2 + n^2) = O(n^4)$. The space complexity of the algorithm is bounded by $O(n^2)$, as the M^{in} and M^{out} tables size is $O(n^2)$. In addition, the size of tables $Score, Full, Lmatch$ and $Rmatch$ is $O(n^2)$ per each two base pairs comparison. The tables $Lextend$ and $Rextend$ size is also bounded by $O(n^2)$.

From this time complexity analysis we immediately observe that the bottleneck of the algorithm is computing the maximal matching score inside the base pairs. In the next Sections 4 and 5 we show how to improve this time complexity.

4 An $\mathbf{O(n^3 \log n)}$ Algorithm for Local Exact Pattern Matching

In this algorithm we use similar ideas of Klein's tree edit distance algorithm [14]. We first explain the heavy path decomposition concept in regarding RNAs and continue with the modifications to the $O(n^4)$ algorithm.

Definition 5 (heavy-light base pairs). *For a given RNA $R = (S, B)$, we define each base pair in B as heavy or light by the following recursive definition: the base pair $bp_1 = (1, |R|, p)$ is defined light (if such base pair does not exist, we add it as a fictive base pair). For each base pair $bp \in B$, we pick a child base pair of bp with maximal span among the children of bp (breaking ties arbitrarily) and mark it as heavy, the rest of the children are marked as light. We say that $heavy(bp) = hp$ if hp is the heavy child base pair of bp.*

The sequence of $bp_1, heavy(bp_1), heavy(heavy(bp_1)), \ldots$ defines a descending path called the *heavy path*, let $P(bp_1)$ denote this path. We recursively decompose R into heavy paths: we start with $P(bp_1)$ and add the heavy path of each light child base pair of bp_1 (see Fig. 6). We denote each light base pair as the *root* of the heavy path that it contains.

The following Lemma of Sleator and Tarjan [22] bounds the number of light base pairs that contain a base in R:

Lemma 1 (Sleator and Tarjan [22]). *Each base in RNA $R = (S, B)$, of size n, is contained in at most $O(\log n)$ light base pairs.*

Definition 6 (Special Substrings). *The set of special substrings of an arc annotated substring $s = (s_a, \ldots, s_b)$, that is defined over a base pair $bp = (a, b, p) \in B$ with $heavy(bp) = (x, y, r) \in B$, consists of the suffixes of $(s_a, \ldots, s_y)$ starting at positions $a, \ldots, x$, and the prefixes of $(s_a, \ldots, s_b)$ ending at positions $y, \ldots, b$ (see Fig. 7).*

We denote the special substrings that are prefixes of $(s_a, \ldots, s_b)$ as *prefix special substrings*, and the suffixes of $(s_a, \ldots, s_y)$ as *suffix special substrings*. Let s be a substring. We denote *last(s)* as either the rightmost or the leftmost base of s. We define $last(s)$ of a suffix special substring, s, to be the leftmost base in s, and $last(s)$ of a prefix special substring s to be its rightmost base. Each base i in $(a, \ldots, b)$ that is not contained in the heavy child base pair of bp, hp, defines exactly one special substring that contains i as its *last* base. Thus, the number of special substrings defined over a base pair is: $span(bp) - span(hp)$.

Let $bp_1 = (a, b, p) \in B_1$ with $heavy(bp_1) = hp = (x, y, r) \in B_1$, $bp_2 = (c, d, q) \in B_2$, and let $s = (s_a, \ldots, s_b)$, $h = (s_x, \ldots, s_y)$ and $t = (t_c, \ldots, t_d)$ be the substrings defined over bp_1, hp and bp_2, respectively.

The algorithm is based on two changes to the $O(n^4)$ algorithm: the first modification is in the compared substrings: we compare *all* substrings of t and only the special

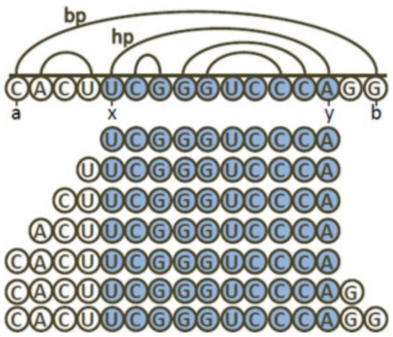


Fig. 7. Special substrings example: the special substrings of a base pair, $bp = (a, b, p)$, with a heavy child base pair, $hp = (x, y, r)$.

substrings of s as part of the $patternMatch()$ function. The special substrings are compared in increasing order of their sizes: we start with the heavy child base pair's substring, h, and increase the substring from left, until the left endpoint of bp_1 is reached (the suffixes special substrings). Then, we continue with the prefixes of bp_1, starting from $s_a, \ldots, s_y$, and continue going from left to right until the right endpoint of bp_1 is reached. Using this specific order of comparisons, we are able to use previously computed values (of the comparison between the heavy base pair substring, h, and all substrings of t) in a more efficient way.

The second modification is in the main procedure of $patternMatch()$: in the previous algorithm, $last(s)$ was always the rightmost base, in this version it is sometimes the leftmost base. Thus, the function should support ignoring or matching of both $last(s)$ positions. The function is therefore the combination of two $patternMatch()$ versions: for the prefix comparisons the computation is exactly as described in Section 3.2. Using the previous notations of bp_1, hp and bp_2, the suffix comparisons are computed for every $a \leq i < x$ and $c \leq j \leq d$ according to the following equations:

$$Full(i \ldots y, j \ldots d) = \max\begin{cases} Full(i+1 \ldots y, j+1 \ldots d) + \alpha(i,j) \\ Full(e+1 \ldots y, f+1 \ldots d) + M^{in}_{b_1,b_2}, \end{cases} \tag{6}$$

$$Lmatch(i \ldots y, j \ldots d) = \max\begin{cases} Lmatch(i+1 \ldots y, j+1 \ldots d) + \alpha(i,j) \\ Lmatch(e+1 \ldots y, f+1 \ldots d) + M^{in}_{b_1,b_2} \\ 0, \end{cases} \tag{7}$$

$$Rmatch(i \ldots y, j \ldots d) = \max\begin{cases} Rmatch(i+1 \ldots y, j \ldots d) \\ Rmatch(i \ldots y, j+1 \ldots d) \\ Full(i \ldots y, j \ldots d), \end{cases} \tag{8}$$

$$Score(i \ldots y, j \ldots d) = \max\begin{cases} Rmatch(i \ldots y, j \ldots d) \\ Score(i+1 \ldots y, j+1 \ldots d) + \alpha(i,j) \\ Score(e+1 \ldots y, f+1 \ldots d) + M^{in}_{b_1,b_2}, \end{cases} \tag{9}$$

where $b_1 = (i, e, r) \in B_1$ and $b_2 = (j, f, w) \in B_2$, and the initialization values of invalid substrings ($j = n + 1$) are set to 0.

Eventually, the value in M^{in} table is set for $bp_1 = (a, b, p) \in B_1$ and $bp_2 = (c, d, q) \in B_2$ to: $M^{in}_{bp_1,bp_2} = \beta(bp_1, bp_2) + Score(a \ldots b, c \ldots d)$.

Note that the above equations are symmetrical to the equations of the prefixes comparison defined in Section 3.2. The values of $Full, Rmatch$ and $Lmatch$ are computed according to previously computed suffixes scores and smaller base pairs matching scores. The value of $Score$ is similarly the maximum between $Rmatch$ and previously computed $Score$ values.

Time Complexity: the same reasons that the $O(n^4)$ algorithm gave constant time for each $patternMatch(s, t)$ function call apply here, too. We therefore count the number of compared substrings: following Lemma 1, each base is defined as $last(s)$ of at most $O(\log n)$ special substrings, which gives a total of $O(n \log n)$ special substrings. The set of substrings t, are all $O(n^2)$ substrings of R_2. The number of compared substrings is therefore $O(n \log n \times n^2) = O(n^3 \log n)$.

Thus, the time complexity of the above algorithm for computing the matching inside each combination of a base pair from B_1 and a base pair from B_2 is $O(n^3 \log n)$. The space complexity of the algorithm is bounded by $O(n^2)$, as the space complexity of [14] (see also [4]).

5 An $O(n^3)$ Algorithm for Local Exact Pattern Matching

In the previous algorithm (Section 4) we select the larger RNA structure as the *dominant* structure. w.l.o.g. we defined R_1 to be the dominant structure, and for each $bp_1 \in B_1$, bp_1 was the dominant base pair, by which special substrings were defined.

An improvement for this algorithm can be done using the optimal decomposition algorithm described in [7]. The key observation is that the dominant structure can be decided for each combination of base pairs comparison rather than once for the entire algorithm. The complete description and proof of the algorithm are given in [7]. In this section we give the highlights of the algorithm and "translate" it into the arc-annotated representation of RNA molecules.

As an initialization step of the algorithm, *both* R_1 and R_2 are recursively decomposed into heavy paths (see Fig. 8). The algorithm computes the matching between each combination of a base pair $bp_1 \in B_1$ and a base pair $bp_2 \in B_2$. The difference is that on each such comparison, the algorithm selects the dominant base pair to be the one with the larger root (i.e., $|root(bp_1)|$ and $|root(bp_2)|$). The rest of the algorithm is exactly the same as the previous $O(n^3 \log n)$ algorithm, meaning that the special substrings of the dominant base pair are compared with all substrings of the other base pair (see Fig. 8 for an example).

This enhancement to the algorithm improves the time complexity to $O(nm^2 \log(n/m))$, which is bounded by $O(n^3)$. The intuition behind this improvement is that on each comparison between two base pairs, we compare all substrings of the *relatively smaller* base pair with the special substrings of the *relatively larger* base pair (see

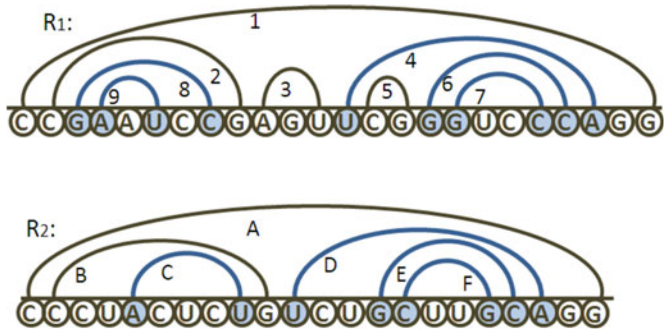


Fig. 8. Heavy path decomposition of RNA molecules: R_1 contains the heavy path $(1, 4, 6, 7, U)$. In addition R_1 contains the heavy paths $(2, 8, 9, A)$, $(3, G)$, and 5. In the comparison between $6 \in B_1$ and $E \in B_2$ the dominant base pair is 6, whereas in the comparison between $8 \in B_1$ (or $2 \in B_1$) and $B \in B_2$ the dominant base pair is B.

complete proof in [7]). The space complexity is bounded by $O(n^2)$, as explained in [7].

6 Local Exact Pattern Matching for (Nested, Bounded-Unlimited) Inputs

The input to this algorithm consists of two RNA structures $R_1 = (S_1, B_1)$ and $R_2 = (S_2, B_2)$, where R_1 is a nested structure and R_2 is a bounded-unlimited structure. The output is the maximal local exact matching set M defined over R_1 and R_2.

The algorithm is similar to the $O(n^3 \log n)$ algorithm described in Section 4. The difference is that the bounded-unlimited structure of R_2 needs to be handled: as opposed to the previous algorithm, where each base can be connected by a bond connection to at most one other base, in the bounded-unlimited structure it can be connected to $O(1)$ other bases. Let i be $last(s)$ of substring s, and let the $last(s)$ be the rightmost base in s, w.l.o.g. If i is a right endpoint of a base pair $bp_1 = (e, i, p) \in R_1$, there can be several base pairs in R_2 with j being their right endpoint (e.g., $bp_k = (f_k, j, q_k) \in R_2$). All of these base pairs should be considered in the matching between s and t (see Fig. 9 for examples).

Note that even though R_2 has a bounded-unlimited structure, the output matching structure is always nested. Hence the only modification that is necessary is to iterate over all base pairs with right endpoint j and pick the one that gives the maximal total score.

In an analogous way, the algorithm for extending the matching outside of the base pairs, as described in Section 3.3, is also modified to support the bounded-unlimited structure of R_2. Again, on each base pairs comparison the algorithm compares at most $O(1)$ options of base pairs matching.

Time Complexity: the only modification to $patternMatch()$ function is that we compare $O(1)$ base pairs of substring t with the base pair that starts at $last(s)$, if such exist. This, of course, does not add to the overall time complexity analysis. In a similar way, the modification to the algorithm for computing the maximal extensions does not change its time complexity.

The total time complexity of the entire algorithm is therefore $O(n^3 \log n)$.

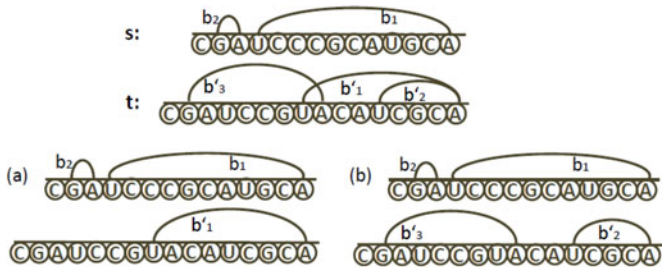


Fig. 9. Bounded-Unlimited base pairs matching: the matching between s and t may contain b_1 and b'_1 base pairs (case (a)), or b_1 and b'_2 base pairs (case (b)). Note that in case (a) the crossing base pair b'_3 is not valid, thus not considered, whereas in case (b) it might be a part of the matching.

7 Local Approximate Pattern Matching for (Nested, Nested) Inputs

In this section we solve the problem of local approximate pattern matching. The problem is defined as follows:

Given two RNAs, $R_1 = (S_1, B_1)$ and $R_2 = (S_2, B_2)$ with sizes n and m, resp. $(n \geq m)$, scoring functions $\alpha()$ and $\beta()$, two numbers k and c, we want to find the set $\mathcal{M}$ containing all matchings with a score greater than c that have at most k mismatches. i.e,

$$\mathcal{M} = \{M | M \textit{ is a matching with at most } k \textit{ mismatches and } score(M) \geq c)\}.$$

Formally, a mismatch is any $(i, j) \in M$ such that $s_i \neq t_j$. Note that, as in the previous algorithms, the operation of arc-breaking is allowed, and furthermore, it is not calculated as a mismatch.

Again, we find the approximate matchings between each combination of a *base pair* from B_1 and a *base pair* from B_2, in increasing order of the base pairs' spans. The algorithm is divided into two stages: calculating the approximate match inside the base pairs (in Section 7.1) and extending it outside of them (in Section 7.2). The algorithms run in $O(n^4k^2)$ and $O(n^2k^2)$, respectively. In Section 7.3, we explain how to modify the algorithm to an $O(n^3k^2)$ algorithm, and in Section 7.4, we show how to modify the algorithm to find local approximate pattern matching between $(Nested, Bounded-Unlimited)$ RNAs in $O(n^3k^2 \log n)$ time.

7.1 An $O(n^4k^2)$ Algorithm for Finding the Maximal Approximate Matching Score Inside the Base Pairs

The input of the algorithm consists of two RNAs $R_1 = (S_1, B_1)$ and $R_2 = (S_2, B_2)$, and a number k. The output is M^{in} table, in which an entry $M^{in}_{bp_1, bp_2}(\ell)$ contains the maximal matching score between the base pairs $bp_1 \in B_1$ and $bp_2 \in B_2$ and their inner parts having at most ℓ mismatches $(0 \leq \ell \leq k)$.

In a similar way to the algorithm for exact matching (described in Section 3.2), the main procedure of the algorithm computes for every combination of a base pair $bp_1 = (a, b, p) \in B_1$ and a base pair $bp_2 = (c, d, q) \in B_2$, their maximal approximate matching score by comparing the two substrings $s = (s_a, \ldots, s_b)$ and $t = (t_c, \ldots, t_d)$ that are defined over bp_1 and bp_2, respectively. It is a dynamic programming algorithm that computes matchings between prefixes of the substrings s and t, in increasing order of their sizes.

For every two substrings $s = (s_a, \ldots, s_i)$ and $t = (t_c, \ldots, t_j)$ the algorithm computes the *Full*, *Lmatch*, *Rmatch*, and *Score* matchings for all of the allowed mismatches ℓ, $0 \le \ell \le k$.

Note that when comparing mismatching single bases, the current score of the matching is actually the score of a previous matching with $\ell - 1$ mismatches. When comparing base pairs, there are ℓ options to split the mismatches between the previous score of the prefixes until the base pairs and the mismatches inside of them. For all ℓ $(0 \le \ell \le k)$ in total there are $O(k^2)$ splits that need to be considered.

We use $Score(a \ldots i, c \ldots j, \ell)$ to refer to the *Score* between substrings $s = (s_a, \ldots, s_i)$ and $t = (t_c, \ldots, t_j)$ with at most ℓ mismatches. We refer to *Lmatch*, *Rmatch* and *Full* properties in a similar way. As an initialization step we set $Full(i, j, -1) = Rmatch(i, j, -1) = Score(i, j, -1) = -\infty$. The values are computed according to the following equations (in the same order):

$$Full(a \ldots i, c \ldots j, \ell) = \max \begin{cases} Full(a \ldots i-1, c \ldots j-1, \ell) + \alpha(i,j) \\ Full(a \ldots i-1, c \ldots j-1, \ell-1) \\ Full(a \ldots e-1, c \ldots f-1, mis) \\ \quad + M^{in}_{b_1,b_2}(\ell - mis) \quad \forall 0 \le mis \le \ell, \end{cases} \tag{10}$$

$$Lmatch(a \ldots i, c \ldots j, \ell) = \max \begin{cases} Lmatch(a \ldots i-1, c \ldots j, \ell) \\ Lmatch(a \ldots i, c \ldots j-1, \ell) \\ \quad Full(a \ldots i, c \ldots j, \ell), \end{cases} \tag{11}$$

$$Rmatch(a \ldots i, c \ldots j, \ell) = \max \begin{cases} Rmatch(a \ldots i-1, c \ldots j-1, \ell) + \alpha(i,j) \\ Rmatch(a \ldots i-1, c \ldots j-1, \ell-1) \\ Rmatch(a \ldots e-1, c \ldots f-1, mis) \\ \quad + M^{in}_{b_1,b_2}(\ell - mis) \quad \forall 0 \le mis \le \ell \\ 0, \end{cases} \tag{12}$$

$$Score(a \ldots i, c \ldots j, \ell) = \max \begin{cases} Lmatch(a \ldots i, c \ldots j, \ell) \\ Score(a \ldots i-1, c \ldots j-1, \ell) + \alpha(i,j) \\ Score(a \ldots i-1, c \ldots j-1, \ell-1) \\ Score(a \ldots e-1, c \ldots f-1, mis) \\ \quad + M^{in}_{b_1,b_2}(\ell - mis) \quad \forall 0 \le mis \le \ell, \end{cases} \tag{13}$$

where $b_1 = (e, i, r) \in B_1$ and $b_2 = (f, j, w) \in B_2$.

Mismatches are handled as follows: for each entry computation (except for *Lmatch*) either use $\alpha(i,j)$ score (when there is a matching between i and j) or don't use it, but pay for one mismatch (thus, take the value from entry $(i-1, j-1, \ell-1)$). In addition, all values of $M^{in}_{b_1,b_2}(mis)$ are checked with the $(e-1, f-1, \ell - mis)$ values.

The value of $\beta(bp_1, bp_2)$ may also be $-\infty$ if there is a mismatch between the base pairs endpoints. Therefore, in the total score of $M^{in}_{bp_1,bp_2}(\ell)$ all such mismatches are handled:

$$M^{in}_{bp_1,bp_2}(\ell) = \max \begin{cases} Score(a \ldots b, c \ldots d, \ell) \\ \quad + \beta(bp_1, bp_2) & \text{if } s_a = t_c \text{ and } s_b = t_d \\ Score(a \ldots b, c \ldots d, \ell - 1) & \text{if } s_a = t_c \text{ or } s_b = t_d \\ Score(a \ldots b, c \ldots d, \ell - 2) & \text{if } s_a \ne t_c \text{ and } s_b \ne t_d. \end{cases} \tag{14}$$

Time Complexity: Each of the computations requires $O(1)$ time for each single base match and $O(k)$ for each base pair match. This gives a total of $O(n^2k^2)$ for computing the entire *Full*, *Lmatch*, *Rmatch*, and *Score* tables.

For each combination of a base pair from B_1 and a base pair from B_2, we compute the approximate matching inside the base pairs in $O(n^2k^2)$ time, which gives a total of $O(n^2 \times n^2k^2) = O(n^4k^2)$ time for the entire algorithm.

7.2 Extending the Approximate Match Outside the Base Pairs

The calculation of the approximate matching extensions of the base pairs works in a similar way to the algorithm described in Section 3.3. The difference is that in both the preprocessing step (where the auxiliary tables are filled) and the final step (where the table $M^{out}_{bp_1,bp_2}$ is filled) we now compute the scores for any number of mismatches separately.

In the preprocessing step we compute the maximal approximate matching extension for *every* position $i \in R_1$ and $j \in R_2$ into two auxiliary tables $Rextend$ and $Lextend$ of sizes $O(n^2k)$. We further explain the extension to right, the left extension is symmetric. On each step of the algorithm, we compute the extension to right starting from index i to $|R_1|$ and from index j to $|R_2|$. An entry $Rextend(i, j, \ell)$ denotes the maximal score that can be achieved when comparing R_1 and R_2 starting from indices i in R_1 and j in R_2, going from left to right with at most ℓ mismatches. It is computed as follows:

$$Rextend(i, j, \ell) = \max \begin{cases} Rextend(i+1, j+1, \ell) + \alpha(i,j) \\ Rextend(i+1, j+1, \ell-1) \\ Rextend(b+1, d+1, mis) \\ + M^{in}_{b_1,b_2}(\ell - mis) \quad \forall 0 \le mis \le \ell \\ 0, \end{cases} \tag{15}$$

where $b_1 = (i, b, r) \in B_1$ and $b_2 = (j, d, w) \in B_2$.

In the final step, the entries $M^{out}_{bp1,bp2}(\ell)$, for all $bp_1 = (a, b, p) \in B_1$ and $bp_2 = (c, d, q) \in B_2$ and all ℓ with $0 \le \ell \le k$ are computed. Each of these entries is obtained as the maximum of $Rextend(b+1, d+1, \ell - mis) +$ $Lextend(a-1, c-1, mis)$ over all mis such that $0 \le mis \le \ell$.

Hence, each of the $O(n^2k)$ entries requires $O(k)$ time to calculate, which gives a total time complexity of $O(n^2k^2)$ for this stage.

Analogously, we combine the entries of M^{in} and M^{out} to obtain the final result. Again, we maximize over $M^{in}_{bp_1,bp_2}(mis) + M^{out}_{bp1,bp2}(\ell - mis)$ for all mis $(0 \leq mis \leq \ell)$.

Time Complexity: since the most expensive part of the computation is the one described in Section 7.1, the total time complexity of the entire algorithm is $O(n^4k^2)$.

7.3 Improving the Algorithm to Run in $\mathbf{O(n^3k^2)}$ Time

The main idea of the improved algorithm, is the same idea that was described in Section 5: we decompose both R_1 and R_2 into heavy paths and for each combination of a base pair from B_1 and a base pair from B_2 we decide on the dominant base pair by its root in the heavy path route (see full explanation in Section 5).

In a similar way to the modification done in Section 4, the substrings being compared are either prefixes (as in the $O(n^4k^2)$ algorithm) or suffixes. Hence, the approximate pattern matching function is extended to include suffixes comparison.

Time Complexity: the total number of substrings compared in this algorithm is $O(n^3)$ (as explained in Section 5), and each substrings comparison for all k allowed mismatches takes $O(k^2)$ work (as explained in Section 7.1).

The total time complexity is therefore $O(n^3k^2)$. The total space complexity of the algorithm is $O(n^2k)$.

7.4 Local Approximate Pattern Matching for (Nested, Bounded-Unlimited) RNAs

The input of this algorithm consists of two RNA structures $R_1 = (S_1, B_1)$ and $R_2 = (S_2, B_2)$, where R_1 is a nested structure and R_2 is a bounded-unlimited structure, and a number k. The output is the maximal local approximate pattern matching that can be achieved when using at most k mismatches.

The algorithm is similar to the $O(n^3 \log n)$ algorithm described in Section 6. The difference is that now we allow mismatches: in a similar way to the algorithm presented in Section 7.1, we compute the maximal approximate matching with the minor change, that now every $last(s)$, that is an endpoint of some base pair in B_1, can be compared with a constant number of base pairs in B_2, instead of at most 1 base pair in the $(Nested, Nested)$ version.

In an analogous way, the algorithm for extending the approximate matching outside of the base pairs, as described in Section 7.2, is modified to support the bounded-unlimited structure of R_2. Again, on each base pairs comparison the algorithm compares at most $O(1)$ options of base pairs matching.

Time Complexity: the only modification to the approximate pattern matching function is that we compare $O(1)$ base pairs of substring t with the base pair that starts at $last(s)$, if such exist. This, of course, does not add to the overall time complexity analysis. In a similar way, the modification to the algorithm for computing the maximal extensions does not change its time complexity.

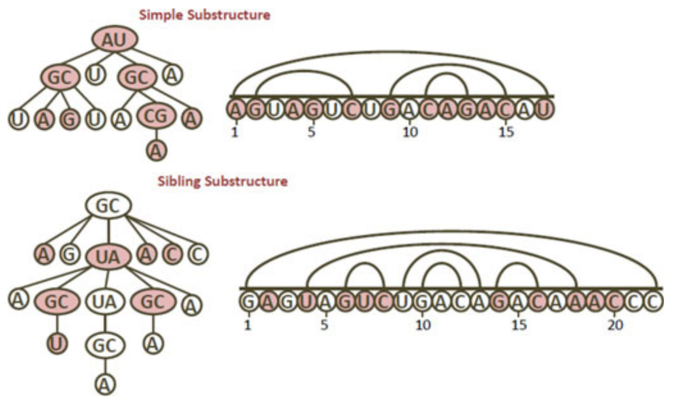


Fig. 10. Simple and sibling substructures. Examples for simple and sibling structures. Each substructure is presented as both tree and arc annotated substring. The substructures are shadowed in both representations.

The total time complexity of the entire algorithm is therefore $O(n^3k^2 \log n)$, and the total space complexity of the algorithm is $O(n^2k)$.

8 Finding the Most Similar Sibling Substructures

The most similar subforest problem was introduced in [12] as follows:

> Given an ordered labeled forest F (“the target forest”) and an ordered labeled forest G (“the pattern forest”), the most similar subforest problem is to find a subforest F′ of F such that the distance between F′ and G is minimum over all possible F′.

The definitions in [12] are presented for forests, here we translate them to arc-annotated sequence representations. Note that in this section we assume that the RNA structure is nested. We start with definitions relevant to this problem and continue with the algorithm presentation.

8.1 Definitions

Definition 7 (Simple substructure). *A simple substructure of RNA R is a set of positions $SUB_s = \langle a, \ldots, b\rangle$ such that $\forall a < i < b$ both endpoints of the parent of i are in SUB_s and (a, b, p) is a base pair in B. We denote the base pair $bp = (a, b, p) \in B$ as the* root *of the substructure. In the case where the set SUB_s contains only one single base, i.e., $SUB_s = \langle i\rangle$, we denote the root of the substructure with i.*

In the *simple* substructure with root r, not all the single bases and base pairs contained in r must be included in the substructure (see Fig. 10).

Definition 8 (Sibling substructure). *A sibling substructure of RNA R is a set of simple substructures whose* roots *are siblings.*

Note that if we add the parent of the *root* of any sibling substructures, we get a simple substructure.

The *most similar substructure* problem definition is as follows: Given two nested RNAs, $R_1 = (S_1, B_1)$ and $R_2 = (S_2, B_2)$ with sizes n and m resp. $(m < n)$, and scoring functions $\alpha()$ and $\beta()$, we want to find a sibling substructure of R_1 that has the minimum edit distance to R_2 over all possible sibling substructures of R_1.

8.2 The Algorithm

The input to the algorithm consists of two fixed RNAs, $R_1 = (S_1, B_1)$ and $R_2 = (S_2, B_2)$, and the output is the table S, in which an entry S_{bp_1,bp_2} contains the minimum edit distance between a sibling substructure of a base pair $bp_1 = (a, b) \in B_1$ and a base pair $bp_2 = (c, d) \in B_2$. This problem differs from the previously discussed problems in two main aspects: the input RNA molecules are in fixed structure (this means that *arc-breaking* operation is not allowed, thus base pairs cannot be treated as single bases), and the output is restricted to a sibling substructure of R_1 rather than patterns of both R_1 and R_2. This restriction is supported by allowing to delete single bases or base pairs from R_1 with no additional cost. Another difference is that this algorithm finds the minimal edit distance cost instead of the maximal matching, thus, the cost functions $\alpha()$ and $\beta()$ are changed to return 0 for exact matches and 1 for mismatches or deletions. In addition, the value of $\alpha(i, j) = \infty$ when i or j are not single bases. We denote the deletion of a single base j from R_2 as $\alpha(-, j)$ (with $\alpha(-, j) = 1$), and the deletion of a base pair bp_2 from R_2 as $\beta(-, bp_2)$ (with $\beta(-, bp_2) = |bp_2|$). For simplicity, we start with an $O(n^4)$ algorithm.

The main procedure of the algorithm computes the minimal edit distance between a sibling substructure of substring $s = (s_a, \ldots, s_i)$ and a substring $t = (t_c, \ldots, t_j)$ into $Edit$ auxiliary table. The value of $Edit(a \ldots i, c \ldots j)$ is computed as follows:

$$Edit(a \ldots i, c \ldots j) = \min \begin{cases} Edit(a \ldots i-1, c \ldots j) & \backslash\backslash \mathit{delete\ a\ leaf\ from}\ T_1 \\ Edit(a \ldots i, c \ldots j-1) + \alpha(-, j) & \backslash\backslash \mathit{delete\ a\ leaf\ from}\ T_2 \\ Edit(a \ldots i-1, c \ldots j-1) + \alpha(i, j) & \backslash\backslash \mathit{match\ leaves} \\ Edit(a \ldots e-1, c \ldots j) & \backslash\backslash \mathit{delete\ an\ internal\ node\ from}\ T_1 \\ Edit(a \ldots i, c \ldots f-1) + \beta(-, b_2) & \backslash\backslash \mathit{delete\ an\ internal\ node\ from}\ T_2 \\ Edit(a \ldots e-1, c \ldots f-1) + S_{b_1,b_2} & \backslash\backslash \mathit{match\ internal\ nodes}, \end{cases} \tag{16}$$

where $b_1 = (e, i) \in B_1$ and $b_2 = (f, j) \in B_2$, and the initial values $Edit(a \ldots a-1, c \ldots j)$ and $Edit(a \ldots i, c \ldots c-1)$ are set to 0.

Eventually, the value in S table is set to: $S_{bp_1,bp_2} = Edit(a \ldots b, c \ldots d)$.

Note that when a single base (or a base pair) is deleted from substring s, there is no extra penalty for the operation, whereas, when the deletion is from substring t, the function $\alpha(-, j)$ (or $\beta(-, bp_2)$) is added to the $Edit$ score. The reason is that every such deletion from s does not break the sibling substructure of s, and hence it does not increase the edit distance score. Another observation is that the last expression in the equation is the previously computed value S_{b_1,b_2} for some smaller span base pairs $b_1 = (e, i, r) \in B_1$ and $b_2 = (f, j, w) \in B_2$.

In a similar way to the improved algorithm in Section 4, we can decompose R_1 into heavy paths and compare only the special substrings of R_1 with all substrings of R_2. If we use the technique of [7], as described in Section 5, we can improve the number of substrings being compared to $O(n^3)$ (instead of $O(n^4)$ in this algorithm).

Time Complexity: The algorithm compares the base pairs in increasing order of their spans, and their substrings in increasing order of their sizes. Thus, each sub problem of the expressions in the main procedure is already calculated, which gives constant time work for each expression. Therefore, if we use the technique described in Section 5, we get a total time complexity of $O(n^3)$ for computing the most similar substructure between two nested RNAs. The space complexity is, again, bounded by $O(n^2)$.

Acknowledgments

The authors would like to thank Oren Weimann for very helpful discussions and suggestions. In addition, they would also like to thank the anonymous reviewers for helpful comments. Mika Amit was partially supported by the Israel Science Foundation grant 347/09 and DFG. Rolf Backofen, Steffen Heyne, Mathias Möhl, Christina Otto and Sebastian Will were partially supported by the German Research Foundation (BA 2168/3-1 and MO 2402/1-1). Gad M. Landau was partially supported by the US National Science Foundation Award 0904246, Israel Science Foundation grant 347/09, Grant No. 2008217 from the United States-Israel Binational Science Foundation (BSF) and DFG. A preliminary version appeared in the proceeding of CPM 2012 [1].

References

[1] M. Amit, R. Backofen, S. Heyne, G.M. Landau, M. Möhl, C. Schmiedl, and S. Will, "Local Exact Pattern Matching for Non-Fixed RNA Structures," *Proc. 23rd Ann. Symp. Combinatorial Pattern Matching*, pp. 306-320, 2012.

[2] R. Backofen, S. Chen, D. Hermelin, G.M. Landau, M.A. Roytberg, O. Weimann, and K. Zhang, "Locality and Gaps in RNA Comparison," *J. Computational Biology*, vol. 14, no. 8, pp. 1074-1087, 2007.

[3] G. Blin, M. Crochemore, and S. Vialette, *Algorithmic Aspects of ARC-Annotated Sequences*, pp. 113-127, John Wiley & Sons, 2011.

[4] P. Bille, "A Survey on Tree Edit Distance and Related Problems," *Theoretical Computer Science*, vol. 337, no. 1-3, pp. 217-239, 2005.

[5] R. Backofen, G.M. Landau, M. Möhl, D. Tsur, and O. Weimann, " Fast RNA Structure Alignment for Crossing Input Structures," *Proc. 20th Ann. Symp. Combinatorial Pattern Matching*, pp. 236-248, 2009.

[6] J. Couzin, "Small RNAs Make Big Splash," *Science*, vol. 298, no. 5602, pp. 2296-2297, 2002.

[7] E.D. Demaine, S. Mozes, B. Rossman, and O. Weimann, "An Optimal Decomposition Algorithm for Tree Edit Distance," *ACM Trans. Algorithms*, vol. 6, no. 1, pp. 2:1-2:19, 2009.

[8] S. Dulucq and H. Touzet, "Decomposition Algorithms for the Tree Edit Distance Problem," *J. Discrete Algorithms*, vol. 3, no. 2-4, pp. 448-471, 2005.

[9] P.A. Evans, "Algorithms and Complexity for Annotated Sequence Analysis," PhD thesis, Univ. of Alberta, 1999.

[10] T. Jiang, G. Lin, B. Ma, and K. Zhang, "A General Edit Distance Between RNA Structures," *J. Computational Biology*, vol. 9, no. 2, pp. 371-388, 2002.

[11] T. Jiang, G. Lin, B. Ma, and K. Zhang, "The Longest Common Subsequence Problem for Arc-Annotated Sequences," *J. Discrete Algorithms*, vol. 2, no. 2, pp. 257-270, 2004.

[12] J. Jansson and Z. Peng, "Algorithms for Finding a Most Similar Subforest," *Theory of Computing System*, vol. 48, no. 4, pp. 865-887, 2011.

[13] T. Jiang, L. Wang, and K. Zhang, "Alignment of Trees—An Alternative to Tree Edit," *Theoretical Computer Science*, vol. 143, no. 1, pp. 137-148, 1995.

[14] P.N. Klein, "Computing the Edit-Distance between Unrooted Ordered Trees," *Proc. Sixth Ann. European Symp. Algorithms*, pp. 91-102, 1998.

[15] G.H. Lin, Z.Z. Chen, T. Jiang, and J. Wen, "The Longest Common Subsequence Problem for Sequences with Nested Arc Annotations," *J. Computer and System Sciences*, vol. 65, no. 3, pp. 465-480, 2002.

[16] G.M. Landau and U. Vishkin, "Fast Parallel and Serial Approximate String Matching," *J. Algorithms*, vol. 10, no. 2, pp. 157-169, 1989.

[17] P.B. Moore, "Structural Motifs in RNA," *Ann. Rev. of Biochemistry*, vol. 68, pp. 287-300, 1999.

[18] M. Möhl, S. Will, and R. Backofen, "Lifting Prediction to Alignment of RNA Pseudoknots," *Proc. 13th Ann. Int'l Conf. Research in Computational Molecular Biology*, pp. 285-301, 2009.

[19] S. Siebert and R. Backofen, "A Dynamic Programming Approach for Finding Common Patterns in RNAs," *J. Computational Biology*, vol. 14, no. 1, pp. 33-44, 2007.

[20] S. Schirmer and R. Giegerich, "Forest Alignment with Affine Gaps and Anchors," *Proc. 22nd Ann. Conf. Combinatorial Pattern Matching*, pp. 104-117, 2011.

[21] C. Schmiedl, M. Möhl, S. Heyne, M. Amit, G.M. Landau, S. Will, and R. Backofen, "Exact Pattern Matching for RNA Structure Ensembles," *Proc. 16th Int'l Conf. Research in Computational Molecular Biology (RECOMB '12)*, pp. 245-260, 2012.

[22] D.D. Sleator and R.E. Tarjan, "A Data Structure for Dynamic Trees," *J. Computer and System Sciences*, vol. 26, no. 3, pp. 362-391, 1983.

[23] K.C. Tai, "The Tree-To-Tree Correction Problem," *J. ACM*, vol. 26, no. 3, pp. 422-433, 1979.

[24] K. Zhang and D. Shasha, "Simple Fast Algorithms for the Editing Distance between Trees and Related Problems," *SIAM J. Computing*, vol. 18, no. 6, pp. 1245-1262, 1989.

[25] K. Zhang, L. Wang, and B. Ma, "Computing Similarity between RNA Structures," *Proc. 10th Ann. Symp. Combinatorial Pattern Matching*, pp. 281-293, 1999.

Mika Amit received the MSc degree in computer science from the University of Haifa in 2012. She is working toward the PhD degree in the Department of Computer Science, University of Haifa, Israel. Her research interests include computational biology, string algorithms, and data structures.

Rolf Backofen studied computer science at the University of Erlangen, and received the PhD degree in computer science from the University of Saarland in December 1994, where he worked at the German Research Center for Artificial Intelligence. He received the habilitation from the University Munich in February 2000. He was the chair for bioinformatics at the University of Jena from November 2001 till June 2005. After declining an offer for a full professorship at the University of Linz in 2004, he became the chair for Bioinformatics at the University of Freiburg, Institute of Computer Science. He is a coauthor of the book *Computational Molecular Biology: An Introduction* (Wiley&Sons, Mathematical and Computational Biology Series, 2000). His research is concented around RNA bioinformatics, where his lab developed several leading tools for the detection and analysis of non-coding RNA, and for the investigation of RNA-based regulation.

Steffen Heyne received the master's degree in bioinformatics from the University of Jena, Germany. He is a dipl.-bioinformatician (MSc) at the University of Freiburg, Germany. His research interest include local RNA motifs and clustering of RNA sequences. While currently finishing the PhD degree, he recently joined the Max-Planck-Institute of Immunobiology and Epigenetics in Freiburg, Germany.

Gad M. Landau received the PhD degree in computer science from Tel Aviv University in 1987. His research interests include string algorithms, data structures and computational biology. He is currently a full professor in the Department of Computer Science at the University of Haifa and a research professor in the Department of Computer Science and Engineering at NYU Polytechnic School of Engineering.

Mathias Möhl received the doctoral degree in computer science from Saarland University. He has just quit his academic career and is now the founder and CEO of mamoworld. As postdodctoral researcher at Freiburg University, his research interests included RNA bioinformatics and dynamic programming.

Christina Otto received the diploma in bioinformatics from the University of Tuebingen, Germany. She is currently working toward the PhD degree in Prof. Backofen's chair for Bioinformatics in Freiburg, Germany. Her research interests include the design and implementation of efficient algorithms for RNA-related tasks and sparsification techniques.

Sebastian Will received the PhD degree in computer science from the University of Jena. He is currently a post-doctoral researcher at the University Leipzig, where he is working in the bioinformatics lab of Peter Stadler. In 2005-2012, he was appointed assistant professor ("Akademischer Rat") at the University Freiburg with Rolf Backofen. In 2010-2011, he worked as a post-doctoral researcher with Bonnie Berger at the Massachusetts Institute of Technology (MIT), Cambridge, Massachusetts, holding a research fellowship of the German Research Foundation. His main research interests are in algorithmic bioinformatics with a focus on RNA structure and comparative analysis.