Local Exact Pattern Matching for Non-Fixed RNA Structures

Mika Amit[1*], Rolf Backofen[3,4**], Steffen Heyne[3**], Gad M. Landau[1,2***], Mathias Möhl[3**], Christina Schmiedl[3**], Sebastian Will[3,5**]

[1] Department of Computer Science, University of Haifa, Mount Carmel, Haifa, Israel.
[2] Department of Computer Science and Engineering, NYU-Poly, Brooklyn NY, USA
[3] Bioinformatics, Institute of Computer Science, Albert-Ludwigs-Universität, Freiburg, Germany
[4] Center for Biological Signaling Studies (BIOSS), Albert-Ludwigs-Universität, Freiburg, Germany
[5] CSAIL and Mathematics Department, MIT, Cambridge MA, USA

Abstract. Detecting local common sequence-structure regions of RNAs is a biologically meaningful problem. By detecting such regions, biologists are able to identify functional similarity 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 which matches fixed structures, we support the *arc breaking* edit operation; this allows to match only a subset of the given 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.

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 [4].

* partially supported by the Israel Science Foundation grant 347/09 and DFG.

** partially supported by the German Research Foundation (BA 2168/3-1and MO 2402/1-1).

*** partially supported by the 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.

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 [15]. 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 arcs at all (this is the primary structure of the RNA)
– *Nested*: each base can be connected to at most one other base, and there are no crossing arcs
– *Crossing*: each base can be maximally connected to one other base
– *Bounded-Unlimited*: each base can be maximally connected to a constant number of other bases
– *Unlimited*: no restrictions on the arcs

Figure 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 figure 1).

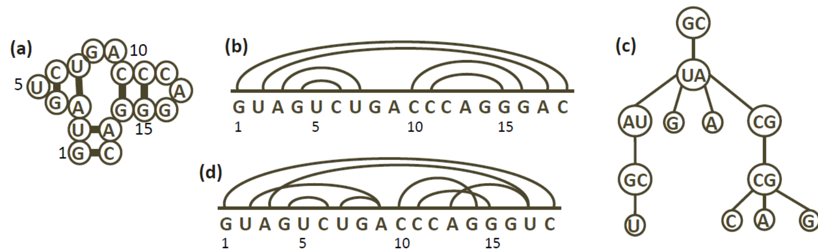

Fig. 1: RNA secondary structure representations: figures (a-c) represent the same RNA sample of length 18 with depth 5. (a) schematic two dimensional description of RNA folding (b) arc annotated sequence (c) an ordered tree: a single base is represented as a leaf and a base pair is represented as either a leaf (if the base pair's size 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.

There are several approaches to compute the similarity between two given RNAs, among them are tree similarity algorithms such as edit distance ([19], [20], [11], [5], [2], [6]), alignment ([10], [16], [1], [14]), and LAPCS ([7], [13], [9]). An edit distance between two ordered trees, T_1 and T_2, of sizes n and m ($n > m$), 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] for a thorough survey). Zhang and Shasha [20] present an edit distance algorithm that works in $O(nm \times min\{D_1, L_1\} \times min\{D_2, L_2\})$ where D_i is the depth of tree i and L_i is the number of leaves in tree i.

Klein [11] presents an $O(m^2 n \log n)$ algorithm, which in some cases performs better than the previous algorithm. Recently an optimal $O(n^3)$ decomposition algorithm for tree edit distance was given by Demaine et al. [5]. Ma et al. [21] 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. Jansson and Peng [8] 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.
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 [17] solve this problem in $O(n^2)$ time.

1.1 Our Results

In this work, we are looking for local exact pattern matching between two RNA molecules. We use the definitions from [17], 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 not supported in [17],[21], or any other algorithms based on tree edit distance, and it is one of our major achievements in this paper. Figure 2 demonstrates the *arc breaking* edit operation. The formal definitions of the problems are given in section 2.

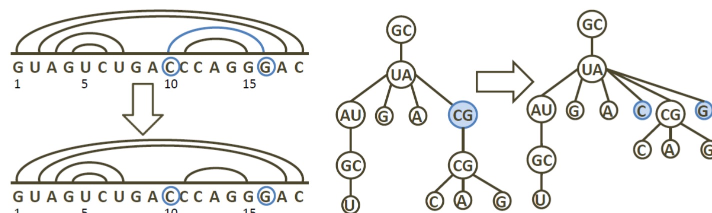

Fig. 2: Arc breaking operation: both representations show the result of the arc breaking operation for base pair CG in positions (10,16).

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 $O(n^3)$.
Due to space limitations we will describe the following algorithms in the extended version of this paper:

– An $O(n^3k^2)$ 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 $O(n^3k^2 \log n)$ for $(Nested, Bounded - Unlimited)$ RNAs.
– An $O(n^3)$ algorithm for computing the most similar sibling substructure between two $(Nested, Nested)$ RNAs, as defined in [8].

2 Notations and Definitions

RNA sequence 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, 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 *size* 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.

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 size 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.*

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

Definition 2 (Path). *A path in RNA R is a sequence of positions $(i_1, \ldots, i_y)$ such that $\forall 1 \leq k < y$, 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 set of positions $P = \{i_1, \ldots, i_y\}$ such that $\forall k, l \in P$ 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 sizes n and m respectively, an exact pattern matching (in short, matching) M, over R_1 and R_2 is a set of pairs $M = \{(i_1, j_1), \ldots, (i_k, j_k)|\ \forall 1 \leq \ell \leq k, 1 \leq i_\ell \leq n, 1 \leq j_\ell \leq m\}$ that satisfies the following conditions:*

1. *$S_1(i_\ell) = S_2(j_\ell)$ $\forall 1 \leq \ell \leq k$.*
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 $1 \leq x, y \leq k$, 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 $(bp_1, bp_2) \in M$.

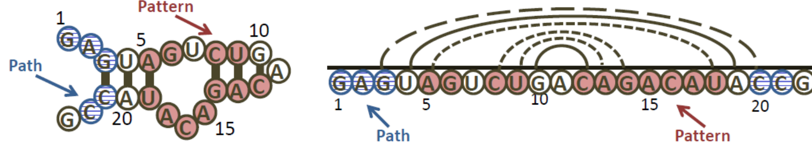

Fig. 3: Path and pattern examples in two different representations of the same RNA sample. A path is marked with horizontal lines and contains the bases {1,2,3,20,21}, a pattern is shadowed and contains the bases {5,6,8,9,12,13,14,15,16,17,18}. Note that the pattern contains the base pairs (5,18), (8,14)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

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 $\alpha : [1, |\Sigma|] \times [1, |\Sigma|] \to \mathbb{R}$ returns the score of matching two single bases:

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

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

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

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 figure 4).

2.1 Local Exact Pattern Matching Problem Definition

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 $\beta()$, 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 molecules. We will explore two different settings of RNAs: $(Nested, Nested)$ and $(Nested, Bounded - Unlimited)$.

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 2.1. We use similar ideas to those in Zhang and Shasha's tree edit distance

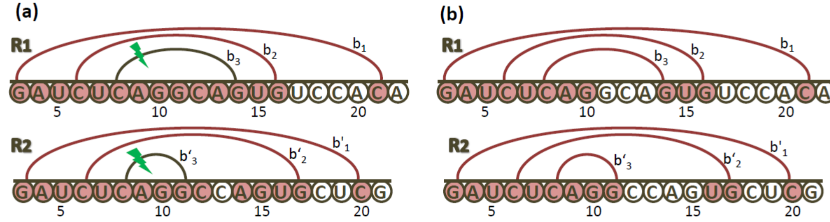

Fig. 4: Two matchings example. The figure presents two matching examples that can be defined between R_1 and R_2. In both cases the matchings are maximally extended. Note that matching (a) contains the base pairs (b_1, b'_1) and (b_2, b'_2), and matching (b) contains (b_1, b'_1), (b_2, b'_2) and (b_3, b'_3). The matching scores depend on the definition of α and β functions. Given $b_1 = (3, 21, 0.9)$, $b_2 = (6, 16, 0.6)$, $b_3 = (8, 14, 0.1)$, $b'_1 = (3, 20, 0.8)$, $b'_2 = (6, 17, 0.5)$, and $b'_3 = (8, 11, 0.3)$ and using our function definitions, $score(a) = 15 + (1.9 \cdot 1.8) + (1.6 \cdot 1.5) = 20.82$ and $score(b) = 12 + (1.9 \cdot 1.8) + (1.6 \cdot 1.5) + (1.1 \cdot 1.3) = 19.25$, thus the matching with the maximal score is (a).

algorithm [20]. 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 $O(n + m)$ ([12]). 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 sections 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.

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' sizes 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 figure 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 $s = (s_a, \ldots, s_b)$ and $t = (t_c, \ldots, t_d)$ that are

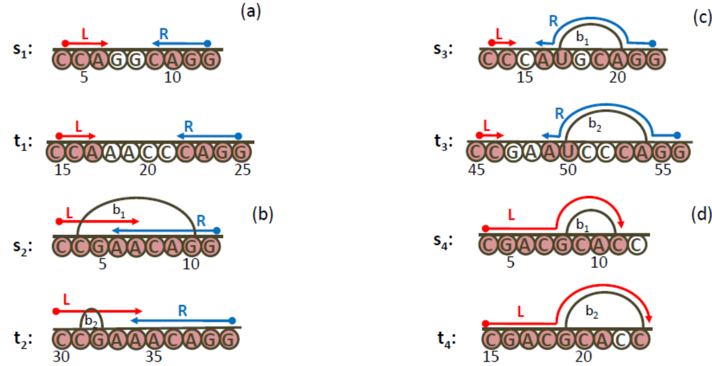

Fig. 5: $Lmatch$,$Rmatch$,$Full$ and $Score$ matchings between substrings s_i and t_i: $Lmatch$ is marked with 'L' right arrows, and $Rmatch$ is marked with 'R' left arrows. The probabilities of b_1 and b_2 to exist in s_i and t_i are 0.2 and 0.2, resp.
(a) Non-overlapping: $Lmatch(s_1, t_1)$ contains '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 (6,33) and (7,34), $Rmatch(s_2, t_2)$ contains positions (7,35) and (6,34). 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. $Lmatch(s_4, t_4) = 9.43$ by "jumping over" the base pairs b_1 and b_2, $Rmatch(s_4, t_4) = 9$. Hence, $Score(s_4, t_4) = Lmatch(s_4, t_4)$.

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 $s = (s_a, \ldots, s_i)$ and $t = (t_c, \ldots, t_j)$ 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 is not always greater than $Score$ (see figure 5 for examples).
We use $Score(a \dots i, c \dots j)$ to refer to $Score$ 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 values are computed according to the following equations (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 $M^{in}_{bp_1, bp_2}$ 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.

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 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.
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 $b_2 = (j, d, w) \in B_2$.
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: $M^{out}_{bp_1,bp_2} = Rextend(b+1, d+1) + Lextend(a-1, c-1)$.

3.4 Complete $O(n^4)$ Algorithm

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

(a) Compute the pattern matching inside all base pairs into M^{in}.
(b) Compute the extension tables $Rextend$ and $Lextend$ and the table M^{out} accordingly.
(c) 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 the total number of prefixes compared (since each of the allowed operations computation is done in constant time), which can be bounded by $O(n^4)$. In step (b), the computation of each entry in $Rextend$ and $Lextend$ tables is again done in constant time. Therefore, its time complexity is the number of entries in the tables, which is $O(n^2)$. In addition, in step (b) the algorithm computes M^{out} table is $O(n^2)$ time.

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 is done in constant time.
Therefore, the time complexity of the complete algorithm is $O(n^4+n^2+n^2) = O(n^4)$. 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 $O(n^3 \log n)$ Algorithm for Local Exact Pattern Matching

In this algorithm we take advantage of the fact that not all substrings that are compared as part of the $O(n^4)$ algorithm need to be compared. We use similar ideas of Klein's tree edit distance algorithm [11]. 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 size among the children of bp 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 $bp_1, heavy(bp_1), heavy(heavy(bp_1)), \dots$ 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 figure 6). We denote each light base pair as the *root* of the heavy path that it contains.

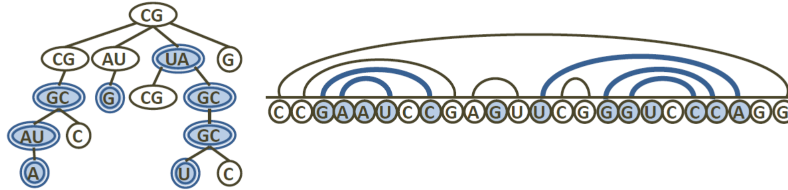

Fig. 6: Heavy path decomposition: in this RNA structure, we have three heavy path routes. They are presented in both tree and arc-annotated structures.

The following Lemma of Sleator and Tarjan [18] bounds the number of light base pairs that contain a base in R:

Lemma 1 (Sleator and Tarjan [18]). *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 a substring $s = (s_a, \dots, s_b)$, that is defined over a base pair $bp = (a, b, p) \in B_1$ with $heavy(bp) = (x, y, r) \in B_1$, consists of the suffixes of $(s_a, \dots, s_y)$ starting at positions $a, \dots, x$, and the prefixes of $(s_a, \dots, s_b)$ ending at positions $y, \dots, b$ (see figure 7).*

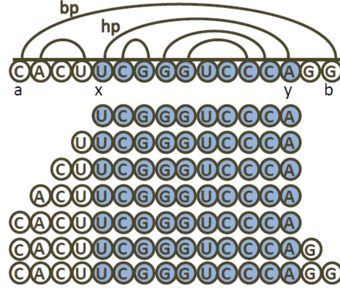

Fig. 7: Special Substrings Example: the special substrings of a base pair, $bp = (a, b, p)$, with a heavy child base pair, $hp = (x, y, r)$.

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 the base pair is: $size(bp) - size(hp)$.

Let $bp_1 = (a, b, p) \in B_1$ with $heavy(bp) = hp = (x, y, r) \in B_1$, $bp_2 = (c, d, q) \in B_2$, 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 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 is reached (the suffixes special substrings). Then, we continue with the prefixes of bp, starting from $s_a, \ldots, s_y$, and continue going from left to right until the right endpoint of bp is reached.

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 prefixes comparisons the computation is exactly as described in section 3.2. For the suffixes comparisons the values are set according to the following equations:

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

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

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

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

where $b_1 = (i, e, r) \in B_1$ and $b_2 = (j, f, w) \in B_2$.
Eventually, the value in M^{in} table is set to:
$M^{in}_{bp_1,bp_2} = \beta(bp_1, bp_2) + Score(a \dots b, c \dots d)$.

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 $O(n^3 \log n)$ time.

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 [5]. 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 [5]. 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 figure 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 figure 8 for an example).
This enhancement to the algorithm improves the time complexity to $O(n^3)$. The intuition to 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 complete proof in [5]).

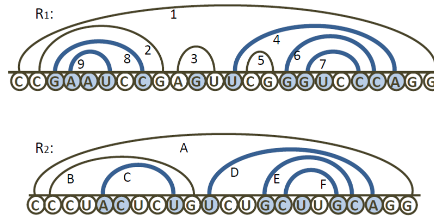

Fig. 8: Heavy Path Decomposition of Both RNA Molecules: R_1 contains the heavy path $(1, 4, 6, 7, U)$. In addition R_1 contains the heavy paths $(2, 8, 9, A)$, $(3, G)$, and 5. In the comparison between $6 \in B_1$ and $E \in B_2$ the dominant base pair is 6, whereas in the comparison between $8 \in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

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

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. 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.
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 $O(n^3 \log n)$.

References

1. R. Backofen, S. Chen, D. Hermelin, G.M. Landau, M.A. Roytberg, O. Weimann, and K. Zhang. Locality and gaps in rna comparison. *Journal of Computational Biology*, 14:1074–1087, 2007.

2. R. Backofen, G.M. Landau, M. Möhl, D. Tsur, and O. Weimann. Fast rna structure alignment for crossing input structures. In *Combinatorial Pattern Matching, 20th Annual Symposium, CPM 2009, Lille, France, June 22-24, 2009, Proceedings*, volume 5577 of *Lecture Notes in Computer Science*, pages 236–248, 2009.
3. P. Bille. A survey on tree edit distance and related problems. *Theoretical Computer Science*, 337:217–239, 2005.
4. J. Couzin. Small rnas make big splash. *Science*, 298(5602):2296–2297, 2002.
5. E.D. Demaine, S. Mozes, B. Rossman, and O. Weimann. An optimal decomposition algorithm for tree edit distance. In *Automata, Languages and Programming, 34th International Colloquium, ICALP 2007, Wroclaw, Poland, July 9-13, 2007, Proceedings*, volume 4596 of *Lecture Notes in Computer Science*, pages 146–157, 2007.
6. S. Dulucq and H. Touzet. Decomposition algorithms for the tree edit distance problem. *J. Discrete Algorithms*, 3(2-4):448–471, 2005.
7. P.A. Evans. *Algorithms and Complexity for Annotated Sequence Analysis*. PhD thesis, University of Alberta, 1999.
8. J. Jansson and Z. Peng. Algorithms for finding a most similar subforest. *Theory Comput. Syst*, 48(4):865–887, 2011.
9. T. Jiang, G. Lin, B. Ma, and K. Zhang. The longest common subsequence problem for arc-annotated sequences. *J. Discrete Algorithms*, 2(2):257–270, 2004.
10. T. Jiang, L. Wang, and K. Zhang. Alignment of trees – an alternative to tree edit. *TCS: Theoretical Computer Science*, 143, 1995.
11. P.N. Klein. Computing the edit-distance between unrooted ordered trees. In *Algorithms - ESA '98, 6th Annual European Symposium, Venice, Italy*, pages 91–102, 1998.
12. G. M. Landau and U. Vishkin. Fast parallel and serial approximate string matching. *Journal of Algorithms*, 10:157–169, 1989.
13. G.H. Lin, Z.Z. Chen, T. Jiang, and J. Wen. The longest common subsequence problem for sequences with nested arc annotations. *JCSS: Journal of Computer and System Sciences*, 65(3):465–480, 2002.
14. M. Mohl, S. Will, and R. Backofen. Lifting prediction to alignment of rna pseudoknots. In *Research in Computational Molecular Biology, 13th Annual International Conference, RECOMB 2009, Tucson, AZ, USA, May 18-21, 2009. Proceedings*, volume 5541, pages 285–301, 2009.
15. P.B. Moore. Structural motifs in rna. *Annual Review of Biochemistry*, 68:287–300, 1999.
16. S. Schirmer and R. Giegerich. Forest alignment with affine gaps and anchors. In *Combinatorial Pattern Matching - 22nd Annual Symposium, CPM 2011, Palermo, Italy, June 27-29, 2011. Proceedings*, volume 6661 of *Lecture Notes in Computer Science*, pages 104–117, 2011.
17. S. Siebert and R. Backofen. A dynamic programming approach for finding common patterns in rnas. *Journal of Computational Biology*, 14(1):33–44, 2007.
18. D. D. Sleator and R. E. Tarjan. A data structure for dynamic trees. *Journal of Computer and System Sciences*, 26(3):362–391, 1983.
19. K.C. Tai. The tree-to-tree correction problem. *JACM: Journal of the ACM*, 26(3):422–433, 1979.
20. K. Zhang and D. Shasha. Simple fast algorithms for the editing distance between trees and related problems. *SIAM Journal on Computing*, 18(6):1245–1262, 1989.
21. K. Zhang, L. Wang, and B. Ma. Computing similarity between rna structures. In *Combinatorial Pattern Matching*, volume 1645 of *Lecture Notes in Computer Science*, pages 281–293. 1999.