Development and Implementation of an Alignment Program for Canonical Pseudoknots

Bachelorarbeit

Bettina Hübner

Universität Freiburg

Lehrstuhl für Bioinformatik

1. Gutachter: Dr. Ing. Mathias Möhl

2. Gutachter: Prof. Dr. Rolf Backofen

Abgabetermin: 18. Februar 2011

ERKLÄRUNG

Hiermit erkläre ich, dass ich diese Abschlussarbeit selbständig verfasst habe, keine anderen als die angegebenen Quellen/Hilfsmittel verwendet habe und alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten Schriften entnommen wurden, als solche kenntlich gemacht habe. Darüber hinaus erkläre ich, dass diese Abschlussarbeit nicht, auch nicht auszugsweise, bereits für eine andere Prüfung angefertigt wurde.

____________________ ____________________________________

Ort, Datum Unterschrift

Contents

1 Introduction **5**

2 Definitions and fundamentals **7**
2.1 Arc-annotated sequences . . . 7
2.1.1 Crossing arcs / Pseudoknots . . . 7
2.1.2 Nested / crossing arc-annotated sequences . . . 7
2.2 Alignment of arc-annotated sequences . . . 7
2.2.1 Alignment costs . . . 8
2.3 Fragments . . . 9
2.3.1 Arc-complete fragments . . . 9
2.3.2 Atomic fragments . . . 9
2.4 Alignments of fragments . . . 10
2.4.1 Optimal cost of an alignment A restricted to fragments F_a, F_b . . . 10
2.5 Splits . . . 10
2.5.1 Arc-preserving splits . . . 10
2.5.2 Basic type of a split . . . 10
2.5.3 Compatible splits . . . 11
2.5.4 Size constrained split types . . . 11
2.6 Parse tree of a sequence . . . 11

3 The alignment algorithm scheme **12**

4 The alignment algorithm for R&G Pseudoknots **15**
4.1 The R&G Pseudoknot class . . . 15
4.2 How the algorithm works . . . 16
4.2.1 Base case at leaf nodes . . . 17
4.2.2 Arcmatch . . . 18
4.2.3 Split type 1'21' . . . 18
4.2.4 Split type 2'1 . . . 19
4.2.5 Split type 12' . . . 19
4.2.6 Split type 12 . . . 19
4.2.7 Pseudoknots . . . 20
4.2.8 Complexity . . . 22
4.3 A few details about the implementation . . . 23
4.4 Results . . . 23

Zusammenfassung. In dieser Arbeit ging es darum, ein Sequenz-Struktur-Alignmentprogramm für RNA mit kanonischen Pseudoknoten zu entwickeln und in der Programmiersprache C++ zu implementieren. Der Ausgangspunkt hierfür war der am Lehrstuhl für Bioinformatik der Universität Freiburg entwickelte Algorithmus für eingeschränkte Pseudoknotenklassen. Dieses allgemeine Schema sollte für eine effiziente Implementierung für die Klasse der Reeder&Giegerich Pseudoknoten umgesetzt werden. Die Verwendung einer eingeschränkten Klasse von Pseudoknoten ermöglicht einen effizienteren Algorithmus, der in gleichem Maße wie der zugrunde liegende Struktur-Vorhersage-Alogrithmus von den Einschränkungen profitiert. Die Zerlegungsmethode der Pseudoknotenklasse ist hier die Grundlage für die verwendete dynamische Programmierung. Reeder&Giegerich Pseudoknoten bilden die Klasse der einfachen, rekursiven kanonischen Pseudoknoten mit maximal erweiterten Helices.

1 Introduction

It has recently become clear that RNA molecules do not only act as a messenger, they play very important roles in a cell [1]. These roles involve regulatory and catalytic ones [5]. The functions of non-coding RNAs are related to the structure which makes investigating its structure interesting.

Existing approaches for the computational analysis of non-coding RNA mostly focus on nested secondary structure and neglect pseudoknots since the prediction of pseudoknots is NP-hard [5]. However, pseudoknot motifs are "among the most prevalent RNA structures" as stated in [6] and [1] and they are "functionally relevant in many RNA mediated processes" as stated in [3].

The major problems concerning the analysis of pseudoknotted RNAs [1] depend on each other: There are only few known pseudoknots and their prediction is computationally very expensive. Furthermore, the existing prediction programs are not very reliable. This is mainly caused by the shortage of known structures where the prediction programs can be trained on. On the other hand, the high computational cost and the low prediction quality complicate the research on pseudoknot structures.

The most promising solution is the use of comparative approaches for the prediction of pseudoknotted secondary structure. This requires an alignment of both sequence *and* structure since the structure of RNA is more conserved than the sequence. However, "the research on this topic is far behind the computational analysis of pseudoknot structure prediction" as stated in [1]. The existing algorithms for pseudoknot structure prediction focus on restricted classes of pseudoknots and use their properties to solve the prediction problem in a dynamic programming approach.

The bioinformatics group at the technical faculty of the university Freiburg has developed a general scheme for sequence-structure alignment with known pseudoknot structures [1]. For a restricted class of pseudoknots it generates a pseudoknot alignment algorithm using the decomposition strategy and thus the dynamic programming approach of

the structure prediction. This strategy makes the algorithm efficient: It benefits from the structural restrictions in the same way as the prediction and has only a linear increase in complexity.

A general version of this algorithm scheme has been implemented (*PKalign* [2]) but is rather slow due to its generality. This motivated the topic of this thesis which is the development and implementation of one tailored instance for the class of R&G pseudoknots [3].

2 Definitions and fundamentals

This part is based upon [1].

First of all we need some formal definitions: The algorithm takes as input two sequences of RNA together with their fixed input structures. For one of them, a parse tree is constructed and using this tree and methods of dynamic programming, the alignment is recursively computed.

2.1 Arc-annotated sequences

So-called arc-annotated sequences are used to symbolize a sequence of RNA together with its structure i.e., its base pairs. Let $A = \{A, U, C, G\}$ be an alphabet representing the four bases of an RNA sequences. S is a string over A and P is a set of pairs (l,r) with left end $p^L = l$ and right end $p^R = r$. Then an arc-annotated sequence is a pair (S, P) where P represents bonds between bases.

- $S[i]$ denotes the i-th symbol of S.
- An arc (l,r) with $1 \leq l < r \leq |S|$ represents a bond between the bases S[l] and S[r].
- Each base can occur in only one arc.
- For two arcs $p_1 = (l_1, r_1)$ and $p2 = (l_2, r_2)$, $l_1 = l_2$ implies $r_1 = r_2$ (since each base cannot be part of multiple arcs).

2.1.1 Crossing arcs / Pseudoknots

Two arcs $p_1 \in P$ and $p_2 \in P$ are crossing iff $p_1^L < p_2^L < p_1^R < p_2^R$ or $p_2^L < p_1^L < p_2^R < p_1^R$ i.e., they form a pseudoknot.

2.1.2 Nested / crossing arc-annotated sequences

A nested arc-annotated sequence contains only non-crossing arcs. If a sequence contains at least two arcs that are crossing, it is called crossing.

2.2 Alignment of arc-annotated sequences

An Alignment of two arc-annotated sequences (S_1, P_1) and (S_2, P_2) is represented by a set $M \cup G$ where M is a set of so-called match edges and G is a set of gap edges.

- M is a set of pairs $(i, j) \in [1...|S_1|] \times [1...|S2|]$. For (i_1, j_1), $(i_2, j_2) \in M$, $i_1 > i_2$ implies $j_1 > j_2$ and $i_1 = i_2$ implies $j_1 = j_2$.
- G is a set $\{(x, -)|x \in [1...|S_1|] \wedge \nexists y.(x, y) \in M\} \cup \{(-, y)|y \in [1...|S2|] \wedge \nexists x.(x, y) \in M\}$ that aligns bases x and y to gaps.

$(i, j) \in M$ matches bases $S_1[i]$ and $S_2[j]$. Two arcs $p_1 \in P_1$ and $p_2 \in P_2$ are matched if M contains match edges (p_1^L, p_2^R) and (p_1^R, p_2^R).

2.2.1 Alignment costs

For each alignment a cost based on edit distance can be computed using two kinds of operations that were first introduced by Jiang et al.[8]

Base operations are used for positions that are not incident to arcs:

- *Base mismatch*: replace a base with another
  - cost: w_m
- *Base insertion/deletion*: align a base to a gap i.e., delete or insert it
  - cost: w_d

The second type of operation is used if at least one position of the edge is adjacent to an arc.

- *Arc mismatch*: replace one or two of the bases incident to an arc
  - cost: $\frac{w_{am}}{2}$ if one base is replaced, w_{am} if both are replaced
- *Arc breaking*: remove one arc but leave the adjacent bases unchanged
  - cost: w_b
- *Arc removing*: remove one arc and delete its bases
  - cost: w_r
- *Arc altering*: remove one of the two bases of an arc
  - cost: $w_a = \frac{w_b}{2} + \frac{w_r}{2}$

For computing the cost of an alignment A of two arc-annotated sequences (S_1, P_1) and (S_2, P_2) we use the following definitions[1]:

$$\chi(i, j) := \text{ if } S_1[i] \neq S_2[j] \text{ then } 1 \text{ else } 0$$

$$\psi_k(i) := \text{ if } \exists p \in P_k . p^L = i \text{ or } p^R = j \text{ then } 1 \text{ else } 0$$

The cost for aligning $S_k[i]$ to a gap is given by

$$\text{gap}_k(i) := w_d + \psi_k(i)(\frac{w_r}{2} - w_d)$$

The cost for aligning $S[i]$ to $S[j]$ assuming that their possibly adjacent arcs are not matched can be computed with

$$\text{basematch}(i, j) := \chi(i, j) w_m + (\psi_1(i) + \psi_2(j)) \frac{w_b}{2}$$

[1] $k \in \{1, 2\}$

Using this, the cost for an alignment can be computed recursively which enables using the technique of dynamic programming:

$$\begin{aligned}
\mathrm{cost}(\{(i,-)\} \uplus A') &= \mathrm{gap}_1(i) + \mathrm{cost}(A') \\
\mathrm{cost}(\{(-,j)\} \uplus A') &= \mathrm{gap}_2(j) + \mathrm{cost}(A') \\
\mathrm{cost}(\{(l_1,l_2),(r_1,r_2)\} \uplus A') &= (\chi(l_1,l_2) + \chi(r_1,r_2))\tfrac{w_{am}}{2} + \mathrm{cost}(A') \\
& \quad \text{if } (l_1,r_1) \in P_1, (l_2,r_2) \in P_2 \\
\mathrm{cost}(\{(i,j)\} \uplus A') &= \mathrm{basematch}(i,j) + \mathrm{cost}(A') \\
& \quad \text{if third case is not applicable} \\
& \quad \text{(i.e., adjacent arcs are not matched.)}
\end{aligned} \tag{1}$$

2.3 Fragments

An arc-annotated sequence (S,P) can be decomposed into fragments i.e., subsequences and combinations of subsequences. The dynamic programming alorithm presented in this thesis computes the optimal alignment step by step from optimal alignments for fragments of the two sequences.

Formally, a fragment F of degree k is a k-tuple of intervals $([l_1,r_1],..,[l_k,r_k])$ with $1 \leq l_1 \leq r_1+1 \leq \ldots \leq l_k \leq r_k+1 \leq |S|$, where each $[l_i,r_i]$ denotes a subsequence of S. l_1, r_1 are the boundaries of F and the ranges between the intervals are called gaps of F (e.g. $[r_1+1, l_2-1]$). An empty interval is denoted by $[i+1,i]$ and is allowed due to the above given definition.

For a fragment we consider the following concepts:

- $\hat{F}$: set of positions covered by F
- boundaries of F: $l_1, r_1, \ldots, l_k, r_k$
- gaps of F: the ranges between the intervals, e.g. $[r_1+1, l_2-1]$
- empty intervals $[i+1,i]$ are allowed
- $F[i]$ denotes the i-th interval of F with left boundary $F[i]^L$ and right boundary $F[i]^R$
- $\langle i \rangle$ is abbreviation for $[i,i]$ (interval of size 1)

2.3.1 Arc-complete fragments

A fragment F is arc-complete iff $\forall (l,r) \in P. l \in \hat{F} \Leftrightarrow r \in \hat{F}$.

2.3.2 Atomic fragments

An atomic fragment F covers either a single base position that is not incident to an arc or it covers exactly the two ends of an arc $(l,r) \in P$.

2.4 Alignments of fragments

An alignment can not only be considered for whole sequences, instead it can also be restricted to certain fragments of sequences. Formally, define the restriction of an alignment A to fragments F_a,F_b as

$$A|_{F_a \times F_b} := \{(i,j) \in A | i \in \hat{F}_a \cup \{-\}, j \in \hat{F}_b \cup \{-\}\}$$

For two fragments F_a, F_b, both having the same degree k, their alignment A is denoted by $align_A(F_a, F_b)$ and fulfills the following condition:

$$(\forall (a_1, a_2) \in A) \wedge (\forall i \in \{1, \dots, k\}) \ (a_1 = -) \vee (a_2 = -) \vee (a_1 \in F_a[i] \Leftrightarrow a_2 \in F_b[i])$$

In other words, a pair (a_1, a_2) of $align_A(F_a, F_b)$ either aligns a position from F_a or a position from F_b to a gap or it aligns a position from the i-th interval of F_a to a position from the i-th interval of F_b.

2.4.1 Optimal cost of an alignment A restricted to fragments F_a, F_b

The cost of aligning F_a and F_b by alignment A is defined as $C_A(F_a, F_b) := cost(A|_{F_a \times F_b})$ It can be computed using the equations 1. The optimal cost for an alignment of the two fragments F_a and F_b can be obtained by choosing the minmal cost among all possible alignments of F_a and F_b:

$$C(F_a, F_b) := \min_{A \text{ with } align_A(F_a, F_b)} \{C_A(F_A, F_B)\}$$

2.5 Splits

A sequence or a fragment of a sequence can be split into smaller fragments.

Formally, a split of F is a pair (F^1, F^2) where F, F^1 and F^2 are fragments of the same sequence $\Leftrightarrow \hat{F} = \hat{F}^1 \uplus \hat{F}^2$

F^1 and F^2 are the children and F the parent of the split. This concept can easily be extended for n-ary splits which is neccessary for the R&G class of pseudoknot [3].

2.5.1 Arc-preserving splits

A split (F^1, F^2) of F is called *arc-preserving* if fragments F, F^1 and F^2 each are arc-complete. Two examples for arc-preserving splits can be found in Figure 1.

2.5.2 Basic type of a split

Every split (F^1, F^2) has exactly one basic type that can be determined in the following way: The interval $[min(\hat{F}), max(\hat{F})]$ consists of the intervals of F^1, the intervals of F^2 and the gaps of F. By ordering them from left to right and replacing each interval of F^1 by 1, each interval of F^2 by 2 and the gaps by G, we construct a string T over $\{1, 2, G\}$. This string T is the *basic type of the split* and (F^1, F^2) is called a T-split.

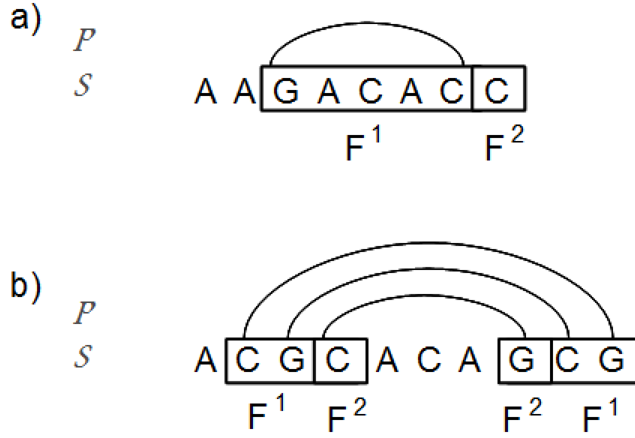

Figure 1: **Two examples for arc-preserving splits of a fragment F of an arc-annotated sequence** (S, P)**.** a) (F^1, F^2) has basic split type 12 and constrained type 12', F^2 is an atomic fragment. b) (F^1, F^2) is of basic split type 12G21 and constrained type 12'G2'1, F^2 is an atomic fragment.

Remark: For n-ary splits, this concept is extended such that T is a string over $\{1, 2, \ldots, n, G\}$ where n is the number of children of the split. Apart from that, T can be obtained in the same way.

2.5.3 Compatible splits

Two splits with the same split type are compatible.

2.5.4 Size constrained split types

A basic type of a split can be further refined by annotating constraints and is then called a constrained type. Introducing a size constraint means restricting the size of an interval that is part of the type. To indicate this, the symbol 1 or 2 belonging to the size-constrained interval is marked with '. Size constraints are used to show that an atomic fragment is split off. A size constraint reduces the number of splits of this type by reducing the degrees of freedom by one.

Examples for the concept of basic split types and size constrained split types are given in Figure 1.

2.6 Parse tree of a sequence

A fixed recursive decomposition of a sequence (S, P) can be represented by a parse tree. Formally, a parse tree is a binary tree where each node stands for an arc-complete fragment of (S, P). Furthermore, the root covers all positions of the sequence i.e., the interval $[1, |S|]$. Each inner node represents a fragment F and has two children F^1 and F^2 forming an arc-preserving split (F^1, F^2) of F. Finally, each leaf is an atomic fragment.

3 The alignment algorithm scheme

The alignment algorithm scheme presented in this section has been developed at the Chair of Bioinformatics at the University of Freiburg and is thus heavily based upon [1] and [2].

The alignment algorithm is a dynamic programming algorithm that uses a recursive decomposition of the RNA sequences. The input consists of two arc-annotated sequences (S_a, P_a) and (S_b, P_b) and a parse tree[2] representing a fixed recursive decomposition of the first sequence. As this already implies, the sequences are handled asymmetrically.

The parse tree and hence the decomposition of (S_a, P_a) determine the algorithm recursion: For all splits given by the tree, all compatible splits of the second sequence are examined. These compatible splits do not have to be arc-preserving. More precisely, for all fragments in the parse tree alignments to all fragments of the second sequence are constructed on condition that they are of the same basic type.

The algorithm computes the optimal alignment for fragments $F_a = ([1, |S_a|])$ covering the entire first sequence and $F_b = ([1, |S_b|])$ and thereby yields the optimal cost $C(F_a, F_b)$.

For the computations the following lemma is used:

Lemma 1 (Split lemma) Let F_a be a fragment of (S_a, P_a), F_b a fragement of (S_b, P_b) and let (F_a^1, F_b^2) be an arc-preserving split of F_a having basic type T. Then

$$C(F_a, F_b) = \min_{T\text{-split}(F_b^1, F_b^1) \text{ of } F_b} \{C(F_a^1, F_b^1) + C(F_a^2, F_b^2)\}$$

This means considering all possible alignments to a T-split of the second sequence and choosing the one with minimal cost.

Since the splits of F_b in contrast to the ones of F_a do not need to be arc-preserving, they may contain arcs that cannot be matched to arcs of F_a i.e., they are broken or removed. Fortunately, the costs for these operations are correctly computed in $C(F_a^1, F_b^1)$ and $C(F_a^2, F_b^2)$ since all costs are local to one single base except for matching an arc.

In order to do the evaluation of the recursion efficiently all intermediate values are tabulated which is typical for dynamic programming. In this case, the resulting values $C(F_a, F_b)$ are stored to avoid recomputations.

At the leafs of the tree that represent atomic fragments the base cases of the recursion are applied:

- The first base case is used for leaf nodes that cover one single position of the sequence:

$$C(\langle i \rangle, [l, r]) = \min \begin{cases} C(\langle i \rangle, [l+1, r]) + \text{gap}_2(l) & \text{if } \; l \leq r \\ C(\langle i \rangle, [l, r-1]) + \text{gap}_2(r) & \text{if } \; l \leq r \\ \text{basematch}(i, l) & \text{if } \; l = r \\ \text{gap}_1(i) & \text{if } \; l > r \end{cases} \tag{2}$$

[2] in my implementation, the first step is parsing Sequence (S_a, P_a) and constructing the parse tree.

- The second base case is used for leaf nodes that represent the two end points of an arc $p = (p^L, p^R)$:

$$C(F_a = (\left\langle p^L \right\rangle, \left\langle p^R \right\rangle), ([l_1, r_1], [l_2, r_2])) =$$

$$\min \begin{cases} C(\left\langle p^L \right\rangle, [l_1, r_1]) + C(\left\langle p^R \right\rangle, [l_2, r_2]) & \\ C(F_a, ([l_1 + 1, r_1], [l_2, r_2])) + \mathrm{gap}_2(l_1) & \text{if} \quad l_1 \leq r_1 \\ C(F_a, ([l_1, r_1 - 1], [l_2, r_2])) + \mathrm{gap}_2(r_1) & \text{if} \quad l_1 \leq r_1 \\ C(F_a, ([l_1, r_1], [l_2 + 1, r_2])) + \mathrm{gap}_2(l_2) & \text{if} \quad l_2 \leq r_2 \\ C(F_a, ([l_1, r_1], [l_2, r_2 - 1])) + \mathrm{gap}_2(r_2) & \text{if} \quad l_2 \leq r_2 \\ (\chi(p^L, l_1) + \chi(p^R, l_2)) \frac{w_{am}}{2} & \text{if} \quad (l_1, l_2) = (r_1, r_2) \in P_b \end{cases} \tag{3}$$

For each split (F_a^1, F_a^2) with basic type T in the parse tree the following recursion is applied:

$$C(F_a, F_b) = \min_{T\text{-split}(F_b^1, F_b^2) \text{ of } F_b} \{C(F_a^1, F_b^1) + C(F_a^2, F_b^2)\}$$

After computing $C(F_a, F_b)$ for all nodes of the parse tree, the actual alignment is constructed by doing a back-trace that starts at the root node of the tree.

To analyze the *complexity of the algorithm* I first need to mention some further considerations concerning the number of children and parents of a split type. The number of parents influences the complexity since for each fragment in the first sequence, all fragments in the second sequence having the same split type are considered. The number of children determines in how many ways an aligned fragment can be split in sub-alignments.

Let m be the length of the second sequence. Then the number of children is defined as

$$\#_C^m(T) = |\{(F^1, F^2) \mid (F^1, F^2) \text{ is a } T\text{-split of some } F \text{ and } \hat{F} \subseteq [1, m]\}|$$

and the number of parents is defined as

$$\#_P^m(T) = |\{F \mid \exists (F^1, F^2) \text{ that is a } T\text{-split of } F \text{ and } \hat{F} \subseteq [1, m]\}|$$

These numbers are restricted by the following lemma:

Lemma 2 Let T be a split type of some sequence with length m and let k_p, k_1 and k_2 be the degrees of the parents and the two children of the split. Additionally, let the constraints of T reduce the degrees of freedom by a factor c and denote the corresponding reduction for the parent instances by $c' \leq c$. Then $\#_C^m(T) \in O(m^{k_p + k_1 + k_2 - c})$ and $\#_P^m(T) \in O(m^{2k_p - c'})$.

As mentioned above, let m be the length of the second sequence and let n be the one of the first sequence. Then, the number of nodes in the parse tree is bounded by $O(n)$

since with each split at least one new boundary is introduced and in total there exist only $O(n)$ many.

Let T_p and T_c be types of splits in the parse tree and let the according $\#_P^m(T_p)$ and $\#_C^m(T_c)$ be maximal among the occurring split types. The algorithm computes the cost $C(F_a, F_b)$ for each node F_a having split type T_p for $\#_P^m(T_p)$ fragments. If one assumes the worst case for each node, this leads to a space complexity of $O(n \cdot \#_P^m(T_p))$.

The computation of T_c-splits dominates the time complexity : According to the split lemma, the algorithm chooses the minimum among $\#_C^m(T_c)$ terms where each term is computed in $O(1)$. This leads to a worst case time complexity of $O(n \cdot \#_C^m(T_c)) \cdot \#_P^m(T_p)$ and $\#_C^m(T_c)$ is asymptotically bounded by Lemma 2. Assuming non-constrained basic split types we get a $O(nm^{2k})$ space and $O(nm^{3k})$ time complexity, with k being the maximal degree among the splits in the parse tree.

A possibilty to *reduce the complexity* is given by the fact that it directly depends on $\#_P^m(T)$ and $\#_C^m(T)$ for the basic types and also by Lemma 2 showing that these numbers can be reduced by constraints of the types. Hence, this potential can be used to improve the complexity. For this purpose, the split lemma and also the recursive cases need to be modified. The reason for this is that an atomic fragment of the first sequence is not neccessarily aligned to an atomic fragment of the second sequence: Let (F_a^1, F_b^1) be a split having constraint type T where F_a^1 is atomic and let the corresponding unconstrained type be T_u. The algorithm aligns F_a^1 to F_b^1 of a T_u-split (F_B^1, F_B^2). For a non-atomic F_b^1 the split lemma implies that at most one of its bases per interval is matched to F_a^1 and the others are aligned to gaps. By introducing so-called 'shrink-cases' and thereby 'eating away' the gaped bases a fragment F_b^1 of F_b can be split off that satisfies the constraint type T.

The following lemma is an extension of the split lemma allowing size constraints in the split type T.

Lemma 3 (Split lemma for constrained types) Let F_a be a fragment of (S_a, P_a) and let F_b be a fragment of (S_b, P_b). Furthermore, let (F_a^1, F_a^2) be an arc-preserving T-split of F_a with size constraints for at most one of the fragments. In addition, at least one boundary of each interval of the constrained fragment must coincide with a boundary of F_a, then these constrained boundaries can be removed by additional shrink cases. Finally, let A be an optimal alignment of F_a and F_b. Then there are two possibilities how A aligns F_a to a T-split (F_b^1, F_B^2) of F_b. Either the constrained fragment of the split is matched to one or two gaps by A and the remaining fragment is aligned to F_a, or A aligns F_a^1 to F_b^1 and F_a^2 to F_b^2.

The remaining step now is to modify the recursive case. Here, $C(-, F_b^i)$ denotes the cost of deleting F_b^i. The base cases as given in equations 2 and 3 stay the same.

$$C(F_a, F_b)) = \min_{T\text{-split}(F_b^1, F_b^2) \text{ of } F_b} \min \begin{cases} C(F_a^1, F_b^1) + C(F_a^2, F_b^2) & \\ C(F_a, F_b^2) + C(-, F_b^1) & \text{if} \quad T \text{ contains some } 1' \\ C(F_a, F_b^1) + C(-, F_b^2) & \text{if} \quad T \text{ contains some } 2' \end{cases}$$

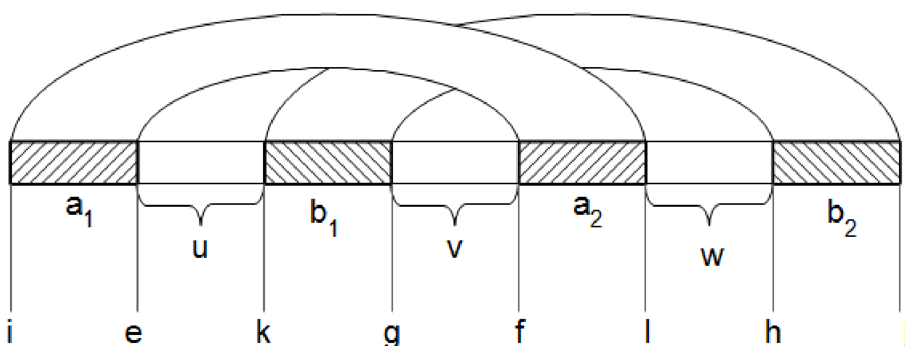

Figure 2: Structure of a canonical pseudoknot

4 The alignment algorithm for R&G Pseudoknots

This section deals with the variant of the alignment algorithm for the restricted class of R&G pseudoknots. As a first step I will describe the pseudoknot class which has been developed by Jens Reeder and Robert Giegerich from the university of Bielefeld [3]. Afterwards, I am going to show the modified recursions for the different split types and mention a few details about my implementation. The final part gives a runtime comparison to the general implementation of the alignment algorithm scheme done by Mathias Möhl [2].

4.1 The R&G Pseudoknot class

In general, RNA secondary structure prediction is NP-complete for arbitrary pseudoknots if the thermodynamic model is used. A solution for this problem is to restrict the classes of pseudoknots which enables polynomial time algorithms. Analogously, the alignment can also benefit from structural restriction.

One example for a restricted class of pseudoknots is the one developed by Jens Reeder and Robert Giegerich [3], the so-called class of canonical simple recursive pseudoknots, which means simple recursive pseudoknots that are further restricted by three rules of canonization.

- Two helices that interact crosswise form a *simple pseudoknot.*
- *Simple recursive pseudoknots* are an extension of simple pseudoknots: Here, the unpaired strands u,v,w (see Figure 2) can internally fold in an arbitrary way. This also includes that they can form further recursive pseudoknots.

The class of simple recursive pseudoknots is reduced to the class of *canonical simple recursive pseudoknots* by three canonization rules. These rules also influence the number of independent moving boundaries between the different parts of a pseudoknot and thereby improve the runtime of the prediction algorithm.

- *Rule I:*
  - The two strands of a helix must be of the same size: $|a_1| = |a_2|$ and $|b_1| = |b_2|$. This decreases the number of moving boundaries from 8 to 6 since it implies $f = l - (e - i)$ and $h = j - (g - k)$.
  - Furthermore, both helices are not allowed to contain bulges.
- *Rule II:*
  - The helices a_1, a_2, b_1 and b_2 must be *maximally extended* which also means that v must be as short as possible. The maximum possible extension depends on the bases of the input sequence and means that the helices must be extended as long as valid Watson-Crick base pairs can be formed.
  - This rule also reduces the number of boundaries since the values of i and l already fix the maximal helix length of a_1 and a_2. Analogously, the choice of k and j influences the length of b_1 and b_2.
  - $e = i + \text{maximalLength}(i, l)$
  - $g = k + \text{maximalLength}(k, j)$
  - Consequently, only four independently moving boundaries remain (i,k,l,j).
- *Rule III:*
  - If two maximal helices overlap i.e, they compete for the same bases of v, an arbitrary point between them is chosen as boundary.

Using the split types for fragments introduced in section 2.5.2 on page 10 and section 2.5.4 on page 11 R&G structures can be decomposed in the following way[3]:
$2'1 \quad 12' \quad 12 \quad 1'21' \quad E_1 : 1^c23^c41^c53^c \quad E_2 : 12'G2'1$
These are the possible split types that can occur in a parse tree for a RNA sequence with a canonical pseudoknot. Split type E_1 is used for canonical pseudoknots where fragments 1 and 3 form the two helices that are further decomposed by E_2.

4.2 How the algorithm works

We have seen that parse trees with a certain restricted set of split types exist for the class of R&G Pseudoknots. Now we are going to consider an optimized variant of the general alignment algorithm scheme that uses an optimized version for each kind of split type.

This variant of the algorithm fitted for RG-PKs uses the results of lemma 3 and the modified recursive case. All kinds of split types used here contain either a length constraint for one or two of the fragments or they contain the constraint for canonical pseudoknot stems. This gives the possibility to use shrink cases and modify the general

[3] E_1 uses a new kind of type constraint ($.^c$) meaning fragments 1 and 3 must each form a canonical stem.

recursive equation for each kind of split node. A special approach is used for Pseuoknots and will be discussed later in this section.

In this section another way of notation as in the previous section is used. All split types except E_2 cover contiguous fragments i.e., they cover a complete range of the sequence. At each node of the parse tree, we want to compute the optimal alignment for the fragment it represents to a fragment of the second sequence. These alignments are stored in a matrix $M[i,]$ for each node with the following meaning:

- A node represents a fragment F_a that covers the positions $\{i_1, \dots, j_1 - 1\}$, which is denoted by a range (i_1, j_1).
- A matrix entry $M[i,j]$ represents an optimal alignment of F_a to a fragment F_b with $\hat{F_b} = \{i, \dots, j-1\}$.
- $M[i, i+1]$ thus corresponds to an alignment to a single base at position i in S_2.
- $M[i,j]$ with $i \geq j$ denotes an alignment to an empty fragment.
- For example, $M[2,5]$ corresponds to an alignment to a fragment F_b with $\hat{F_b} = \{2,3,4\}$ and $M[1,2]$ denotes an alignment to an atomic fragment that spans the single position 1. $M[1,1]$ and $M[5,3]$ are examples for alignments to empty fragments.

The sequence positions are numbered starting with 0. Consequently, the last position is the sequence length minus one[4]. As a result, the cost for the final alignment for (S_1, P_1) and (S_2, P_2) can be found in the matrix entry $[0, \mathrm{m}]$ of the root node with range $(0, \mathrm{n})$.

In the following, we consider the computation of the matrix M for each kind of node separatly under the assumption that each node covers a range (i_1, j_1) of the sequence.

4.2.1 Base case at leaf nodes

The *base case* corresponds to the one given on page 12. It is used at the *leaf nodes* of the tree that cover a single position i_1 which means a single unpaired base to ensure arc-completeness. It is denoted by the range $(i_1, i_1 + 1)$.

$$M[i,j] = \min \begin{cases} M[i+1,j] + \mathrm{gap}_2(i) & \text{if} \quad (j-i) > 2 \\ M[i,j-1] + \mathrm{gap}_2(j-1) & \text{if} \quad (j-i) > 2 \\ \mathrm{basematch}(i, j-1) & \text{if} \quad (j-i) = 1 \\ \mathrm{gap}_1(i_1) & \text{if} \quad i >= j \end{cases}$$

The matrix is filled in decreasing order for index i and increasing order for j. Each entry can be computed in constant time. This leads to a runtime complexity of $O(n^2)$ since there are n^2 entries in the matrix.

[4]In the following, the two sequence lengths are denoted by n for the first and m for the second one.

So far, the second base case was the one for leaf nodes representing atomic fragments that consist of exactly the two end points p^L and p^R of an arc p which ensures arc-completeness. To compute an alignment for the two bases we need to find an alignment for the left endpoint p^L to a fragment [i',j'] of the second sequence and an alignment for p^R to a fragment [i",j"]. This means four indices and would lead to a space complexity of $O(m^4)$ which is not desirable. As a solution to this the corresponding alignment computation for the arc p is done at nodes with split type $1'21'$ and the leaf nodes for arcs are no longer used.

Before I continue I first want to introduce a new operator for computing the cost of an alignment to shorten the notation.

4.2.2 Arcmatch

- **arcmatch**$(l_1, r_1, l_2, r_2) = 0 + (\chi(l_1, r_1 + \chi(l_2, r_2)) \cdot \frac{w_{am}}{2})$ denotes the cost of an arc match or mismatch of two arcs (l_1, r_1) and (l_2, r_2) under the assumption that the alignment A aligns $S_1[l_1]$ to $S_2[l_2]$, $S_1[r_1]$ to $S_2[r_2]$ and there is a pair $(l_1, r_1) \in P_1$ and a pair $(l_2, r_2) \in P_2$.

4.2.3 Split type 1'21'

A node with split type $1'21'$ represents a part of the sequence that is closed by an arc. The range (i_1, j_1) sets p^L to i_1 and p^R to $j_1 - 1$. This leaves a range $(i_1 + 1, j_1 - 1)$ or alternatively a fragment $[i_1 + 1, j_1 - 2]$ for the inner part, which is the second fragment of the split. Here, we also need the result for the child node that represents the second fragment. Its matrix is denoted by M_2. The matrix is filled as follows:

- for empty fragments $[i, j]$ with $i \geq j$ there is only one possibility i.e, aligning p^L and p^R to gaps:

$$M[i, j] = w_r$$

- for non-empty fragments $[i, j]$ with $i < j$:[5]

$$M[i,j] = \min \begin{cases} M[i+1,j] + \text{gap}_2(i) & (I) \\ M[i,j-1] + \text{gap}_2(j-1) & (II) \\ M_2[i,j] + w_r & (III) \\ M_2[i+1,j] + \text{gap}_2(j-1)\text{basematch}(i_1,i) & (IV) \\ M_2[i,j-1] + \text{gap}_1(i) + \text{basematch}(j_1-1,j-1) & (V) \\ M_2[i+1,j-1] + \text{arcmatch}(i_1,j_1-1,i,j-1) \quad \text{if } (i,j-1) \in P_2 & (VI) \\ M_2[i+1,j-1] + \text{bm}(i_1,i) + \text{bm}(j_1-1,j-1) \quad \text{if } (i,j-1) \notin P_2 & (VII) \end{cases}$$

Each entry can be computed in constant time. The space complexity for the matrix is $O(m^2)$. Again the matrix is filled in decreasing order for index i and in increasing order for j.

[5] *bm* is used as abbreviation for *basematch*

4.2.4 Split type 2'1

Another possible split type is $2'1$ for a range (i_1, j_1). The size constraint implies that $S_1[i_1]$ is an unpaired base. In the parse tree, we do not need a node for the left child, the fragment $2'$, instead the possibilities for this alignment are considered in the node $2'1$. M_2 denotes the matrix of the child node for the fragment without size-constraint.

- for empty fragments $[i, j]$ with $i \geq j$ it is only possible to align $S[i_1]$ to a gap which means deleting it:

$$M[i, j] = w_d$$

- for non-empty fragments $[i, j]$ with $i < j$:

$$M[i, j] = \min \begin{cases} M_2[i, j] + \text{gap}_1(i_1) & (I) \\ M_2[i + 1, j] + \text{basematch}(i_1, i) & (II) \\ M[i + 1, j] + \text{gap}_2(i) & (III) \end{cases}$$

4.2.5 Split type 12'

The computation is quite similar for nodes (i_1, j_1) with split type $12'$. Here we also need only one child node for the unconstrained fragment and its matrix M_1. The fragment $\langle j - 1 \rangle$ denotes the unpaired base $j - 1$.

- for empty fragments $[i, j]$ with $i \geq j$ it is only possible to align $S[j_1 - 1]$ to a gap which means deleting it:

$$M[i, j] = w_d$$

- for non-empty fragments $[i, j]$ with $i < j$:

$$M[i, j] = \min \begin{cases} M_1[i, j] + \text{gap}_1(j_1 - 1) & (I) \\ M_1[i, j - 1] + \text{basematch}(j_1 - 1, j - 1) & (II) \\ M[i, j - 1] + \text{gap}_2(j - 1) & (III) \end{cases}$$

4.2.6 Split type 12

A node with range (i_1, j_1) and split type 12 has two child nodes for fragments 1 and 2. Their matrices are denoted by M_1 and M_2. This split type is unconstrained. All 12-splits from the second sequence have to be considered: This is done by iterating over all possible split points k with $i \leq k \leq j$ and using the according entries from M_1 and M_2, respectively. The boundaries for k are chosen to allow empty fragments 1 or 2 which corresponds to aligning one of the fragments completely to gaps.

$$M[i, j] = \min_k \{M_1[i, k] + M_2[k, j]\} \qquad \text{for } i \leq k \leq j$$

For each entry $M[i, j]$ and each k we look up one value of M_1 and one value of M_2, which takes constant time. In total, we do this k-times for each M$[i, j]$. If we assume the worst case with $i = 0$ and $j = m$ k can have m different values which leads to a linear runtime complexity for each entry.

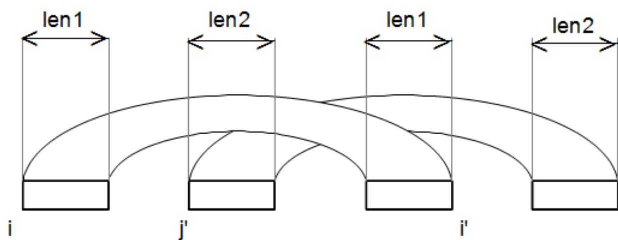

Figure 3: Schematic overview of a pseudoknot in S_2

4.2.7 Pseudoknots

The remaining two split types are E_1 and E_2 that deal with the decomposition of pseudoknots. In order to obtain a faster algorithm, we use a further restriction here: We limit the possible alignments to canonical ones that principally align pseudoknots in the first sequence only to other pseudoknots in the second sequence. If there is no pseudoknot available in S_2, then the pseudoknot from S_1 should completely be aligned to gaps.

The further decomposition of the two stems of a pseudoknot using E_2 takes place during the actual computation of the alignment to a pseudoknot fragment of the second sequence. Consequently, there are no nodes of type E_2 in the parse tree. A node representing a pseudoknot thereby has only three child nodes instead of five as implied by the split type $E_1 : 1^c23^c41^c53^c$.

The resulting alignment for a pseudoknot with range (i_1, j_1) is also stored in a matrix $M[i, j]$, and the matrices M_1, M_2 and M_3 of the three children are used. Additionally, two stem-matrices $\text{STEM}_{ii'}$ and $\text{STEM}_{jj'}$ are filled. They represent alignments of the first respectively the second stem to canonical stems in the second sequence and, during their filling procedure, the stems are decomposed in the way type E_2 proposes.

- For entries $M[i, j]$ with $i \geq j$ that represent empty fragments:

$$M[i, j] = \text{cost for aligning fragment } [i_1, j_1 - 1] \text{ completely to gaps}$$

- Entries $M[i, j]$ where (i, j) does not consist of a canonical pseudoknot in the second sequence should not be part of the final optimal alignment. They are therefore set to infinity since the algorithm chooses minimal values:

$$M[i, j] = \infty$$

For entries $M[i, j]$ with existing pseudoknot (i, j) in the second sequence the computation is more involved. Here the additonal matrices $\text{STEM}_{ii'}$ and $\text{STEM}_{jj'}$ are used. $\text{STEM}_{ii'}$ is used to compute the cost for the alignment of the first stem starting at position i_1 in S_1 to the first stem with outermoset base pair (i, i') in the second sequence. Likewise, $\text{STEM}_{jj'}$ denotes the cost to align the second stem with right endpoint $j - 1$ to the second stem $(j' - 1, j - 1)$.

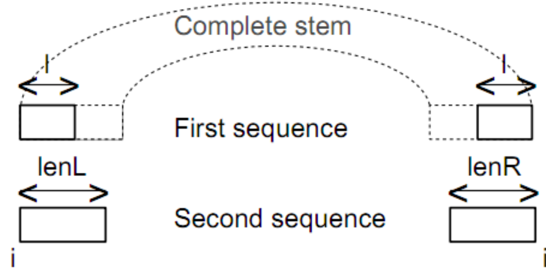

Figure 4: Meaning of l, lenL and lenR

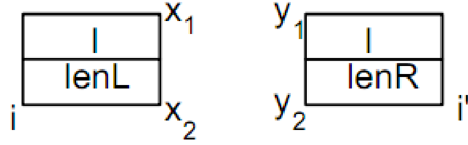

Figure 5: Positions of x_1, y_1, x_2 and y_2

More precisely, an entry $\mathrm{STEM}_{ii'}[lenL][lenR][l]$ denotes the cost to align the outermost l base pairs from the first stem of S_1 to $lenL$ positions from the left part and $lenR$ many position from the right part. This is illustrated in Figure 4. The variables x_1 and y_1 denote the positions of the current innermost base pair of the stem in S_1, x_2 and y_2 denote the ones from S_2 as shown in Figure 5.

For $\mathrm{STEM}_{jj'}$ this is done analogously. Since we want to align the complete stems of S_1, we are interested in the entries $[len1][len1][l_1]$ of $\mathrm{STEM}_{ii'}$ and $[len2][len2][l_2]$ of $\mathrm{STEM}_{jj'}$ where l_1 denotes the consumed length of the first stem in the pseudoknot of S_1 and l_2 the length of the second stem.

In the following recursion I use *bm* twice instead of *basematch* to reduce the length of the second line, the meaning is the same. For $\mathrm{STEM}_{ii'}$ it holds that $0 \leq l \leq l_1$, $0 \leq lenL \leq len1$ and $0 \leq lenR \leq len1$. The first entry $[0][0][0] = 0$ is only used for completition to avoid many exceptions for the marginal values and does not represent an actual part of the alignment, since this would mean aligning zero bases from the stem in S_1 and zero bases from S_2. During the traceback this is taken into consideration.

$$\mathrm{STEM}_{ii'}[lenL][lenR][l] =$$

$$\min \begin{cases} \mathrm{STEM}_{ii'}[lenL-1][lenR-1][l-1] + \mathrm{arcmatch}(x_1, y_1, x_2, y_2) & \text{if } (x_2, y_2) \in P_2 \\ \mathrm{STEM}_{ii'}[lenL-1][lenR-1][l-1] + \mathrm{bm}(x_1, x_2) + \mathrm{bm}(y_1, y_2) & \text{if } (x_2, y_2) \notin P_2 \\ \mathrm{STEM}_{ii'}[lenL][lenR][l-1] + w_r \\ \mathrm{STEM}_{ii'}[lenL-1][lenR][l-1] + \mathrm{basematch}(x_1, x_2) + \frac{w_r}{2} \\ \mathrm{STEM}_{ii'}[lenL][lenR-1][l-1] + \mathrm{basematch}(y_1, y_2) + \frac{w_r}{2} \\ \mathrm{STEM}_{ii'}[lenL-1][lenR][l] + \mathrm{gap}_2(x_2) \\ \mathrm{STEM}_{ii'}[lenL][lenR-1][l] + \mathrm{gap}_2(y_2) \end{cases}$$

The entries are filled in increasing order for l. In each step, only entries for l and $l-1$ are needed. It is thus possible to use $l \mod 2$ for the third index instead of l. This reduces the needed space to quadratic size.

Unfortunately, this trick cannot be used for the traceback. Here we start at a certain entry of a stem matrix and follow the recursion backwards. This is the reason why we need the entries for all possible values of l here. On the other hand, the maximal value of l is bounded by the maximal stem length among all pseudoknots that occur in the first sequence. Since there has to be the same number of bases to form the base pairs we can conclude that $l < n/2$. Consequently, the space complexity for this node is $O(m \cdot m \cdot n) = O(m^2 \cdot n)$

The entry $M[i,j]$ is then combined using the three children matrices. The minimal value among all possible $len1$ and $len2$ is chosen:

$$\begin{aligned} M[i,j] &= \text{STEM}_{ii'}[len1][len1][l_1] + \\ &\quad \text{STEM}_{jj'}[len2][len2][l_2] + \\ &\quad M_1[i + len1, j'] + \\ &\quad M_2[j' + len2, i' - len1 + 1] + \\ &\quad M_3[i' + 1, j - len2] \end{aligned}$$

The matrix M is filled for m^2 entries. The most complex computation is done for entries that represent an alignment to a pseudoknot in S_2. All possible values for $len1$ and $len2$ have to be considered which means $O(m^2)$ for each entry. This is $O(m^4)$ for all entries of M. The computation of each $\text{STEM}_{ii'}$ matrix costs $O(l^3)$ with l being the maximal length of a pseudoknot stem in the input. Consequently, the computation of all $\text{STEM}_{ii'}$ costs $O(m^2 l^3)$. For both matrices together, this leads to $O(m^2 + l^3)$ for each entry and $O(m^2 \cdot (m^2 + l^3))$ for all of them.

4.2.8 Complexity

For all split types the computation can be done in $O(m^2)$ time. The only exception is the computation at pseudoknots. In total, the parse tree can consist of $O(n)$ nodes. If we assume the worst case, which means that each node is a pseudoknot, this leads to an over-all complexity of $O(n \cdot m^2(m^2 + l^3))$. For a constant length l of a canonical pseudoknot stem it results in $O(n \cdot m^4)$. This is a linear increase compared to the $O(n^4)$ time complexity of the prediction algorithm by Reeder&Giegerich for this restricted class of pseudoknots and a sequence length n [3].

The space complexity is $O(m^2)$ for all nodes except for pseudoknots, which require additional $O(l^3)$ for precomputing the stems. Assuming again that l is constant, in the worst case, this leads to a space complexity of $O(n \cdot m^2)$. This is only a linear increase compared to the space complexity of the prediction algorithm by Reeder&Giegerich [3] which is $O(n^2)$.

4.3 A few details about the implementation

The algorithm has been implemented in C++. The structure comprises classes for sequences, the parse tree, alignments, a general node class and derived classes for the different split types. Methods for filling the matrices and doing the traceback are part of the node classes. This final class structure is the result of a number of modifications and improvements. Especially the recursions and the marginal values for matrix entries were the reason for many changes.

It takes as input two dotbracket files representing the two sequences. In a first step, it creates sequence objects for them and the second step is building the parse tree for the first sequence. Next the matrices denoting the alignment costs for all nodes are filled. Finally, the resulting alignment is reconstructed by the traceback that step by step composes the final alignment out of smaller parts for the fragments.

4.4 Results

In order to gain an insight if this tailored algorithm, denoted with $PKalignRG$ in the following, really runs faster than the general implementation, denoted with $PKalign$ [2], a few test runs have been made on a Intel Core Duo T5800 processor with 2.0GHz. Three example sequences were chosen from the Antizyme FSE family which were slightly modified to make them canonical. The following table shows the measured time : In all cases, $PKalignRG$ was clearly faster than $PKalign$.

First input sequence	Second input sequence	Time PKalignRG	Time PKalign
AF291576.1/215-270	BC056833.1/241-299	0.016s	0.640s
BC056833.1/241-299	AF291576.1/215-270	0.012s	0.620s
AF291576.1/215-270	AC004152.1/16641-16699	0.008s	0.672s
AC004152.1/16641-16699	AF291576.1/215-270	0.008s	0.636s
BC056833.1/241-299	AC004152.1/16641-16699	0.004s	0.640s
AC004152.1/16641-16699	BC056833.1/241-299	0.012s	0.704s

Danksagung. Abschließend möchte ich mich noch bei Mathias Möhl bedanken, der mir bei allen Fragen schnell und zuverlässig weitergeholfen hat.

References

[1] Möhl, M., Will, S., Backofen, R.: Lifting Prediction to Alignment of RNA Pseudoknots. 2009

[2] Möhl, M.: Dynamic Programming Based RNA Pseudoknot Alignment. Dissertation, University of Saarland. 2009

[3] Reeder, J., Giegerich, R.: Design, implementation and evaluation of a practical pseudoknot folding algorithm based on thermodynamics. BMC Bioinformatics 5 (2004) 104

[4] Möhl, M.: Script for the lecture RNA Bioinformatics. University of Freiburg. Winter semester 2010/2011.

[5] Couzin, J.: Breakthrough of the year. Small RNAs make big splash. Science 298(5602) (2002) 2296–7

[6] Staple, D.W., Butcher, S.E.: Pseudoknots: RNA structures with diverse functions.PLoS Biol 3(6) (2005) e213

[7] Lyngso, R.B., Pedersen, C.N.S.: Pseudoknots in RNA secondary structures. In: Proc. of the Fourth Annual International Conferences on Comptutational Molecular Biology (RECOMB00), ACM Press (2000) BRICS Report Series RS-00-1.

[8] Jiang, T., Lin, G., Ma, B., Zhang, K.: A general edit distance between RNA structures. Journal of Computational Biology 9(2) (2002) 371–88