Université Paris-Sud

Master Bioinformatique et Biostatistiques (BIBS)

Rapport de stage

Laboratoire d'Informatique de Polytechnique (LIX)

Non-ambiguous tree alignment for homology detection and common pattern finding in ribonucleic acid 3D structure

Author:
Pauline Pommeret

Supervisors:
Mireille Régnier
Yann Ponty

20th May 2016

Part I
Preamble

List of Figures

List of Tables

List of definitions[show=definition]

Acronyms

AAS arc-annotated sequence

CSV Comma-Separated Values

DP dynamic programming

dps dot-parenthesis sequence

dsRNA double-stranded RNA

iff if and only if

LCP largest common set point

mRNA messenger RNA

PDB Protein Data Bank

RMSD root-mean-square deviation of atomic positions

RNA ribonucleic acid

rRNA ribosomal RNA

tRNA transfer RNA

Contents

1 Introduction

During the course of the 20$^{\text{th}}$ century, the structural biology community mainly focused on the study of the structure of proteins which lead to a *protein-centric* view of both cellular and molecular biology.

However, the study of RNA structure is of growing importance thanks to the discovery of the key roles RNA play in cellular function: they perform a wide range of functions in biological systems, from catalysing the peptidyltransferase reactions leading to the formation of the peptide bond in the ribosomal unit, to regulating transcriptional gene expression through riboswitches.

While the conservation of the sequence is of primary importance for a protein to carry their function, it appeared that this dogma could not be applied to RNA. Indeed, the factors of great importance for an active RNA seem to be their secondary and tertiary structure.

Since the comparison of the 3D shape of RNA is the end-goal, one is ought to think that a geometrical approach to the problem would do the trick. Unfortunately, as we will explain later, is of no use in the context of structures with a high number of coordinates i. e. atoms.

This is why the community focused on the alignment of secondary structures of RNA using different paradigm: sequence alignment and edition, arc-annotated sequences comparison and edition or correctly bracketed edition.

Though those approach helped learn a lot about RNA structure and design a classification of RNA families, they did not reflect the *shape* of the RNA molecules. To mitigate this problem in order to find common *structural* patterns and to study the evolution of structural RNA families, the tree representation of the secondary structure was introduced and the community started to focus on *tree alignment* as a way to compare two RNA secondary structure.

In this work, we focused on the implementation of an unambiguous tree alignment dynamic programming scheme whose scoring scheme take into account primary structure (the sequence), secondary structure (base pairing and their interaction type) and tertiary structure (through dihedral angles). If the use of those 3 levels of structural information already exist in the litterature [**Ferre2007**] it used sequence alignment and not tree alignment which represents the real structure of RNA better.

The particularity of this tree alignment dynamic programming is that it is at the same time complete and unambiguous which allows the exploration and sampling of the space of all (both optimal and sub-optimal) tree alignments, as opposed to other ambiguous dynamic programming scheme that cannot be used with existing frameworks.

This alignment depends on a scoring scheme that needs to integrate all 3 levels of structure of an RNA which can be extracted or produced from its PDB file. The catch is that `Biopython` does not support RNA-specific features such as producing a representation of a secondary structure or computing the dihedral angles of a molecule so we developped a `python` package that could be integrated into `Biopython`

2 RNA representation and alignment

2.1 Limitations of the *squiggle-plot* representations

The *squiggle-plot* representation, as seen on Figure 10 is at the same time the most popular representation of RNA secondary structure and the least useful for structure comparison.

If the graphical visualization is a helpful step, it is produced by algorithms that tend to be designed to enhance the aesthetic of the representation as well as to avoid overlapping of the substructures. Moreover, they enforce conventions such as the circular representation of $5'$ and $3'$ extremities.

Anyhow, the squiggle-plot representation is not a suitable entry for a computer-based structure comparison as explained in RNA secondary structure comparison review [**Schirmer2014**].

2.2 *Dot-parenthesis* representation

The *dot-parenthesis* representation (see Figure 15), as used in VARNA [**Hofacker1994**], is a good compromise between readability and convenience.

Each structure of length n is represented by a sequence of characters that are parentheses or dots. Every base pair $(i, j), i < j$ is represented by a pair of parenthesis such as:
— the i-th position's character is an open parenthesis, `(`;
— the j-th position's character in a closing parenthesis, `)`;

Each single base is represented by a dot.

```
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17
G A G U A C A A U  A  U  G  U  A  C  C  G
. . ( ( ( ( . . .  .  .  )  )  )  )  .  .
```

Figure 1: Dot-parenthesis representation of a structure where bases $(3, 15)$, $(4, 14)$, $(4, 13)$ and $(6, 12)$ were detected as forming an interaction

If the dot-parenthesis representation has many benefits, it is not a suitable representation for structure comparison. Consider Figure 16: the string alignment is clearly optimal but it is not compatible with the structure.

```
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23
( ( ( . . . ) ) ) .  .  .  (  (  (  (  .  .  .  )  )  )  )
( ( ( . . . . . . . . . .  (  (  .  .  .  .  )  )  )  )  )
```

Figure 2: Optimal alignment of two dot-parenthesis sequences

Therefore, the dot-parenthesis is not a suitable representation for structural alignment, but it allows to create a Tree representation that is well-designed to represent structural alignments.

2.3 Tree representation and alignment

The tree structure of an RNA secondary structure can be deduced from its (correctly parenthesised) dps using a recursive algorithm such as Algorithm 5.

As trees are defined, one then needs to define tree alignment.

Notations. Let S and T be two rooted ordered trees with labeled vertices:
— V_S (resp. V_T): set of vertex of S (resp. T)
— $L(c)$: the label of any vertex $c \in V_S \cup V_T$
— $x \prec y$: the vertex $x \in V_T$ is the ancestor of the vertex $y \in V_T$
— $x < y$: the vertex x is visited before y in preorder traversal of T
— $p(x)$: the parent of a non-root vertex x
— T_x: subtree of T rooted in x
— $\sigma = (a, b)$: pair of vertex that match, one from S ($\sigma_S = a$) and one from T

Definition 2.1 (Tree alignment)**.** An alignment of two rooted ordered trees S and T is a set $\mathcal{A} \subset \Sigma_m = V_S \times V_T$ that satisfies the following properties:

1. $\forall a \in V_S$, there exists at most one $\sigma \in \mathcal{A}$ such as $\sigma_S = a$
2. $\forall b \in V_T$, there exists at most one $\sigma \in \mathcal{A}$ such as $\sigma_T = b$
3. for every $\sigma, \tau \in \mathcal{A}, \sigma_S \prec \tau_S$ if and only if $\sigma_T \prec \tau_T$ (ancestrality condition)
4. for every $\sigma, \tau \in \mathcal{A}, \sigma_S < \tau_S$ if and only if $\sigma_T < \tau_T$ (order condition)

Algorithm 1: Obtaining tree structure from dps (see `dps::parse` for implementation)

```
Function parseDPS(S, i)
    Data: S                                   /* string representing the sequence */
    Data: i                                   /* integer, position in the sequence */
    Result: T                                 /* tree representation of the structure */
begin
    start ← i                  /* Keep track of the starting index for recursive calls */
    root ← Tree()                             /* Root of the tree for the current call */
    while i < len(S) do
        i++
        switch S[i − 1] do
            case '.' do
                /* Leaf to be added to the current root                              */
                position ← {i, i}
                tree ← Tree(position)
                root.append(tree)
            case '(' do
                /* Beginning of a node:  recursive call                              */
                tree ← parseDPs(S, i)
                root.append(tree)
            case ')' do
                /* End of recursive call:  start and stop indexes acquired           */
                position ← {start, i}
                root.set_position(position)
                return root
            otherwise do
                /* This should never happen thanks to Algorithm 8                    */
                throw "Unauthorized character"
            end
        end
    end
    return root
end
```

The overall advantage of tree alignment over tree edit is that no node goes unnoticed and all nodes are involved in a relationship in the common supertree.

In other words a *tree alignment* is a (partial) mapping between the nodes of two trees that respects both the ancestrality and kindship relationships.

A tree alignment can be thought of as a *supertree* whose nodes represent either a match or an insertion or a deletion. The structure of the supertree is such that the topology of the entry trees can be deduce back as it can be seen on Figure 17.

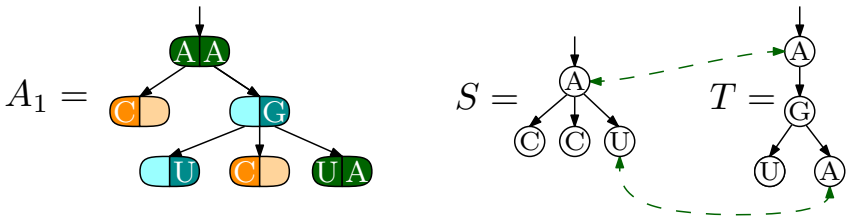


Figure 3: A_1 represents the supertree of the alignment of tree S with tree T

The purpose of tree alignment is, given S and T two trees and a cost for each operations (match, insertion, deletion), to compute the most parcimonious supertree of S and T.

Tree alignment is used to compare the structure of two different sequences, as described in the following subsection.

2.4 Original algorithm to align ordered trees

In 1995, JIANG *et al.* proposed tree alignment as an alternative to tree edit to measure the similarity between two trees [**Jiang1995**].

Notations. As introduced in [**Jiang1995**]:
- $D(F_1, F_2)$ is the alignment distance between 2 random forests F_1 and F_2
- θ is the empty tree
- λ is a space
- $\mu(a, b)$ is the score of opposing letters a and b (assumed to satisfy the triangle inequality)
- $l_i[j]$ is the label of node j in tree T_i
- $T_i[j]$ is the subtree of T_i rooted at node j
- if i is a node of T_1 and the degree of i is m_i, the children of i are $i_1, \ldots, i_{m_i}$
- $F_1[i_s, i_r]$ is the forest consisting of the subtrees $T_1[i_s] \ldots T_1[i_r]$
- $F_1[i_1, i_{m_i}]$ is noted $F_1[i]$

2.4.1 Algorithm

It is clear then that to compute the alignment distance between two trees T_1 and T_2 it is required to align $F_1[i]$ with each subforest of $F_2[j]$ and conversely.

Let's introduce a first procedure that allows to do such a thing.

Algorithm 2: Computes the alignment distances between two subforests [**Jiang1995**]

Procedure *ComputeAlignmentDistanceBetweenForest*

```
/* Computes the alignment distances of all alignments between two subforests,
assuming that all D(F1[ik], F2[jp, jq]), 1 ≤ k ≤ m1, 1 ≤ p ≤ q ≤ nj are known (and
conversely). */
```

Data:
$F_1[i_s, i_p], F_2[j_t, j_q]$ /* subforests */

Result:
$\{D(F_1[i_s, i_p], F_2[j_t, j_q]) \mid s \leq p \leq m_i, t \leq q \leq n_j\}$, for s and t fixed.

begin
- **for** $p = [s \ldots m_i]$ **do**
  - $D(F_1[i_s, i_p], F_2[j_t, j_{t-1}] := D(F_1[i_s, i_{p-1}], F_2[j_t, j_{t-1}]) + D(T_1[i_p], \theta)$
- **end**
- **for** $q = [t \ldots n_j]$ **do**
  - $D(F_1[i_s, i_{s-1}], F_2[j_t, j_q]) := D(F_1[i_s, i_{s-1}], F_2[j_t, j_{q-1}]) + D(\theta, T_2[j_q])$
- **end**
- **for** $p = [s \ldots m_i]$ **do**
  - **for** $q = [t \ldots n_j]$ **do**
    - Compute $D(F_1[i_s, i_p], F_2[j_t, j_q])$ as in Lemma in [**Jiang1995**]
  - **end**
- **end**

end

It is possible to access $D(F_1[i], F_2[j_s, j_t]), \forall 1 \leq s \leq t \leq n_j$ by calling n_j times the procedure *ComputeAlignmentDistanceBetweenForest* from Algorithm 6 on page 18.

The time complexity of this procedure is:

$$\mathcal{O}\left((m_i - s) \cdot (n_j - t) \cdot (m_i - s + n_j - t)\right) = \mathcal{O}\left(m_i \cdot n_j \cdot (m_i + n_j)\right) \tag{1}$$

On this basis, Algorithm 7 on page 19 can be used to compute $D(T_1, T_2)$:

Algorithm 3: Computes the alignment distances between two trees [**Jiang1995**]

Procedure *ComputeAlignmentDistanceBetweenTrees*

/* Computes the alignment distance between two trees, using Lemmas in [**Jiang1995**] and Algorithm 6 on page 18 */

Data:

T_1, T_2 /* trees */

Result:

$D(T_1[|T_1|], T_2[|T_2|])$

begin

- $D(\theta, \theta) := 0$ /* see lemma in [**Jiang1995**] */
- **for** $i = [1 \dots |T_1|]$ **do**
  - Initialize $D(T_1[i], \theta)$ and $D(F_1[i], \theta)$ /* see lemma in [**Jiang1995**] */
- **end**
- **for** $j = [1 \dots |T_2|]$ **do**
  - Initialize $D(\theta, T_2[j])$ and $D(\theta, F_2[j])$ /* see lemma in [**Jiang1995**] */
- **end**
- **for** $i = [1 \dots |T_1|]$ **do**
  - **for** $j = [1 \dots |T_2|]$ **do**
    - **for** $s = [1 \dots m_i]$ **do**
      - *ComputeAlignmentDistanceBetweenForest*$(F_1[i_s, i_{m_i}], F_2[j])$
    - **end**
    - **for** $t = [1 \dots n_j]$ **do**
      - *ComputeAlignmentDistanceBetweenForest*$(F_1[i], F_2[i_t, i_{n_j}])$
    - **end**
    - Compute $D(T_1[i], T_2[j])$ /* see lemma in [**Jiang1995**] */
  - **end**
- **end**

end

Neglecting the time complexity of the initiazation steps, the time complexity of Algorithm 7 on page 19 is:

$$\sum_{i=1}^{|T_1|}\sum_{j=1}^{|T_2|}\mathcal{O}\left(m_i \cdot n_j \cdot (m_i+n_j)^2\right) \leq \sum_{i=1}^{|T_1|}\sum_{j=1}^{|T_2|}\mathcal{O}\left(m_i \cdot n_j \cdot (\deg(T_1)+\deg(T_2))^2\right)$$

$$\leq \mathcal{O}\left((\deg(T_1)+\deg(T_2))^2 \sum_{i=1}^{|T_1|} m_i \sum_{j=1}^{|T_2|} n_j\right)$$

Therefore

$$\sum_{i=1}^{|T_1|}\sum_{j=1}^{|T_2|}\mathcal{O}\left(m_i \cdot n_j \cdot (m_i+n_j)^2\right) \leq \mathcal{O}\left(|T_1| \cdot |T_2| \cdot (\deg(T_1)+\deg(T_2))^2\right) \quad (2)$$

2.4.2 Dynamic programming scheme

The Jiang *et al.* algorithm can be rewritten as a DP scheme [**spire**]. The set-theoretical algebraic adapation of Algorithm 7 on page 19 consists of recurrences over $\mathbf{JS}[X, Y]$, the set of executions between forests X and Y.

Initialization:

$$\mathbf{JS}[\varnothing, \varnothing] = \{\varepsilon\} \tag{3}$$

Recursive step with one empty forest:

$$\mathbf{JS}[a(u) \circ X, \varnothing] = \{(a, -)\} \times \mathbf{JS}[u, \varnothing] \times \mathbf{JS}[X, \varnothing] \tag{4}$$

$$\mathbf{JS}[\varnothing, b(v) \circ Y] = \{(-, b)\} \times \mathbf{JS}[\varnothing, v] \times \mathbf{JS}[\varnothing, Y] \tag{5}$$

Recursive step with no empty forest:

$$\mathbf{JS}[a(u) \circ X, b(v) \circ Y] = \bigcup \begin{cases} \{(a, b)\} \times \mathbf{JS}[u, v] \times \mathbf{JS}[X, Y] & \text{(6a)} \\ \bigcup_{Y' \circ Y'' = b(v) \circ Y} \{(a, -)\} \times \mathbf{JS}[u, Y'] \times \mathbf{JS}[X, Y''] & \text{(6b)} \\ \bigcup_{X' \circ X'' = a(u) \circ X} \{(-, b)\} \times \mathbf{JS}[X', b] \times \mathbf{JS}[X'', Y] & \text{(6c)} \end{cases}$$

Figure 4: Original DP scheme [**Jiang1995**]

The DP scheme described in Figure 18 is complete (by induction using the fact that equations 15a, 15b and 15c consider all 3 cases when aligning 2 forests) but ambiguous, for 2 main reasons.

The first reason is that if the alignment between 2 forests is empty, i. e. contains no match, the sequence of insertions and deletions created by equations 15b, 15c, 13 and 14 can be shuffled around and still represent the same (empty) alignment, as seen on Figure 19.

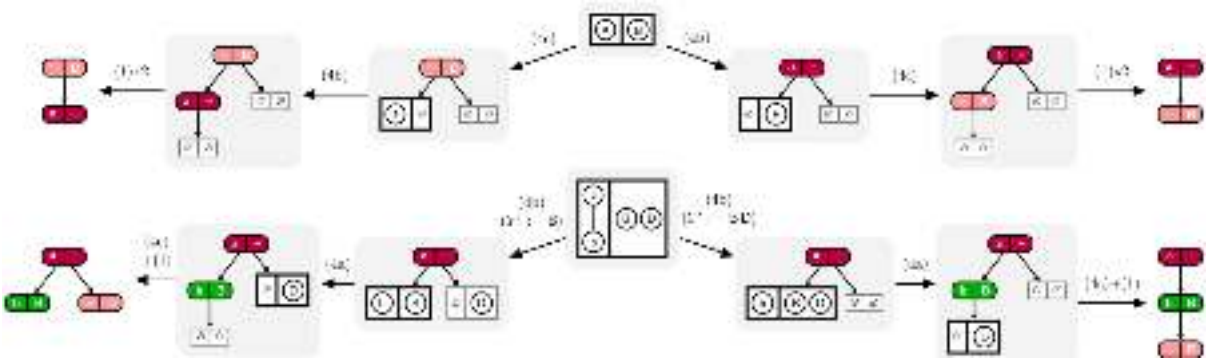


Figure 5: Two examples of ambiguity. Top: an empty alignment can be obtained in multiple ways by arbitrarily interleaving insertions and deletions. Bottom: While aligning two forests, inserting the first root a of the left forest leads to a decomposition of the right forest into two parts, recursively aligned to the children of a and the rest of the left forest respectively. Unfortunately, multiple decompositions may still yield the same alignment,leading to a more intricate level of ambiguity. Figure taken from [**spire**]

The second reason is the partitioning of forests in 2 subforests that occur in Equations 15b and 15c. If the last tree t of Y' is not matched, the same alignment can be obtained with an alternative decomposition $Y = Y_1 \circ Y_2$ in which t is the first tree of Y_2, as seen on Figure 19.

The ambiguity of the DP scheme is a huge draw-back since it prevents the use of numerous theoretical and practical optimization and explorational frameworks [**Giegerich2000**]. Those frameworks provide many advantages:

— exploration of the solutional space (optimal and suboptimal solutions)

— enumeration of all optimal solutions [**Bansal2013**]
— random sampling of solutions under a probability distribution influenced by their cost [**Ding2003**], [**Ponty2011**]

For problems that can be solved by using DP, these goals can be achieved through algebraic substitutions and transforms of the DP scheme.

Part II
Prerequisites and state of the art

3 Overview of ribonucleic acid structures and objectives

This section aims, firstly, at introducing basic notions and definitions regarding RNA structure. Furthermore, it will introduce problems and observations that lead to pinpoint specific issues whose study implied this internship.

Definition 3.1 (3D structure of a RNA)**.** The 3D structure of an RNA is defined by the position of its atoms in a 3D space.

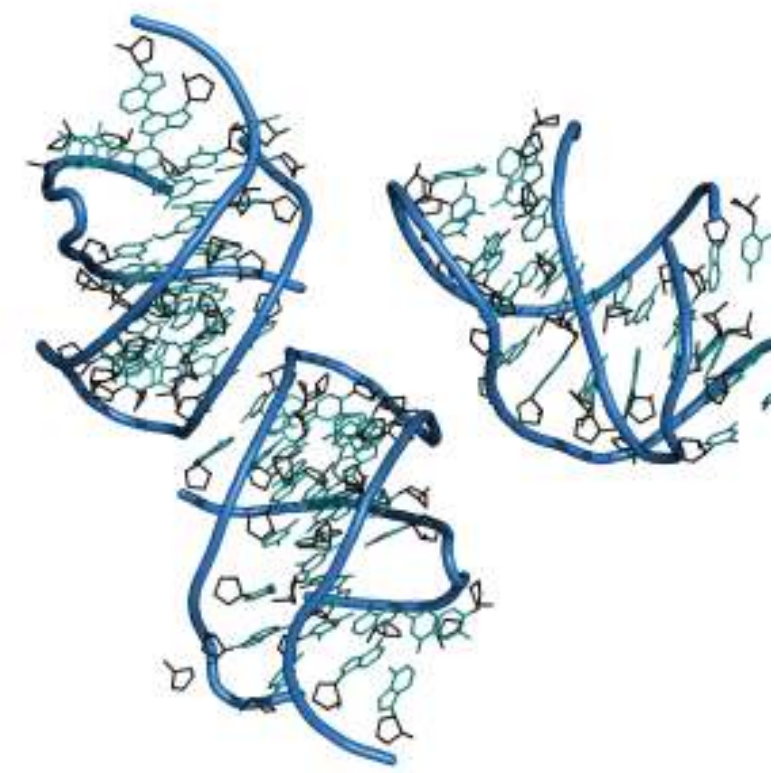


Figure 6: Representation of the 3D structure of 3FU2 (riboswitch) created by PyMol[**PyMOL**] from the atomic coordinates

Definition 3.2 (Riboswitch)**.** A riboswitch is a structure formed in a messenger RNA (mRNA) that regulates gene expression, usually by binding to a small molecule, which results in a change of conformation of the protein encoded by the mRNA. An mRNA that contains a riboswitch is involved in the regulation of its activity, depending on the intra-cellular concentration of its effector mo-

lecule [**VitreschakAG2004**] or on environmental parameters such as temperature [**Nocker2001**].

The comparison of RNA 3D structures leads to the highlight of both structural and functional domains within the structures as well as the detection of homologies in a much more precise and specific terms than sequence or secondary structure comparisons.

3.1 Definition of the 3D alignment of RNA problem

Formally, the problem of the 3D alignment of two RNAs molecules has been described by Dror *et al* [**Dror2005**].

Problem 1. Let $\varepsilon > 0$.
Data:
— two RNAs structures represented by their atomic coordinates in the PDB file format from which are extracted the phosphate atoms coordinates to create two sets of points $A = \{a_i\}$ and $B = \{b_i\}$
— distance function
Output:
— a 3D transformation T
— $A' = \{a'_i\}_{i=1}^{k} \subseteq A$ and $B' = \{b'_i\}_{i=1}^{k} \subseteq B$ of maximum cardinality (k) such as the maximum distance between A' and $T(B')$ is at most ε.

Problem 1 is also known as the largest common set point (LCP) problem [**Alt1996**] and has been widely studied. However, there exist neither an exact nor an approximate algorithm for the root-mean-square deviation of atomic positions (RMSD) metric. Anyhow, the RMSD metric would not have been the most relevant metric to use due to the high flexibility of RNA molecules.

Another metric that can be used is the bottleneck matching metric for which an exact algorith exist. This algorithm time complexity is in $\mathcal{O}(n^{32.5})$ where $n = \max(|A|, |B|) \in [\![3; 2930]\!]$ [**Ambuhl2000**]; which is quite impractical.

Therefore it is not possible to use the full atomic coordinates information when comparing 3D structures and one need another approach than the LCP problem to compare the 3D structure of two RNAs. For instance, one could compute the dihedral angles of the RNA molecules and use those angles to include a 3D perspective into an alignement method.

3.2 RNA secondary structure definition

RNA secondary structures are the result of bases pairing (called arcs) within an RNA thanks to hydrogen bonds and can form five main types of patterns. These arcs can be of two forms.

Definition 3.3 (*nested* and *independent* arcs [**Jiang2002**])**.** Lets consider two arcs $(i_1, i_2), (i_3, i_4)$ in a RNA structure.

- They are *independent* if they satisfy one of the following inequalities

$$\begin{aligned} &i_1 < i_2 < i_3 < i_4 \\ &i_3 < i_4 < i_1 < i_2 \end{aligned} \tag{7}$$

- They are *nested* if they satisfy one of the following inequalities

$$\begin{aligned} &i_1 < i_3 < i_4 < i_2 \\ &i_3 < i_1 < i_2 < i_4 \end{aligned} \tag{8}$$

Definition 3.4 (RNA secondary structure)**.** The secondary structure of an RNA is the largest set of nested and independent arcs that can be extracted from its arc-annotated sequence.

RNA external and internal base pairing can be predicted from the atomic coordinates. For instance the `rnaview` [**Yang2003**]program predicts both intra-chain and extra-chain interactions from the atomic coordinates given in PDB format.

In order to retrieve a secondary structure from the interactions predicted by `rnaview` [**Yang2003**], a planarization step is required. The method is used in the implementation describded Section 5 on page 21, taken from [**Ponty2006**] follows.

Definition 3.5 (General structure)**.** A *general structure* is a graph (V, A) where $V = [\![1; n]\!]$ (n is the length of the structure) and $A \subseteq \{(i, j) \in [\![1; n]\!]^2 | i < j\}$.

A *general structure* $S = (V, A)$ is included in $S' = (V, A')$ ($S \subseteq S'$) if and only if (iff) $A \subseteq A'$.

3.3 Planarization of RNA secondary structure

In this part, we introduce planarization problem for a RNA secondary structure. The idea behind this problem is, given a general structure S, to retreive the largest substructure that matches a planarity condition. The overall idea is to *split* a general structure into two smaller structures as explained on Figure 7. A formalization of this problem follows.

Problem 2 (Planirization of an RNA secondary structure)**.** Given a general structure $S = (V, A)$, return $S' = (V, A')$ such as:

1. inclusivity: $S' \subseteq S$
2. planarity:

$$\forall (i, j) \neq (i', j') \in A', (i \neq i') \wedge (j \neq j') \wedge \left([\![i; j]\!] \cap [\![i'; j']\!] = \left\{ \begin{array}{l} \varnothing \\ [\![i; j]\!] \\ [\![i'; j']\!] \end{array} \right. \right)$$

3. optimality: $\forall A'' \subseteq A$ such as (V, A'') planar, $|A''| \leq |A'|$,

Problem 2 is solved by Algorithm 4. An example of this algorithm output is shown on Figure 8.

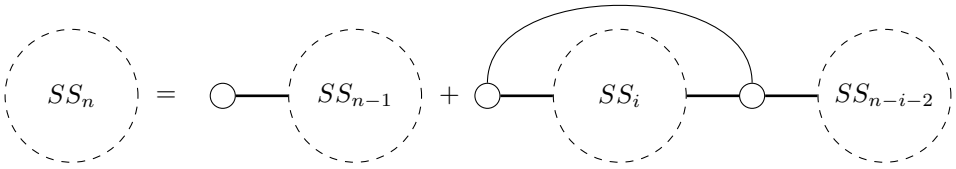


Figure 7: The possible decomposition of a general structure: either a structure of size $n - 1$ and an unpaired base or a structure of size i nested in an arc and followed by a structure of size $n - i - 2$.

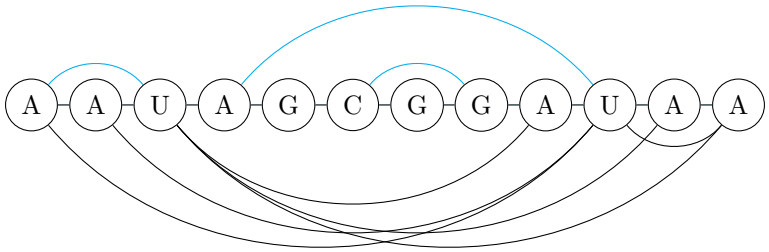


Figure 8: In blue, the secondary structure of artificial RNA molecule AAUAGCGGAUAA, in black the arcs removed by the planarization method described in Algorithm 4

Algorithm 4: Planarization of an RNA secondary structure [**Ponty2006**] thanks to decomposition explained in Figure 7

Procedure *FillMatrix(θ, n, A)*
  Data: A /* set of detected arcs */
  Data: n /* sequence length */
  Data: θ /* dynamic programming matrix */
  /* Fills the matrix with the number of arcs in subsequences */
begin
  $\theta_{\varnothing} \leftarrow 0$
  for $m \in [\![0; n-1]\!]$ **do**
    for $i \in [\![1; n-m]\!]$ **do**
      $j \leftarrow i + m$
      $\theta_{[\![i;j]\!]} \leftarrow \theta_{[\![i+1;j]\!]}$
      for $(i, \alpha) \in A, \alpha \leq j$ **do**
        $\theta_{[\![i;j]\!]} \leftarrow \max\left(1 + \theta_{[\![i+1;\alpha-1]\!]} + \theta_{[\![\alpha+1;j]\!]}; \theta_{[\![i;j]\!]}\right)$
      end
    end
  end
end
Function *Traceback(θ, i, j, n)*
  Data: A /* set of detected arcs */
  Data: n /* sequence length */
  Data: θ /* dynamic programming matrix */
  Data: i, j /* start and stop index */
begin
  if $i \geq j$ **then**
    return $\varnothing$
  end
  if $\theta_{[\![i;j]\!]} = \theta_{[\![i+1;j]\!]}$ **then**
    return Traceback$(\theta, i+1, j, n)$
  end
  for $(i, \alpha) \in A, \alpha \leq j$ **do**
    if $\theta_{[\![i;j]\!]} = 1 + \theta_{[\![i+1;\alpha-1]\!]} + \theta_{[\![\alpha+1;j]\!]}$ **then**
      return $\{i, \alpha\} \cup$ Traceback$(\theta, i+1, \alpha-1, n) \cup$ Traceback$(\theta, \alpha+1, j, n)$
    end
  end
end

The most obvious consequence of the use of the planarization on the set of interactions of an RNA is the exclusion of pseudoknots, that is, the exclusion of one of the arcs that are crossing each other.

As it can be seen on Figure 9 the exclusion of pseudoknots can exclude a very small number of interactions, even within complex structures such as tRNAs, and does not seem that far of a stretch.

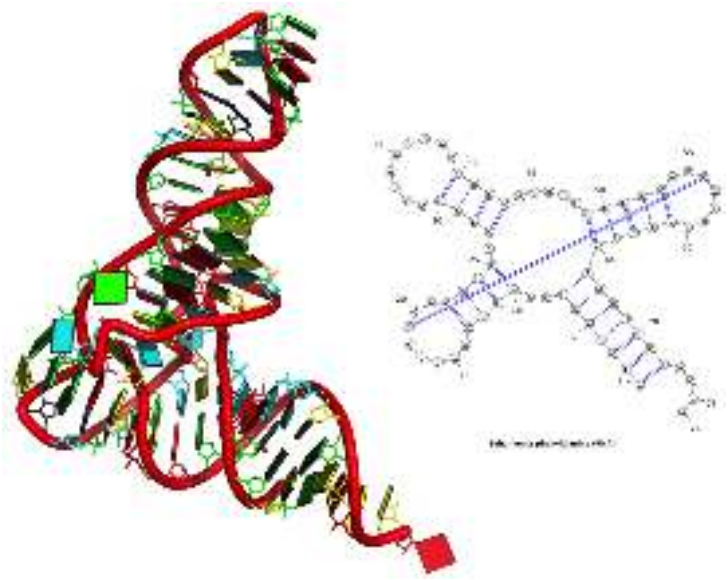


Figure 9: 3D structure of tRNA 1EHZ and its secondary structure with its single pseudoknot

Moreover, the exclusion of pseudoknots of the definition of secondary structures is a requirement to avoid facing NP-hard problems when aligning arc-annotated sequences (see Table 5 on page x).

3.4 RNA secondary structure patterns

Here we describe some common patterns that may be encountered in a RNA secondary structure.

Definition 3.6 (Stem)**.** A stem RNA secondary structure consists of double-stranded RNA (dsRNA), that is contiguous stacked base pairs (see Subfigure 10a below)

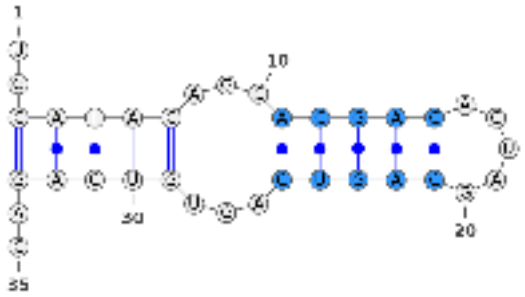
The second great type of pattern that can be found in RNA secondary structure is called a *loop* and consists of single stranded subsequences bounded by base pairs. There are different types of loop.

Definition 3.7 (Hairpin loop)**.** A *hairpin loop* is the most simple kind of loop. Every substructure consisting of a stem (Definition 3.6) and a loop are called hairpin loop, because of the resemblance with a hairpin when drawn (see Figure 10b).

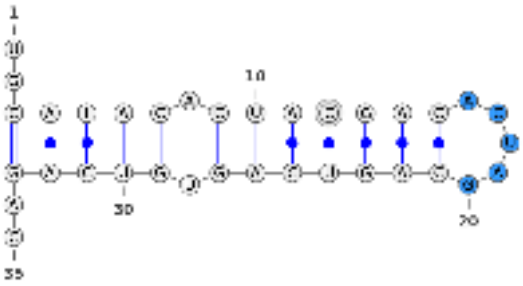
Definition 3.8 (Bulge)**.** Single stranded bases that occurs within a stem on a single side of the stem (Definition 3.6) as seen on Figure 10c.

Definition 3.9 (Interior loop)**.** When there are singled stranded bases on both sides of a stem (Definition 3.6), the structure is called an *interior loop*, as it can be seen in Figure 10d.

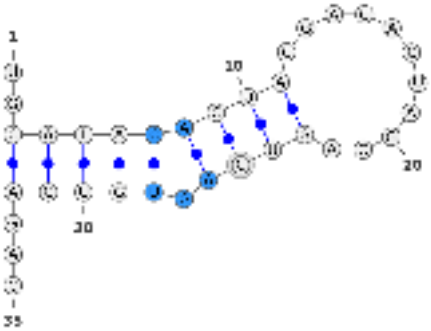
Definition 3.10 (Multi-branched loop)**.** If more than two stems emerge from one loop, such a loop is called a *multi-branched loop* (see Figure 10e).



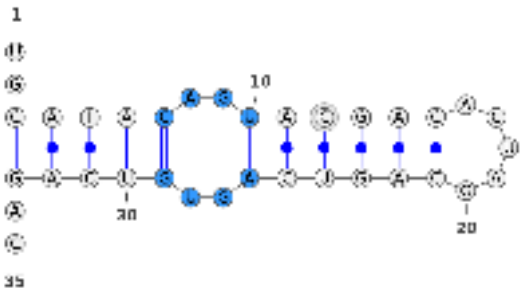
(a) Stem structure (in blue)



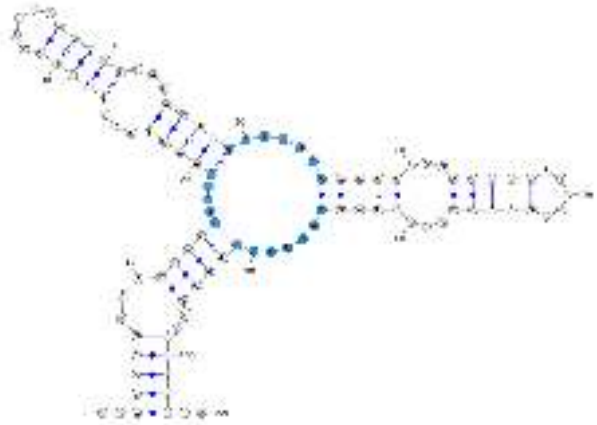
(b) Hairpin structure (in blue)



(c) Bulge structure (in blue)



(d) Interior-loop (in blue)



(e) Multi-branch-loop (in blue)

Figure 10: Example of the most commonly studied RNA secondary structures patterns (produced by VARNA [**Hofacker1994**])

Those patterns come in handy while describing an RNA 3D structure, for instance while describing the different functional subparts of the well-known tRNA 1EHZ, *Sacchromyces cerevisiae*'s phenylalanine tRNA, in Figure 11.

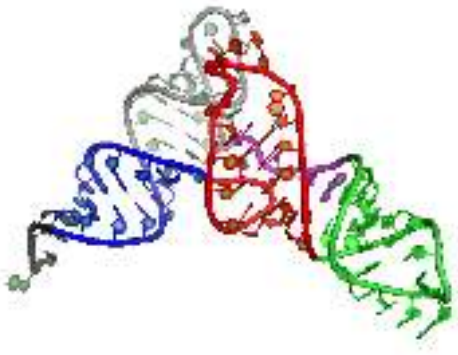


Figure 11: X-ray structure of tRNA 1EHZ: in green the anticodon hairpin loop (Definition 3.7), in blue the acceptor stem (Definition 3.6), in light gray the TPsiC-loop, in red the D-loop, in violet the variable loop (image produced by PyMol [**PyMOL**])

Those patterns, more than a useful tool to decribe RNA secondary structure, are used to produce classifications and hierachies of RNA molecules, those hierarchies being structural or functional.

Unfortunately, comparing secondary structures and those patterns is not sufficient to highlight the differences and similarities between two RNAs molecules.

For instance the mutation of a single base between two RNAs can lead to two profoundly different RNA secondary structures when the 3D structures seem to still be similar.

Besides, those representations are of no use when it comes to computational comparison.

3.5 Dihedral, torsion and pseudo-dihedrals angles

Definition 3.11 (Dihedral angle)**.** In chemistry, a dihedral angle is the angle between 2 planes that go through two sets of 3 atoms, those plane having 2 atoms is common. (see Figure 12)

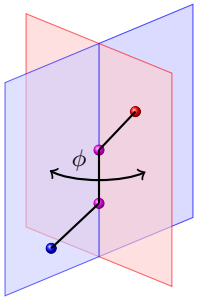


Figure 12: An example of dihedral angle ϕ formed by a two connected sets of 3 atoms.

Six dihedrals angles (α, β, γ, δ, ε and ζ) can be defined on the RNA backbone as well as a dihedral angle describing the rotation between the backbone and the base (χ). Those angles are shown on Figure 13.

Furthermore, two virtual angles η and θ were introduced by Olson [**Olson1975**]:

$$\eta = C_{4'}(i-1) - P(i) - C_{4'}(i) - P(i+1), \theta = P(i) - C_{4'}(i) - P(i+1) - C_{4'}(i+1) \tag{9}$$

The distribution of the dihedral angles in a set of 1088 RNA structures computed by a produced software (See Section 5 on page 21) can be found in Figure 28 and Figure 27 on page 26.

Those angles can be used to introduce 3D information in the scoring scheme of alignments without considering solving Problem 1, for instance by including the *matching* of dihedrals angles in the computation of the scoring matrix of the tree alignment (See Section 6 on page 28).

The use of dihedral angles to represent 3D information isn't a fallback since 3D alignment is NP-complete since they allow to take into consideration RNA intrinsic flexibility. Indeed, a local torsion or rotation would have a low cost in terms of dihedral angles while having a spectacular impact on the more typical RMSD metric.

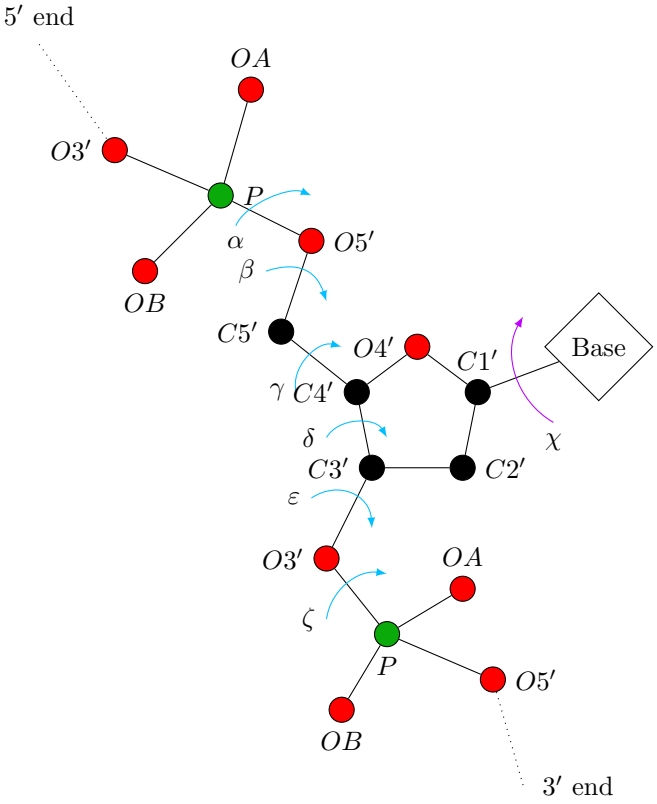


Figure 13: RNA backbone where the torsion angles are labeled on the central bond of the four atoms defining the dihedral angles. In blue (α, β, γ, δ and ζ) the backbone's dihedral angles and in violet (χ) the base's rotation angle (figure inspired by [**Neidle2007**]). In red the oxygen atoms, in green the phosphate atoms and in black the carbon atoms.

3.6 Interaction types and orientations

When refering to base pair interactions, two kinds of interactions come to mind: stacking and edge-to-edge interactions. Here, only the edge-to-edge interactions, mediated by hydrogen bonding will be discussed and looked upon.

If the canonical Watson-Crick base pairings are edge-to-edge interactions, they are far from being the only kind of edge-to-edge interactions that may occur. The classification and nomenclature by Leontis and Westhof [**Leontis2001**] defines 12 types of interactions, depending on the *edges* of the bases that interact and the orientation of the glycosidic bond, as shown on Figure 14.

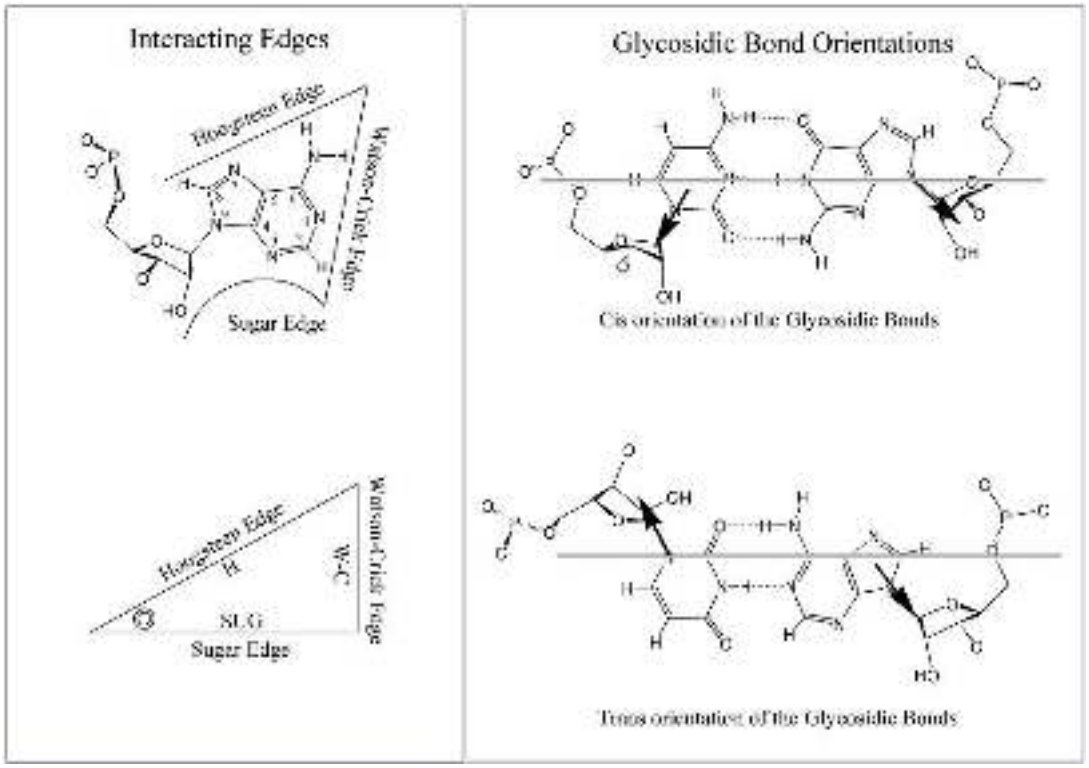


Figure 14: Left panel: interactiong edges for purines (here adenosine) and pyrimidines (here cytosine). Right panel: glycosidic bonds orientation defined relative to a line drawn parallel to and between the base-to-base hydrogen bonds in the case of 2 hydrogen bonds or in the middle hydrogen bond in case of 3. Figure taken from [**Leontis2001**]

As shown in Figure 14, RNA bases present 3 edges for interactions:

— the Watson-Crick edge
— the Hoogsteen edge (purines) or CH edge (pyrimidines), both will be called Hoogsteen edges from now on
— the Sugar edge (involves the 2′ hydroxyl group

Each edge can potentially interact in a plane with any other edge, in both *cis* and *trans* orientations which accounts for 12 distinct families of edge-to-edge interactions. For 11 of those 12 families, X-ray structures illustrating the family have been found, the exception being the *cis* Hoogsteen/Hoogsteen family.

4 RNA representation and alignment

4.1 Limitations of the *squiggle-plot* representations

The *squiggle-plot* representation, as seen on Figure 10 is at the same time the most popular representation of RNA secondary structure and the least useful for structure comparison.

If the graphical visualization is a helpful step, it is produced by algorithms that tend to be designed to enhance the aesthetic of the representation as well as to avoid overlapping of the substructures. Moreover, they enforce conventions such as the circular representation of 5′ and 3′ extremities.

Anyhow, the squiggle-plot representation is not a suitable entry for a computer-based structure comparison as explained in RNA secondary structure comparison review [**Schirmer2014**].

4.2 *Dot-parenthesis* representation

The *dot-parenthesis* representation (see Figure 15), as used in VARNA [**Hofacker1994**], is a good compromise between readability and convenience.

Each structure of length n is represented by a sequence of characters that are parentheses or dots. Every base pair $(i, j), i < j$ is represented by a pair of parenthesis such as:
— the i-th position's character is an open parenthesis, `(`;
— the j-th position's character in a closing parenthesis, `)`;

Each single base is represented by a dot.

```
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17
G A G U A C A A U  A  U  G  U  A  C  C  G
. . ( ( ( ( . . .  .  .  )  )  )  )  .  .
```

Figure 15: Dot-parenthesis representation of a structure where bases $(3, 15)$, $(4, 14)$, $(4, 13)$ and $(6, 12)$ were detected as forming an interaction

If the dot-parenthesis representation has many benefits, it is not a suitable representation for structure comparison. Consider Figure 16: the string alignment is clearly optimal but it is not compatible with the structure.

```
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23
( ( ( . . . ) ) ) .  .  .  (  (  (  (  .  .  .  )  )  )  )
( ( ( . . . . . . . . . .  (  (  .  .  .  .  )  )  )  )  )
```

Figure 16: Optimal alignment of two dot-parenthesis sequences

Therefore, the dot-parenthesis is not a suitable representation for structural alignment, but it allows to create a Tree representation that is well-designed to represent structural alignments.

4.3 Tree representation and alignment

The tree structure of an RNA secondary structure can be deduced from its (correctly parenthesised) dps using a recursive algorithm such as Algorithm 5.

As trees are defined, one then needs to define tree alignment.

Notations. Let S and T be two rooted ordered trees with labeled vertices:
— V_S (resp. V_T): set of vertex of S (resp. T)
— $L(c)$: the label of any vertex $c \in V_S \cup V_T$
— $x \prec y$: the vertex $x \in V_T$ is the ancestor of the vertex $y \in V_T$
— $x < y$: the vertex x is visited before y in preorder traversal of T
— $p(x)$: the parent of a non-root vertex x
— T_x: subtree of T rooted in x
— $\sigma = (a, b)$: pair of vertex that match, one from S ($\sigma_S = a$) and one from T

Definition 4.1 (Tree alignment)**.** An alignment of two rooted ordered trees S and T is a set $\mathcal{A} \subset \Sigma_m = V_S \times V_T$ that satisfies the following properties:

1. $\forall a \in V_S$, there exists at most one $\sigma \in \mathcal{A}$ such as $\sigma_S = a$
2. $\forall b \in V_T$, there exists at most one $\sigma \in \mathcal{A}$ such as $\sigma_T = b$
3. for every $\sigma, \tau \in \mathcal{A}, \sigma_S \prec \tau_S$ if and only if $\sigma_T \prec \tau_T$ (ancestrality condition)
4. for every $\sigma, \tau \in \mathcal{A}, \sigma_S < \tau_S$ if and only if $\sigma_T < \tau_T$ (order condition)

Algorithm 5: Obtaining tree structure from dps (see `dps::parse` for implementation)

```
Function parseDPS(S, i)
    Data: S                                   /* string representing the sequence */
    Data: i                                   /* integer, position in the sequence */
    Result: T                                 /* tree representation of the structure */
begin
    start ← i                  /* Keep track of the starting index for recursive calls */
    root ← Tree()                             /* Root of the tree for the current call */
    while i < len(S) do
        i++
        switch S[i − 1] do
            case '.' do
                /* Leaf to be added to the current root                              */
                position ← {i, i}
                tree ← Tree(position)
                root.append(tree)
            case '(' do
                /* Beginning of a node:  recursive call                              */
                tree ← parseDPS(S, i)
                root.append(tree)
            case ')' do
                /* End of recursive call:  start and stop indexes acquired           */
                position ← {start, i}
                root.set_position(position)
                return root
            otherwise do
                /* This should never happen thanks to Algorithm 8                    */
                throw "Unauthorized character"
            end
        end
    end
    return root
end
```

The overall advantage of tree alignment over tree edit is that no node goes unnoticed and all nodes are involved in a relationship in the common supertree.

In other words a *tree alignment* is a (partial) mapping between the nodes of two trees that respects both the ancestrality and kindship relationships.

A tree alignment can be thought of as a *supertree* whose nodes represent either a match or an insertion or a deletion. The structure of the supertree is such that the topology of the entry trees can be deduce back as it can be seen on Figure 17.

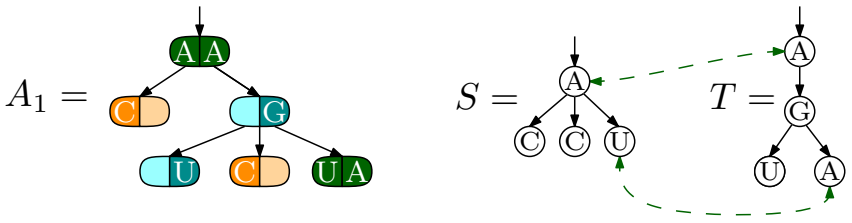


Figure 17: A_1 represents the supertree of the alignment of tree S with tree T

The purpose of tree alignment is, given S and T two trees and a cost for each operations (match, insertion, deletion), to compute the most parcimonious supertree of S and T.

Tree alignment is used to compare the structure of two different sequences, as described in the following subsection.

4.4 Original algorithm to align ordered trees

In 1995, JIANG *et al.* proposed tree alignment as an alternative to tree edit to measure the similarity between two trees [**Jiang1995**].

Notations. As introduced in [**Jiang1995**]:
- $D(F_1, F_2)$ is the alignment distance between 2 random forests F_1 and F_2
- θ is the empty tree
- λ is a space
- $\mu(a, b)$ is the score of opposing letters a and b (assumed to satisfy the triangle inequality)
- $l_i[j]$ is the label of node j in tree T_i
- $T_i[j]$ is the subtree of T_i rooted at node j
- if i is a node of T_1 and the degree of i is m_i, the children of i are $i_1, \ldots, i_{m_i}$
- $F_1[i_s, i_r]$ is the forest consisting of the subtrees $T_1[i_s] \ldots T_1[i_r]$
- $F_1[i_1, i_{m_i}]$ is noted $F_1[i]$

4.4.1 Algorithm

It is clear then that to compute the alignment distance between two trees T_1 and T_2 it is required to align $F_1[i]$ with each subforest of $F_2[j]$ and conversely.

Let's introduce a first procedure that allows to do such a thing.

Algorithm 6: Computes the alignment distances between two subforests [**Jiang1995**]

Procedure *ComputeAlignmentDistanceBetweenForest*

```
  /* Computes the alignment distances of all alignments between two subforests,
 assuming that all D(F1[ik], F2[jp, jq]), 1 ≤ k ≤ m1, 1 ≤ p ≤ q ≤ nj are known (and
 conversely).  */
```

Data:
$F_1[i_s, i_p], F_2[j_t, j_q]$ /* subforests */

Result:
$\{D(F_1[i_s, i_p], F_2[j_t, j_q]) \mid s \leq p \leq m_i, t \leq q \leq n_j\}$, for s and t fixed.

begin
- **for** $p = [s \ldots m_i]$ **do**
  - $D(F_1[i_s, i_p], F_2[j_t, j_{t-1}] := D(F_1[i_s, i_{p-1}], F_2[j_t, j_{t-1}]) + D(T_1[i_p], \theta)$
- **end**
- **for** $q = [t \ldots n_j]$ **do**
  - $D(F_1[i_s, i_{s-1}], F_2[j_t, j_q]) := D(F_1[i_s, i_{s-1}], F_2[j_t, j_{q-1}]) + D(\theta, T_2[j_q])$
- **end**
- **for** $p = [s \ldots m_i]$ **do**
  - **for** $q = [t \ldots n_j]$ **do**
    - Compute $D(F_1[i_s, i_p], F_2[j_t, j_q])$ as in Lemma in [**Jiang1995**]
  - **end**
- **end**

end

It is possible to access $D(F_1[i], F_2[j_s, j_t]), \forall 1 \leq s \leq t \leq n_j$ by calling n_j times the procedure *ComputeAlignmentDistanceBetweenForest* from Algorithm 6.

The time complexity of this procedure is:

$$\mathcal{O}\left((m_i - s) \cdot (n_j - t) \cdot (m_i - s + n_j - t)\right) = \mathcal{O}\left(m_i \cdot n_j \cdot (m_i + n_j)\right) \tag{10}$$

On this basis, Algorithm 7 on the next page can be used to compute $D(T_1, T_2)$:

Algorithm 7: Computes the alignment distances between two trees [**Jiang1995**]

Procedure *ComputeAlignmentDistanceBetweenTrees*

/* Computes the alignment distance between two trees, using Lemmas in [**Jiang1995**] and Algorithm 6 on the facing page */

Data:

T_1, T_2 /* trees */

Result:

$D(T_1[|T_1|], T_2[|T_2|])$

begin

- $D(\theta, \theta) := 0$ /* see lemma in [**Jiang1995**] */
- **for** $i = [1 \dots |T_1|]$ **do**
  - Initialize $D(T_1[i], \theta)$ and $D(F_1[i], \theta)$ /* see lemma in [**Jiang1995**] */
- **end**
- **for** $j = [1 \dots |T_2|]$ **do**
  - Initialize $D(\theta, T_2[j])$ and $D(\theta, F_2[j])$ /* see lemma in [**Jiang1995**] */
- **end**
- **for** $i = [1 \dots |T_1|]$ **do**
  - **for** $j = [1 \dots |T_2|]$ **do**
    - **for** $s = [1 \dots m_i]$ **do**
      - *ComputeAlignmentDistanceBetweenForest*$(F_1[i_s, i_{m_i}], F_2[j])$
    - **end**
    - **for** $t = [1 \dots n_j]$ **do**
      - *ComputeAlignmentDistanceBetweenForest*$(F_1[i], F_2[i_t, i_{n_j}])$
    - **end**
    - Compute $D(T_1[i], T_2[j])$ /* see lemma in [**Jiang1995**] */
  - **end**
- **end**

end

Neglecting the time complexity of the initiazation steps, the time complexity of Algorithm 7 is:

$$\sum_{i=1}^{|T_1|}\sum_{j=1}^{|T_2|} \mathcal{O}\left(m_i \cdot n_j \cdot (m_i + n_j)^2\right) \leq \sum_{i=1}^{|T_1|}\sum_{j=1}^{|T_2|} \mathcal{O}\left(m_i \cdot n_j \cdot (\deg(T_1) + \deg(T_2))^2\right)$$

$$\leq \mathcal{O}\left((\deg(T_1) + \deg(T_2))^2 \sum_{i=1}^{|T_1|} m_i \sum_{j=1}^{|T_2|} n_j\right)$$

Therefore

$$\sum_{i=1}^{|T_1|}\sum_{j=1}^{|T_2|} \mathcal{O}\left(m_i \cdot n_j \cdot (m_i + n_j)^2\right) \leq \mathcal{O}\left(|T_1| \cdot |T_2| \cdot (\deg(T_1) + \deg(T_2))^2\right) \tag{11}$$

4.4.2 Dynamic programming scheme

The Jiang *et al.* algorithm can be rewritten as a DP scheme [**spire**]. The set-theoretical algebraic adapation of Algorithm 7 consists of recurrences over $\mathbf{JS}[X, Y]$, the set of executions between forests X and Y.

Initialization:

$$\mathbf{JS}[\varnothing, \varnothing] = \{\varepsilon\} \tag{12}$$

Recursive step with one empty forest:

$$\mathbf{JS}[a(u) \circ X, \varnothing] = \{(a, -)\} \times \mathbf{JS}[u, \varnothing] \times \mathbf{JS}[X, \varnothing] \tag{13}$$

$$\mathbf{JS}[\varnothing, b(v) \circ Y] = \{(-, b)\} \times \mathbf{JS}[\varnothing, v] \times \mathbf{JS}[\varnothing, Y] \tag{14}$$

Recursive step with no empty forest:

$$\mathbf{JS}[a(u) \circ X, b(v) \circ Y] = \bigcup \begin{cases} \{(a, b)\} \times \mathbf{JS}[u, v] \times \mathbf{JS}[X, Y] & \text{(15a)} \\ \bigcup_{Y' \circ Y'' = b(v) \circ Y} \{(a, -)\} \times \mathbf{JS}[u, Y'] \times \mathbf{JS}[X, Y''] & \text{(15b)} \\ \bigcup_{X' \circ X'' = a(u) \circ X} \{(-, b)\} \times \mathbf{JS}[X', b] \times \mathbf{JS}[X'', Y] & \text{(15c)} \end{cases}$$

Figure 18: Original DP scheme [**Jiang1995**]

The DP scheme described in Figure 18 is complete (by induction using the fact that equations 15a, 15b and 15c consider all 3 cases when aligning 2 forests) but ambiguous, for 2 main reasons.

The first reason is that if the alignment between 2 forests is empty, i. e. contains no match, the sequence of insertions and deletions created by equations 15b, 15c, 13 and 14 can be shuffled around and still represent the same (empty) alignment, as seen on Figure 19.

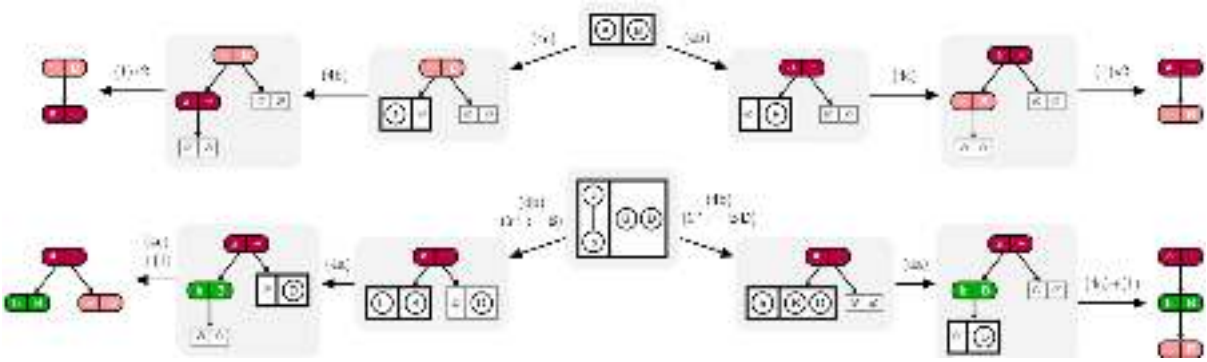


Figure 19: Two examples of ambiguity. Top: an empty alignment can be obtained in multiple ways by arbitrarily interleaving insertions and deletions. Bottom: While aligning two forests, inserting the first root a of the left forest leads to a decomposition of the right forest into two parts, recursively aligned to the children of a and the rest of the left forest respectively. Unfortunately, multiple decompositions may still yield the same alignment,leading to a more intricate level of ambiguity. Figure taken from [**spire**]

The second reason is the partitioning of forests in 2 subforests that occur in Equations 15b and 15c. If the last tree t of Y' is not matched, the same alignment can be obtained with an alternative decomposition $Y = Y_1 \circ Y_2$ in which t is the first tree of Y_2, as seen on Figure 19.

The ambiguity of the DP scheme is a huge draw-back since it prevents the use of numerous theoretical and practical optimization and explorational frameworks [**Giegerich2000**]. Those frameworks provide many advantages:

— exploration of the solutional space (optimal and suboptimal solutions)

— enumeration of all optimal solutions [**Bansal2013**]
— random sampling of solutions under a probability distribution influenced by their cost [**Ding2003**], [**Ponty2011**]

For problems that can be solved by using DP, these goals can be achieved through algebraic substitutions and transforms of the DP scheme.

Part III
Implementation and work

5 Secondary structure of RNAs and python code

In order to incorporate 3D structure information as well as the secondary structure in the scoring matrix of the tree alignment, dihedral angles and secondary structure had to be computed from PDB structures.

Unfortunately the `python` package collection `Biopython` [**biopython**], especially the `Bio.PDB` collection [**biopdb**] do not handle the specifics of RNA PDB structures: no computation of dihedral angles available or secondary structure, not even a test to discriminate purines from pyrimidines.

A new pip installable `python` package was created in order to complete the `Bio.PDB` package: `RNA`.

It is available at `https://gitlab.pimeys.fr/Chopopope/python-pprna` and its documentation is available at `http://doc.pimeys.fr/python-pprna/` on GPLv2.

5.1 `RNA` package workflow

The main purpose of the development of this library was to create a free tool and to give it to the `Biopython` community to expand the software collection. For now, it has been packaged for pypi (`https://pypi.python.org/pypi`) and for debian (`http://debian.org`) as a proof of concept trial.

The pip tarball may be downloaded directly at `http://pommeret.pimeys.fr/pprna-1.0.0b1.tar.gz`, and may be installed using `pip install pprna-1.0.0b1.tar.gz`, whereas the debian package may be downloaded at `http://pommeret.pimeys.fr/python-pprna_1.0.0b1-2_amd64.deb`, and installed using `dpkg -i python-pprna_1.0.0b1-3_amd64.deb`. Such installations requires to be root user, so it assumes that the end user has a good trust in those packages.

With this in mind, the `RNA` library was designed respecting the general class diagram of `Bio.PDB` and by overloading pre-existing classes whenever possible, since the community is concerned by the introduction of new classes.

Most are the result of an overload and customisation of inherited classes from the `Bio.PDB` class collection as it can be seen on the class diagram Figure 20.

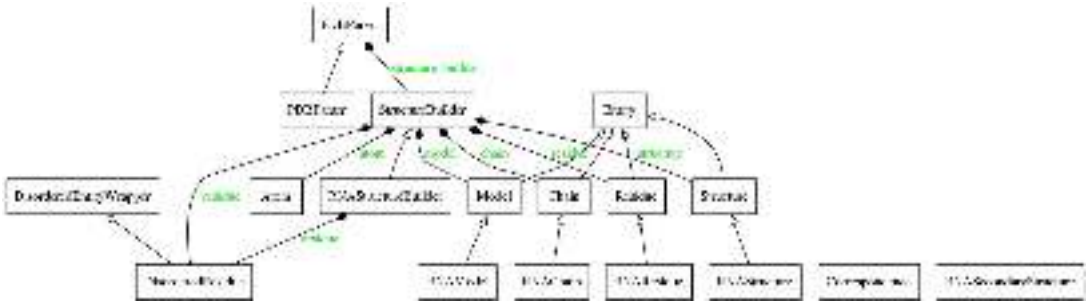


Figure 20: Class diagram of the `RNA` library

5.1.1 Dihedral angles computation

A huge part of the work was the implementation of the computation of all dihedral and torsion angles that are defined in an RNA molecule, for each structure, model and chain that can be defined in a PDB file, considering the fact that some atoms may be missing in the file.

The package allows the computation of every torsion angle of RNA molecules defined in Figure 13 and Equation 9.

Though the process required in-depth modifications of the `Residue`, `Chain`, `Model`, `Struture`, `StructureBuilder` and `PDBParser` classes, it was possible to avoid the creation of new classes.

5.1.2 Secondary structure computation

`Python` does not offer any tool to compute the interaction between RNA base pairs so an external tool had to be used to predict the annotations of an RNA chain: `rnaview` [**Yang2003**].

The `RNASecondaryStructure` had to be created and the `RNAStructureBuilder` had to be modified so that at the instanciation of an RNA object base pairs annotations are created.

The `RNASecondaryStructure` class allows users to get all annotations of every single `RNAChain` object, as well as the annotation preserved by the planarisation algorithm (see Algorithm 4) and the dot-parenthesis sequence (see Subsection 4.2 on page 16).

5.2 Interactions types detected

The first use of the new `RNA` package was to study the distribution of the different types of interactions predicted by the annotation software.

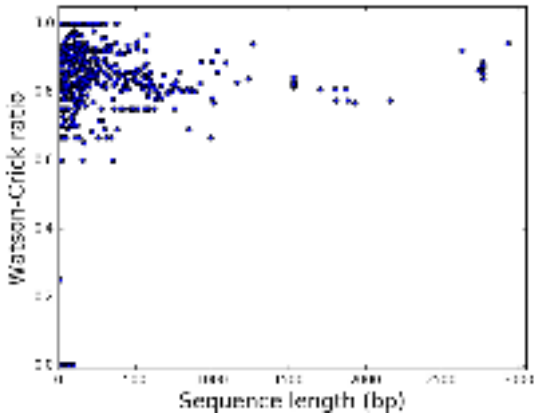


Figure 21: Ratio of Watson-Crick/Watson-Crick interactions detected in the pool of 1088 PDB structures

Among the 12 types of interactions discussed in Subsection 3.6 on page 14, the most common kind is Watson-Crick/Watson-Crick interactions since it represents between 60 and a 100% of the predicted annotations according to Figure 21.

5.3 Effects of the planarization of the secondary structures of the RNA PDB pool

A pool of 1088 structures (list available in Section A) was extracted from the PDB. Those structures were processed by scripts using the new `python` package and their secondary structures annotations computed.

In virtually all structures, it is impossible to keep all the detected interactions between the various edges defined in Figure 14 on page 15 while having a secondary structure, i. e. that satisfies Definition 3.4 on page 9.

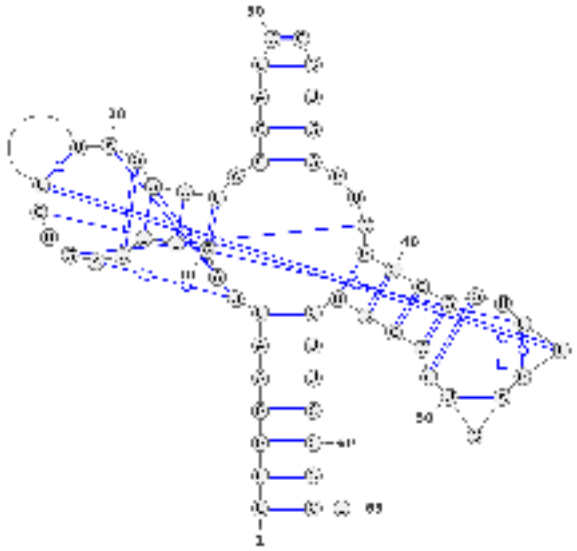


Figure 22: Squiggle-plot representation of RNA 4MGM pre-planarization, i. e. obtained directly from the `rnaview` [**Yang2003**]output using VARNA

For instance, the PDB structure 4MGM pre-planarization is complicated and does not satisfies Definition 3.4, as pseudoknots and crossing arcs can seen in Figure 22. The 27 interactions of various types are not mutually compatible and the planarization algorithm will suppress 9 interactions to produce the secondary structure displayed on Figure 23.

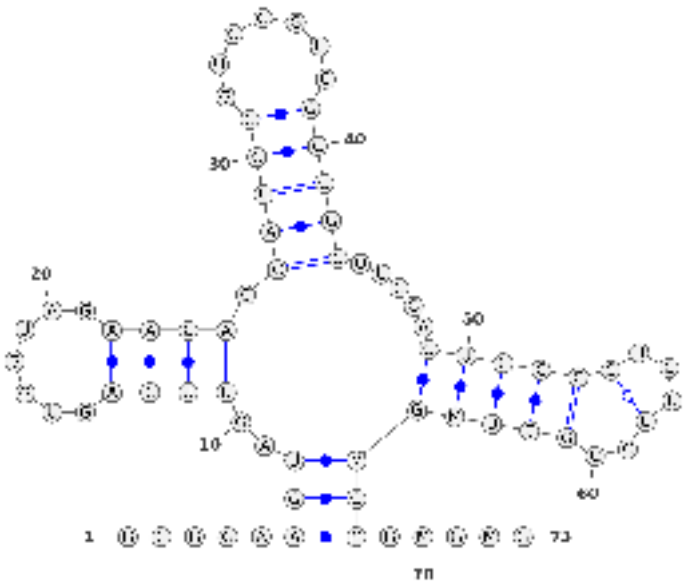


Figure 23: Squiggle-plot representation of RNA 4MGM post-planarization using Algorithm 4

That is the reason why the planarization algorithm (See Algorithm 4 on page 10) is launched automatically at the instanciation of the `RNASecondaryStructure`.

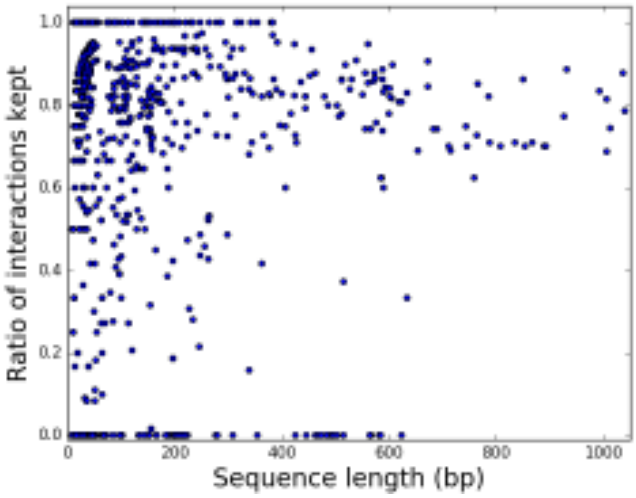


Figure 24: Ratio of arcs kept by the planarization algorithm (Algorithm 4) of the set of annotations detected by `rnaview` [**Yang2003**]in the 1088 structures of length $l < 1050$

Though the work is focused on tree alignment, that is structure that satisfies Definition 3.4 on page 9, it is possible to peak at the suppressed interactions: Figure 24 presents the overall number of interactions suppressed depending on the structure length, while Supplementary Figure 34 on page viii displays the differences of the suppression profiles depending on the edge types that were interacting.

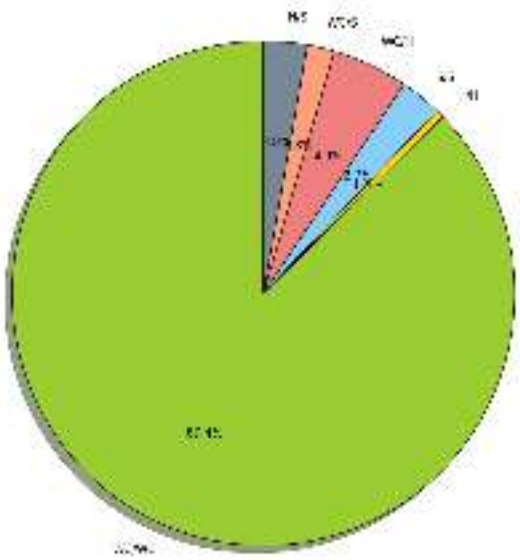


Figure 25: Detected annotations

Moreover, Figure 25 represents the distribution of the annotations detected by `rnaview` [**Yang2003**]in the batch of 1088 structures and shows that the Watson-Crick/Watson-Crick types of annotations are clearly the main type of annotations detected.

Thanks to Figure 26, one can see that after planarization Watson-Crick/Watson-Crick interactions remain the major type of interaction which is consistent with the fact that Watson-Crick/Watson-Crick interactions were the first type of interactions defined and used while studying base pairing.

However, both Figure 25 and Figure 26 show that the other types of interactions defined in Subsection 3.6 on page 14 can not be overlooked and bear some meaning in the shape of an RNA structure.

Those two Figures also show that Hoogsteen/Hoogsteen interactions are the rarest which is consistent with the fact that *cis* Hoogsteen/Hoogsteen interactions were never observed in real structures (see Subsection 3.6 on page 14).

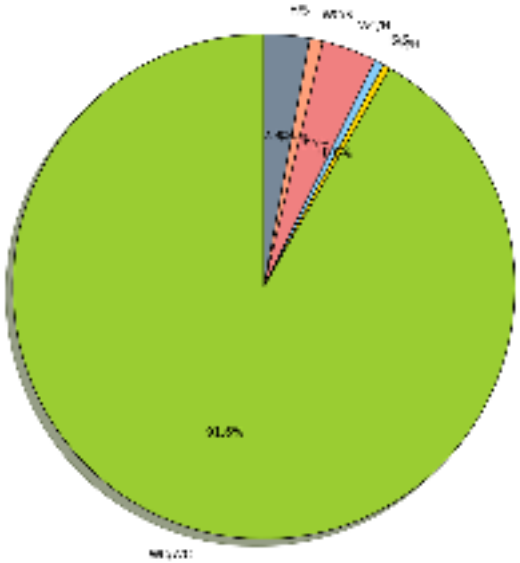
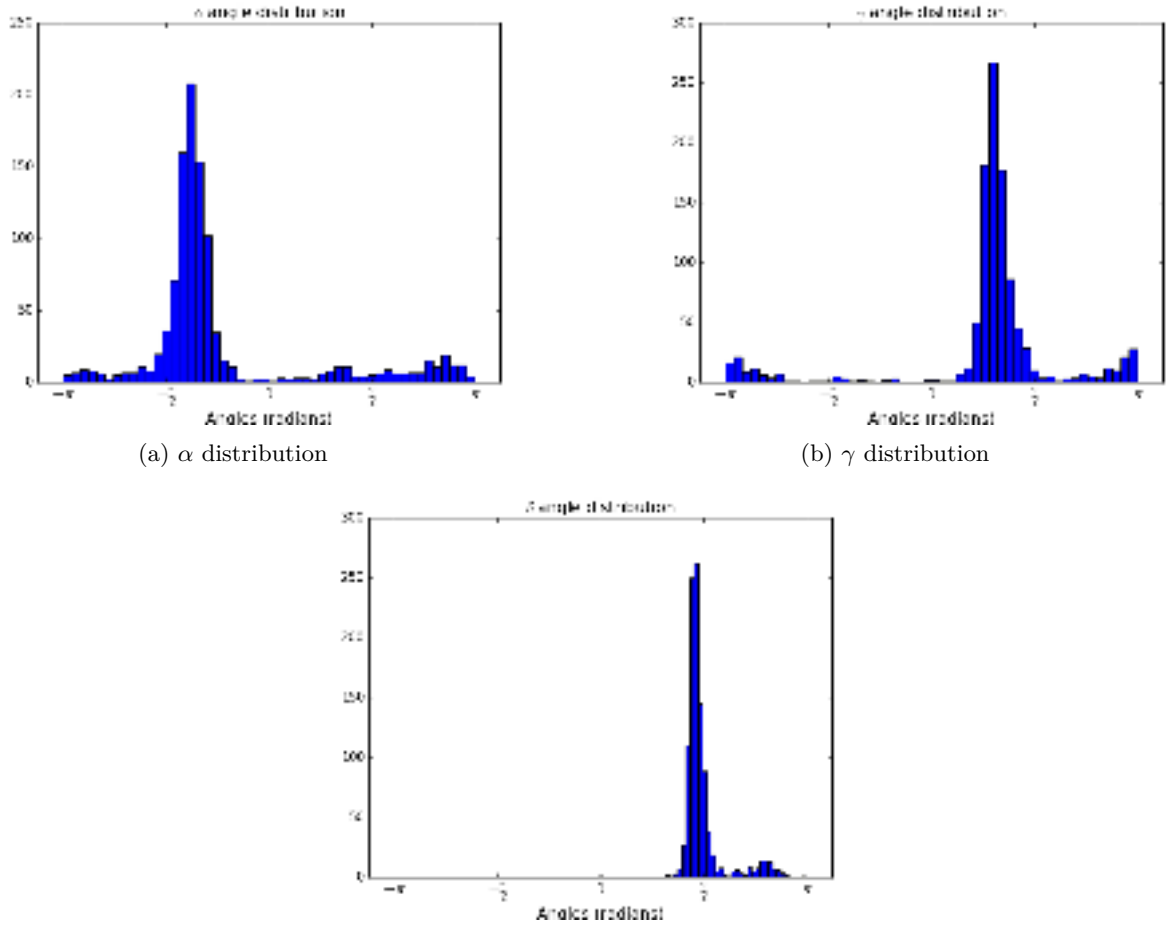


Figure 26: Remaining annotations after planarization

5.4 Analysis of the dihedral angle distribution

Thanks to the developped `python` package, obtaining the distribution of dihedral angles for a PDB structure of an RNA becomes trivial.

It allowed us to conduct a preliminary analysis of the distribution of dihedral angles in a set of RNA structure to test if comparing the value of a given dihedral angle to the median angle was a wise thing to do while producing the scoring matrix in the implementation of the DP scheme.



(a) α distribution

(b) γ distribution

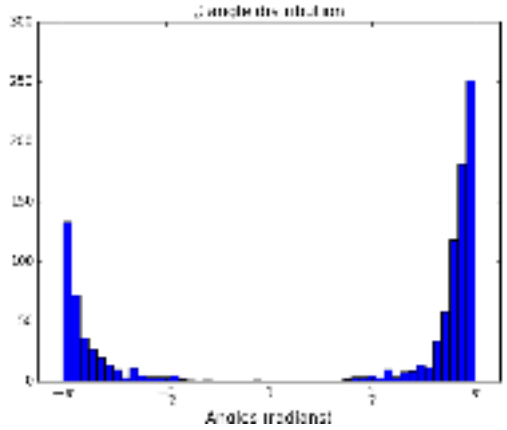
(c) δ distribution

Figure 27: Dihedral angles with bimodal distributions

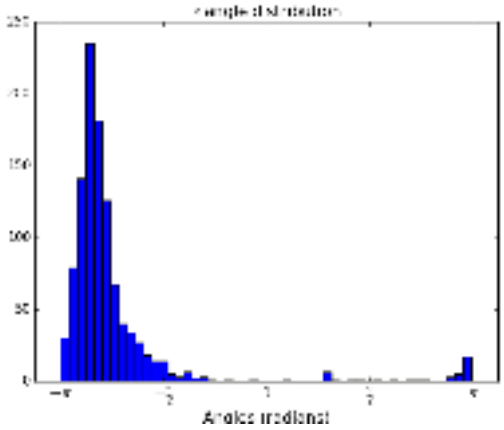
Figure 27 that 3 out of 9 dihedral angles have bimodal distributions with a major and a minor peak, which is unexpected. Moreover, those peaks seem to follow a Gaussian distribution which changes the scoring scheme we had in mind. Indeed, instead of using a simplistic angle substraction in the scoring scheme, we ought to have a scoring scheme that takes into account the probability of the angle being in one of those peaks.

On the other hand, as seen in Figure 28, 6 angles display unimodal distribution that also seem to be Gaussian which indicates that the scoring scheme of all angles should take into account the probability of belonging in the gaussian domain.

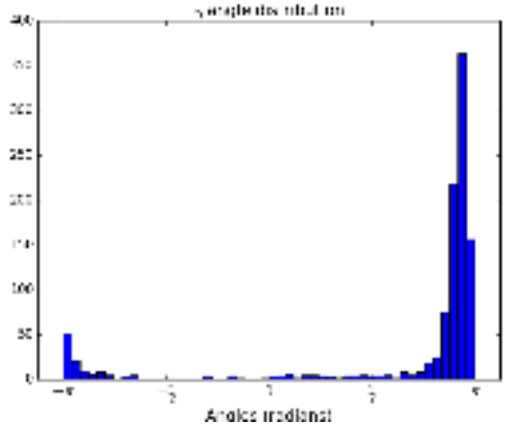
Those results, computed from a set of 1088 RNA structures (the complete list can be found in Section A), confirm previously observed results on 23S and 5S ribosomal RNA (rRNA) of the crystal structure of the 50S ribosomal subunit (PDB code 1JJ2) in [**Bohdan2004**]).



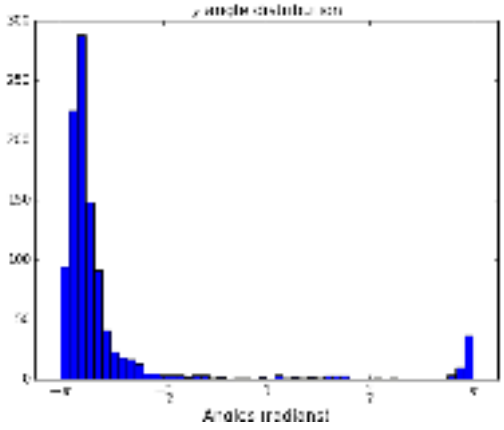
(a) β distribution



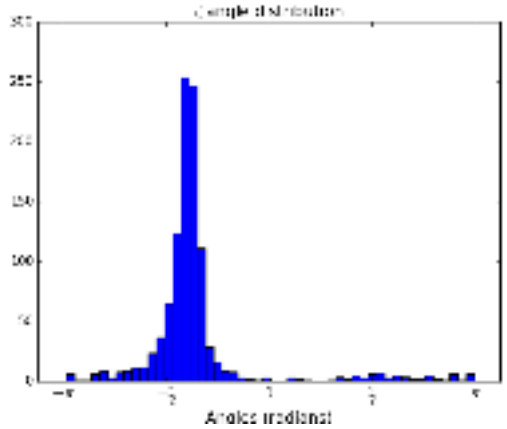
(b) ε distribution



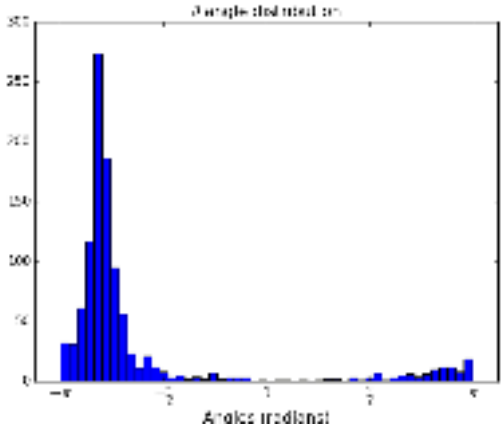
(c) η distribution



(d) χ distribution



(e) ζ distribution



(f) θ distribution

Figure 28: Dihedral angles with unimodal distribution

6 Tree alignment: implementation of the dynamic programming scheme

6.1 *Translation* of the unambiguous dynamic programming scheme

The unambiguous DP scheme written by Chauve *et al.* [**spire**] can be found in Figure 29 on the facing page. It is way more complex and long than the original DP scheme presented Figure 18 on page 20.

Those equations need to be translated into 2 algorithms: the first one to fill the DP tables and the second one to backtrack the results. This is done by creating *control structures* for those 2 parts that will execute Equations 16a through 23c depending on context.

The implementation is in C++ and the code can be found at `https://gitlab.pimeys.fr/Chopopope/cpp-pprna` and its documentation is currently only available in the source files.

The code is not of an easy-access: in order to optimize the code, especially the memory taken, a lot of indirections were created, for instance ranking and unranking of indexes and mappings between different sets of indexes.

Moreover, as stated in Subsubsection 4.4.2, the ambiguity of the DP scheme prevented users to use optimization and exploration frameworks.

The DP scheme implemented here is unambiguous, which opens quite a few applications. The implementation had to allow those applications, this is why every operator, test, constant and initial value weren't *hardcoded* but defined as macros that, depending on compilation and/or execution parameters would allow the users to explore different sort of backtracks.

Table 1: Algebra definition depending on compilation options

Macros	Partition algebra	Optimization algebra
PLUS(a, b)	$a \times b$	$a + b$
MIN(a, b)	$a + b$	$\min(a, b)$
ZERO	1	0
INFTY	0	∞
CHOICE(a, b)	$a < b$	$a = b$
UPDATE(a, b)	$a - b$	$\varnothing$

As Table 1 states, the implementation is *algebra-sensitive*.

The code computes the tree alignment of two RNA secondary structures. It takes as *input*:
- two dps stored in two different files, those dps can be obtained from PDB structures thanks to the work presented Section 5;
- a score matrix given in **csv!** (**csv!**) format.

The output is:
- a dump of the DP tables (in **csv!** format)
- a trace of the program
- the alignment tree (see Figure 30 to understand how the tree alignment is produced from the DP equations)

The DP equations are re-interpreted as subtrees during the backtrack of the matrices in order to produce one final tree. The way the DP equations can be converted into subtrees is explained on Figure 30.

Initialization:

$$H\left[\varnothing,\varnothing\right]^{\alpha}_{M,M'} = \begin{cases} \{\varepsilon\} & \text{if } M,M' = \varnothing,\varnothing \quad (16a) \\ \varnothing & \text{otherwise} \quad (16b) \end{cases}$$

$$V\left[X,\varnothing\right]^{\varnothing} = V\left[\varnothing,X\right]^{\varnothing} = V\left[X,\varnothing\right]^{\uparrow} = V\left[\varnothing,X\right]^{\uparrow} = VH\left[\varnothing,b(v)\right] = \varnothing \quad (17)$$

Recursive step: H table

$$H\left[a(u)\circ X,\varnothing\right]^{\alpha}_{M,M'} = \begin{cases} (a,-)\times H\left[u,\varnothing\right]^{\alpha}_{M,M'} \times H\left[X,\varnothing\right]^{\alpha}_{M,M'} & \text{if } \alpha,M,M' = I|D,\varnothing,\varnothing \quad (18a) \\ \varnothing & \text{otherwise} \quad (18b) \end{cases}$$

$$H\left[\varnothing,b(v)\circ Y\right]^{\alpha}_{M,M'} = \begin{cases} (-,b)\times H\left[\varnothing,v\right]^{\alpha}_{M,M'} \times H\left[\varnothing,Y\right]^{\alpha}_{M,M'} & \text{if } M,M' = \varnothing,\varnothing \quad (19a) \\ \varnothing & \text{otherwise} \quad (19b) \end{cases}$$

$$H\left[a(u)\circ X,b(v)\circ Y\right]^{\alpha}_{M,M'} =$$

$$\bigcup \begin{cases} (a,-)\times H\left[u,\varnothing\right]^{I|D}_{\varnothing,\varnothing} \times H\left[X,b(v)\circ Y\right]^{I|D}_{M,M'} & \text{if } \alpha = I|D, M \neq \leftrightarrow \quad (20a) \\ (-,b)\times H\left[\varnothing,v\right]^{D}_{\varnothing,\varnothing} \times H\left[a(u)\circ X,Y\right]^{\alpha}_{M,M'} & \text{if } M' \neq \leftrightarrow \quad (20b) \\ \bigcup\limits_{\substack{Y'\circ Y''=b(v)\circ Y \\ |Y'|\geq 2}} (a,-)\times H\left[u,Y'\right]^{I|D}_{\varnothing,\leftrightarrow} \times H\left[X,Y''\right]^{I|D}_{\delta(M,X),\delta(M',X'')} & \quad (20c) \\ \bigcup\limits_{\substack{X'\circ X''=a(u)\circ X \\ |X''|\geq 2}} (-,b)\times H\left[X',v\right]^{I|D}_{\leftrightarrow,\varnothing} \times H\left[X'',Y\right]^{I|D}_{\delta(M,X''),\delta(M',Y)} & \quad (20d) \\ V\left[a(u),b(v)\right]^{\varnothing} \times H\left[X,Y\right]^{I|D}_{\delta(M,X),\delta(M',Y)} & \quad (20e) \end{cases}$$

where

$$\delta(\leftrightarrow,S) = \delta(\rightarrow,S) = \begin{cases} \varnothing & \text{if } |S| = \varnothing \\ \rightarrow & \text{otherwise} \end{cases}, \text{ and } \delta(\varnothing,S) = \varnothing.$$

Recursive step: V table

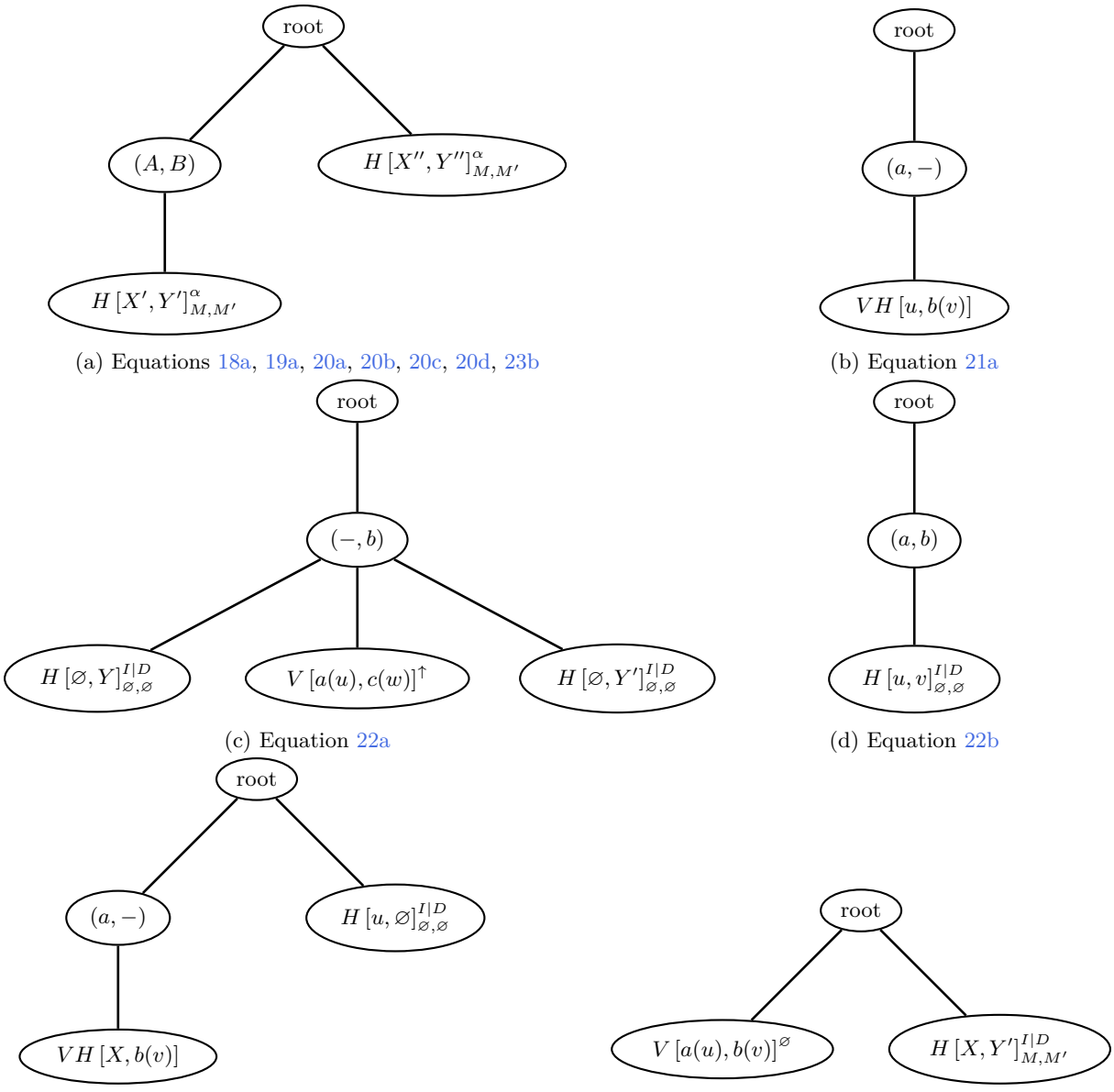
$$V\left[a(u),b(v)\right]^{\varnothing} = \bigcup \begin{cases} (a,-)\times VH\left[u,b(v)\right] & (21a) \\ V\left[a(u),b(v)\right]^{\uparrow} & (21b) \end{cases}$$

$$V\left[a(u),b(v)\right]^{\uparrow} = \bigcup \begin{cases} \bigcup\limits_{v=Y\circ c(w)\circ Y'} (-,b)\times H\left[\varnothing,Y\right]^{I|D}_{\varnothing,\varnothing} \times V\left[a(u),c(w)\right]^{\uparrow} \times H\left[\varnothing,X'\right]^{I|D}_{\varnothing,\varnothing} & (22a) \\ (a,b)\times H\left[u,v\right]^{I|D}_{\varnothing,\varnothing} & (22b) \end{cases}$$

Recursive step: VH step

$$VH\left[a(u)\circ X,b(v)\right] = \bigcup \begin{cases} (a,-)\times H\left[u,\varnothing\right]^{I|D}_{\varnothing,\varnothing} \times VH\left[X,b(v)\right] & (23a) \\ \bigcup\limits_{\substack{X'\circ X''=a(u)\circ X \\ |X'|\geq 2}} (-,b)\times H\left[X',v\right]^{I|D}_{\leftrightarrow,\varnothing} \times H\left[X'',\varnothing\right]^{I|D}_{\varnothing,\varnothing} & (23b) \\ V\left[a(u),b(v)\right]^{\varnothing} \times H\left[X,\varnothing\right]^{I|D}_{\varnothing,\varnothing} & (23c) \end{cases}$$

Figure 29: The unambiguous DP scheme [**spire**]



(a) Equations 18a, 19a, 20a, 20b, 20c, 20d, 23b

(b) Equation 21a

(c) Equation 22a

(d) Equation 22b

(e) Equation 23a

(f) Equations 20e, 23c

Figure 30: Tree conversions for the equations of the DP scheme in Figure 29 to produce the resulting tree alignment as a supertree

6.2 Program inputs

6.2.1 The `dps` structure

The first input are two files storing each a string of characters representing a dps. The files contain a strict dps, that is only the *dot-parenthesis* sequence and not the RNA bases.

After loading the string of character into the memory, the sequence is processed by a check-sum algorithm (see Algorithm 8)to verify the correcteness of the bracketing of the dps.

Once the integrity of the dps has been checked, thanks to Algorithm 5 on page 17 it is possible to obtain a tree structure representing the RNA secondary structure.

Algorithm 8: Preprocessing of the dps to check bracketing (see `dps::scan_sequence` for implementation)

```
Function ScanSequence
    Data: S                                   /* string representing the sequence */
    Result: bool
    /* Returns True if the sequence is a dps; otherwise throws an error.       */
begin
    sum ← 0
    for 0 ≤ i < len(S) do
        if sum < 0 then
            throw "Missing '(' sequence at position %i" % (i)
        end
        if S[i] = '(' then
            sum++
        else if S[i] = ')' then
            sum--
        else if S[i] = '.' then
            pass
        else
            throw "Unauthorized character"
        end
    end
    if sum < 0 then
        throw "At least one missing '(' in sequence"
    else if sum > 0 then
        throw "At least one missing ')' in sequence"
    else
        return True
    end
end
```

It is stored in a `dps` structure, defined in `structures/dps.hpp`.

6.2.2 Score matrix

For the sake of space optimization, the score matrix is condensed to the bare minimum of space (see Table 2). The overall idea is to use the nodes of the trees instead of the bases to index the score matrix. Thus, we spare some values that are irrelevant. For instance, bases 4 and 8 of T_1 (Figure 31a) are represented by node $(4, 8)$ saving one column in the matrix.

Unfortunately, as partially explained in Section 5.4, the scoring scheme we want to implement is complex and thus, we didn't have time to produce a script that creates, given the PDB files of two RNA, the appropriate scoring matrix in Comma-Separated Values (CSV) format and only tested the DP code on simple scoring matrix.

		∅	(1, 1)	(2, 2)	(3, 3)	(4, 8)	(5, 5)	(6, 6)	(7, 7)	(10, 10)	(11, 11)
		0	1	2	3	4	5	6	7	8	0
∅	0	$\alpha_{0,0}$									
(1, 11)	1										
(2, 10)	2										
(3, 9)	3										
(4, 9)	4					$\alpha_{i,j}$					
(5, 5)	5										
(6, 6)	6										
(7, 7)	7										

Table 2: Score matrix for the alignment of tree T_1 (Figure 31a) with tree T_2 (Figure 31b) saving 47.9% space

6.3 Trees

6.3.1 `tree` structure

The tree structure, defined in file `structures/tree.hpp` is created by the `parse` method of a `dps` object. The main features of this class are described in Table 4 in Appendix C.

6.3.2 Indexation of trees

For optimization purposes during the unambiguous alignment of trees and subforests described in Figure 29 a list of nodes ordered in the anti breadth-first search order has to be yield: Figure 31a and Figure 31b.

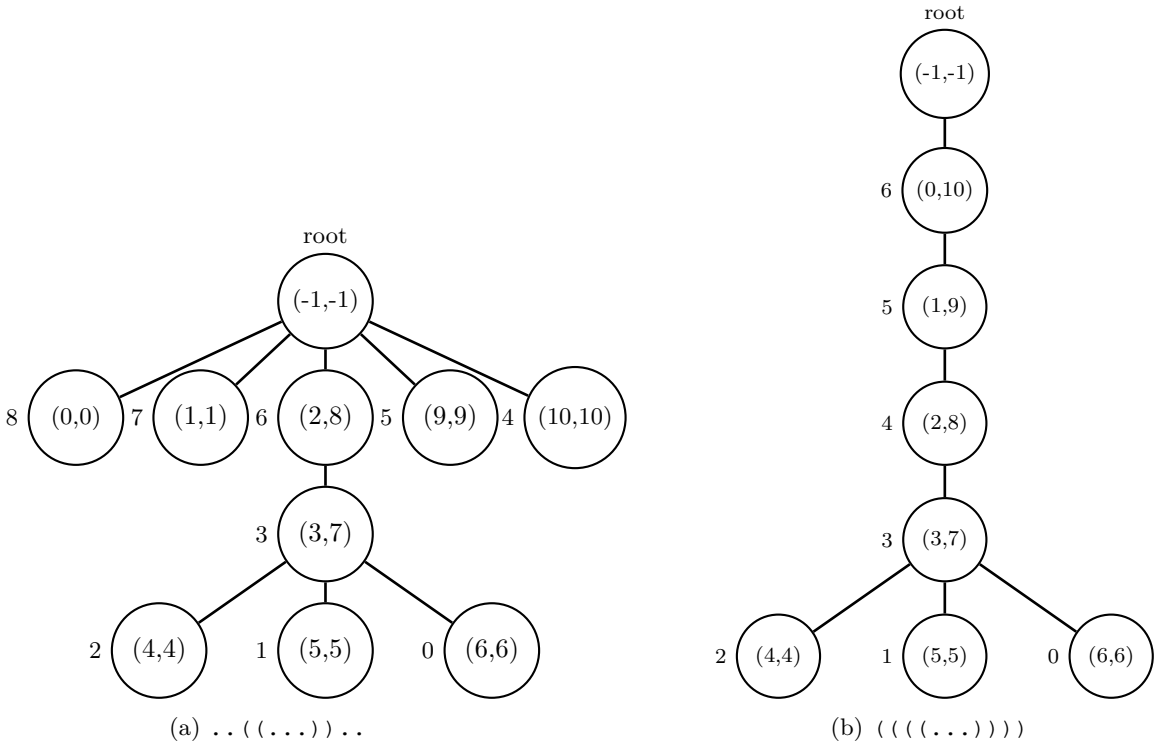


Figure 31: Tree representation with node indexation of two dps

It allows an easier definition of generated forests 6.4.1.

6.4 Structures dealing with forests

Definition 6.1 (Infix forest)**.** An *infix forest* of a tree is a contiguous sequence of subtrees rooted at a given node, the parent of the forest.

A *true infix forest* (called *infix forest* in the following) is an infix forest that isn't a suffix forest. The set of those forests it noted I^*.

Definition 6.2 (Suffix forest)**.** A *suffix forest* of a tree is an infix forest that containts the last node of the sisterhood. The set of those forests is noted S.

6.4.1 The `Forest` structure

The data structure `Forest` is defined in the file `cpp/structures/forest.hpp`. A forest has 4 attributes.

start : the start index of the forest in its super tree indexation

end : the end index of the forest in its super tree indexation

size : the *size* of the forest, that is the number of nodes and leafs it contains

type : a boolean that is at `True` if the forest is infix, `False` otherwises

6.4.2 Generating forests

In Algorithm 9 and 10, we provide two functions designed to generate a forest using one or two trees. The forests are generated while the tree is indexed in order to spare one tree traversal, as it could be time consuming on large trees. As the tree is indexed using anti breadth-first search order, one can guarantee that all possible forests are generated, starting from the deepest leaf nodes, so that the size cache attribute can be updated without any more tree traversals.

Algorithm 9: Generating forest from a single tree

```
Function generateForestFromTree(t)
    Data: t                                                    /* tree node */
    Result: f                                  /* the generated forest of tree t */
begin
    if t.get_parent() = NULL then
        /* t is the root node of the tree, it is suffix                    */
        return Forest(t.get_index(), t.get_index(), t.size(), False)
    end
    if t.get_children().size() = 0 then
        /* No children                                                     */
        return EMPTY_FOREST
    end
    i ← t.get_index()
    s ← t.get_size()
    t ← True
    if t.sister_number(t.get_index()) = t.get_parent().get_children().size() - 1 then
        /* The position of current node in its sisterhood is the last one, the
           forest is therefore suffix                                      */
        t ← False
    end
    return Forest(i, i, s, t)
end
```

Algorithm 10: Generating forests from two trees

```
Function generateForestFromTree(t1, t2)
    Data: t1, t2                                  /* tree nodes of the same sisterhood */
    Result: f                                     /* the generated forest between t1 and t2 */
begin
    if t1.get_index() = t2.get_index() then
        return generateForestFromTree(t1)
    end
    if t1.get_index() > t2.get_index() then
        return EMPTY_FOREST
    end
    i1 ← t1.get_index()
    i2 ← t2.get_index()
    s ← 0
    t ← True
    sisterhood = t1.get_parent().get_children()
    s1 ← t1.sister_number(i1)
    s2 ← t2.sister_number(i2)
    for s1 ≤ j ≤ s2 do
        s ← s + 1
    end
    if s2 =sisterhood.size() −1 then
        t ← False
    end
    return Forest(i1, i2, s, t)
end
```

6.4.3 The `Infixe_suffixe` structure

The `Infixe_suffixe` structure is a *read-only* structures that is instanciated from a vector of `Forest` defined in `cpp/structures/infixe_suffixe.hpp`. It consists of two vectors of `Forest`, one containing the infix forests and the other one the suffixes.

The forests are sorted by size and then by their index so that they are in an order compatible with the order of computation of DP tables summed up in Figure 32.

6.5 Encoding of the configuration

Notations. In the DP scheme defined Figure 29, configurations are defined (see Definition 6.3)
— $\alpha \in \{I|D, D\}$ is the top configuration
— $(M, M') \in \{\varnothing, \leftrightarrow, \rightarrow\}^2$ is the bottom configuration

A `Config` structure in `cpp/structures/configurations.hpp` defines these configurations. They are defined using cpp enumerators.

The *default configuration* is set at $\alpha = I|D, M = \varnothing, M' = \varnothing$, the start configuration for the backtrack.

6.6 DP tables

The unambiguous dynamic programming scheme on Figure 29 uses 4 kinds of dynamic programming tables:

— several H tables,
— a VH table,
— a $V^{\varnothing}$ table,
— a $V^{\uparrow}$ table.

Looking at Figure 29, Equation 20e one can see that H tables call cells from $V^{\varnothing}$ tables with forests of same sizes so $V^{\varnothing}$ tables (for every H table configuration) need to be compute before H tables.

Moreover, the computation of cells from $V^{\varnothing}$ tables in Equation 21 call cells from $V^{\uparrow}$ tables so $V^{\uparrow}$ tables need to be calculated before $V^{\varnothing}$ tables.

VH tables call cells from $V^{\varnothing}$ tables in Equation 23 so those need to be computed before hand.

The conclusion is that the DP tables of each possible configuration need to be computed in a specific order described in Figure 32:

$$V^{\uparrow} \to V^{\varnothing} \to VH \to H$$

Figure 32: Order of the computation of DP tables

6.6.1 H table: forest versus forest

Definition 6.3 (H table)**.** Let X (resp. Y) be a forest of S (resp. T) and $(\alpha, M, M') \in \{I|D, D\} \times \{\varnothing, \leftrightarrow, \rightarrow\}^2$. $H[X, Y]^{\alpha}_{M,M'}$ is the score of the alignment between forests X and Y such as:

1. the first tree in X cannot be inserted if $\alpha = D$
2. the first and last (resp. only last) tree(s) in X must be matched if $M = \leftrightarrow$ (resp. $M = \rightarrow$)
3. the first and last (resp. only last) tree(s) in Y must be matched if $M' = \leftrightarrow$ (resp. $M' = \rightarrow$)

The consequence of Definition 6.3 is that DP scheme does not allow the alignment of two infix forests againt one another.

As it can be seen in Figure 33, since the H table of each configuration $(\alpha, M, M') \in \{I|D, D\} \times \{\varnothing, \leftrightarrow, \rightarrow\}^2$ are indexed by their forests, a large part of the H table isn't used: the left-hand top part.

This is why a ranking-unranking method, schematically explained by the arrows in Figure 33, was used.

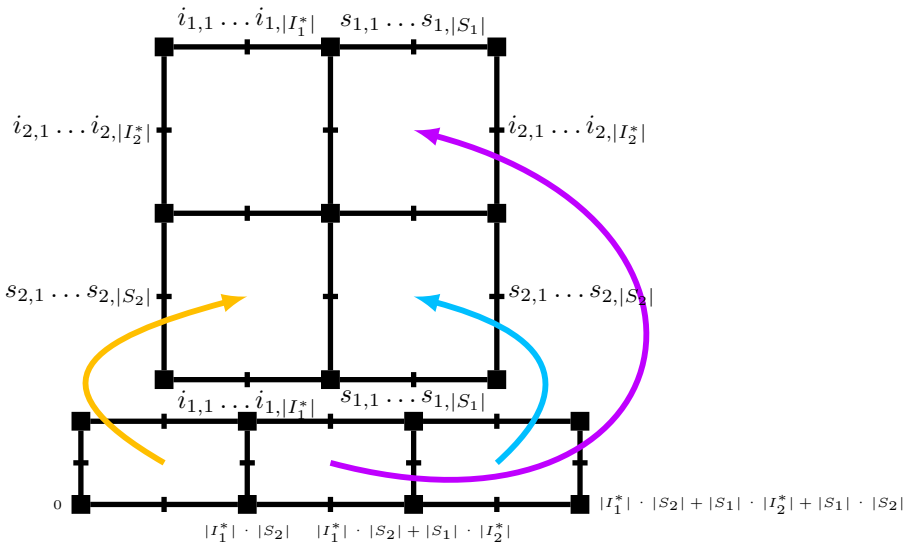


Figure 33: Shifting from a 2 dimensional H table to a 1 dimensional H-table

The goal is to take less space in memory and less time in computing the H table by avoiding the alignments of two infix forests that are not taken into account in the DP scheme described in Figure 29.

In the end, thanks to Algorithm 11 and 12 and Table 3, we get index $\in [\![0; |I_1^*| \times |S_2| + |S_1| \times |I_2^*| + |S_1| \times |S_2|[\![$ instead of matrix of size $(|I_1^*| + |S_1|) \times (|I_2^*| + |S_2|)$.

Algorithm 11: Ranking of two forests for a H table

Function *rankForest*(i, j, s, t)
  Data: i /* index of forest X in vector of generated forests of S */
  Data: j /* index of forest Y in vector of generated forests of T */
  Data: s /* size of the interval to be considered (see Table 3) */
  Data: t /* offset to be considered (see Table 3) */
  Result: index /* index of alignment score of forests X and Y in the H table */
begin
  return $t + i \times s + j$
end

Algorithm 12: Unranking of two forests for a H table

Function *unrankForest*(*index*)
  Data: index /* index of alignment score of forests X and Y in the H table */
  Result: (i, j) /* indexes of forests X and Y in the vector of generated forests of S and T */
begin
  if *index* $\in [\![0; |I_1^*| \times |S_2|]\!]$ **then**
    $j \leftarrow \text{index}\%|S_2|$
    $i \leftarrow (\text{index} - j)/|S_2|$
  end
  $\text{index}' \leftarrow \text{index} - |I_1^*| \times |S_2|$
  if *index*$'$ $\in [\![0; |S_1| \times |I_2^*|]\!]$ **then**
    $j \leftarrow \text{index}'\%|I_2^*|$
    $i \leftarrow (\text{index}' - j)/|I_2^*|$
  end
  $\text{index}'' \leftarrow \text{index}' - |S_1| \times |I_2^*|$
  if *index*$''$ $\in [\![0; |S_1| \times |S_2|]\!]$ **then**
    $j \leftarrow \text{index}''|S_2|$
    $i \leftarrow (\text{index}'' - j)/|S_2|$
  end
  return (i, j)
end

Using another yet similar ranking-unranking method, the H tables of the DP scheme can be represented by a single matrix with $|I_1^*| \times |S_2| + |S_1| \times |I_2^*| + |S_1| \times |S_2|$ columns and $2 \times 3 \times 3 = 19$ lines (configurations).

Table 3: Offset and size of intervals to be considered in the ranking/unranking process of Algorithm 11 and 12 for the H table

Forest X type	Forest Y type	s value	t value
infix	suffix	$\|S_2\|$	0
suffix	infix	$\|I_2^*\|$	$\|I_1^*\| \times \|S_2\|$
suffix	suffix	$\|S_2\|$	$\|I_1^*\| \times \|S_2\| + \|S_1\| \times \|I_2^*\|$

6.6.2 $V^\beta, \beta \in \{\uparrow, \varnothing\}$ tables: tree versus tree

Definition 6.4 ($V^\uparrow$ table)**.** Let $a(u)$ and $b(v)$ be 2 trees. $V[a(u), b(v)]^\uparrow$ is the score of the alignement between $a(u)$ and $b(v)$ when $a(u)$ and $b(v)$ are matched and a must be matched to some $c \in b(v)$.

Definition 6.5 ($V^{\varnothing}$ table)**.** Let $a(u)$ and $b(v)$ be 2 trees. $V[a(u), b(v)]$ is the score of the alignment between $a(u)$ and $b(v)$ when $a(u)$ and $b(v)$ are matched.

As explained in the above definitions, $V^{\uparrow}$ and $V^{\varnothing}$ are DP tables containing the alignments of two trees. The number of values stored in each of these tables cannot be minimized in any way without losing information. However, for convenience purposes, using similar ranking-unranking methods as the one explained in Algorithm 11 and 12, they can be represented by a one row matrix of size $|T_1| \times |T_2|$.

6.6.3 VH table: infix/suffix versus tree

Definition 6.6 (VH table)**.** Let X be a non-empty forest and $b(v)$ be a tree. $VH[X, b(v)]$ is the score of the alignment between X and $b(v)$ such as $b(v)$ is matched.

The VH table is a hybrid table containing the alignment of forests vs. trees. Again, for convenience purposes and thanks to ranking-unranking methods provided in Algorithm 11 and 12, they can be represented by a one-row matrix of size $(|I_1^*| + |S_1|) \times |T_2|$.

6.7 Control structure

The *control structure* is the core structure of the program; it is *translated* from the DP scheme (Figure 29) into an algorithm that can then be implemented.

Algorithm 13: Control structure for the filling step

```
begin
    for i ∈ ⟦0; |I_1^*| + |S_1|⟦ do
        forest_1 ← generated_forests_1[i]
        start_1 ← forest_1.start()
        end_1 ← forest_1.end()
        for j ∈ ⟦0; |I_2^*| + |S_2|⟦ do
            forest_2 ← generated_forests_2[j]
            start_2 ← forest_2.start()
            end_2 ← forest_2.end()
            if start_1 = end_1 & start_2 = end_2 then
                /* Ranking for V^↑ and V^∅                    */
                /* Populate V^∅ table                         */
                /* Populate V^↑ table                         */
            end
            if start_2 = end_2 then
                /* Ranking for VH table                       */
                /* Populate VH table                          */
            end
            if ¬(forest_1.is_infix() & forest_2.is_infix()) then
                for c ∈ ⟦0; 2 × 3 × 3⟦ do
                    /* Ranking configuration i.e.  H table row */
                    /* Ranking H table column                 */
                    /* Populate H table                       */
                end
            end
        end
    end
end
```

It requires numerous bijections and mappings of the different sets of indexes: indexes of the bases in their sequences, nodes indexes, indexes in the *score matrix*, indexes in the various tables, indexes in the generation of forests in order to compute as efficiently as possible the values of the DP tables as

well as their backtrack. For obvious clarity reasons, those mappings will not be displayed in the *control structures* presented here.

The first *control structure* to be considered is the one that explicits how the DP tables are filled, that is Algorithm 13.

The backtrack of the DP scheme can be divided into 4 parts, one for each of the tables. The backtrack control structure for the H table (Algorithm 16) calls the other backtracks, depending on the cell values and the considered algebra (for meanings of capitalized words, see Table 1 on page 28) other backtracks: Algorithm 14, 15 or 17.

Algorithm 14: Backtrack for the $V^{\varnothing}$ table

```
begin
    r ← INIT(0, index_V^∅)
    current_case ← Equation 21a()
    if CHOICE(r, current_case) then
        /* Case Equation 21a on page 29 */
        return Tree of type 30b
    end
    UPDATE(r, current_case)
    current_case ← Equation 21b()
    if CHOICE(r, current_case) then
        /* Case Equation 21b on page 29 */
        return Backtrack_V^↑_table
    end
end
```

Those backtrack algorithms all return rooted trees which configuration depend on the selected equation, as explained on Figure 30 on page 30.

Algorithm 15: Backtrack of VH table

```
begin
    r ← INIT(0, index_VH
    current_case ← Equation 23a()
    if CHOICE(r, current_case) then
        /* Case Equation 23a on page 29 */
        return Tree of type 30e
    end
    UPDATE(r, current_case)
    current_case ← Equation 23b()
    if CHOICE(r, current_case) then
        /* Case Equation 23b on page 29 */
        for forest ∈ decomposition(a(u) ∘ X) do
            current_case ← Equation 23b(forest)
            if CHOICE(r, current_case) then
                return Tree of type 30a
            end
        end
    end
    UPDATE(r, current_case) current_case ← Equation 23c()
    if CHOICE(r, current_case) then
        /* Case Equation 23c on page 29 */
        return Tree of type 30f
    end
end
```

Algorithm 16: Backtrack H table

```
begin
    r ← INIT(index_H_row, index_H_col)
    c ← H [index_H_row, index_H_col]
    current_case ← ZERO
    if CHOICE(r, current_case) then
        /* Case Equations 16 on page 29                                  */
        empty_tree ← Tree()
        if config.m1 = ∅&config.m2 = ∅ then
            /* Case Equation 16a on page 29                              */
            empty_tree.set_data(None, None)
        end
        else
            /* Case Equation 16b on page 29                              */
        end
    end
    UPDATE(r, current_case)
    current_case ← Equation 20a()
    if CHOICE(r, current_case) then
        /* Case Equation 20a on page 29                                  */
        return Tree of type 30a
    end
    UPDATE(r, current_case)
    current_case ← Equation 20b()
    if CHOICE(r, current_case) then
        /* Case Equation 20b on page 29                                  */
        return Tree of type 30a
    end
    UPDATE(r, current_case)
    current_case ← Equation 20c
    if CHOICE(r, current_case) then
        /* Case Equation 20c on page 29                                  */
        for forest ∈ decomposition(b(v) ∘ Y) do
            current_case ← Equation 20c(forest)
            if CHOICE(r, current_case) then
                return Tree of type 30a
            end
        end
    end
    UPDATE(r, current_case)
    current_case ← Equation 20d
    if CHOICE(r, current_case) then
        /* Case Equation 20d on page 29                                  */
        for forest ∈ decomposition(a(u) ∘ X) do
            current_case ← Equation 20d(forest)
            if CHOICE(r, current_case) then
                return Tree of type 30a
            end
        end
    end
    UPDATE(r, current_case)
    current_case ← Equation 20e()
    if CHOICE(r, current_case) then
        /* Case Equation 20e on page 29                                  */
        return Tree of type 30f
    end
    UPDATE(r, current_case)
end
```

Algorithm 17: Backtrack for $V^{\uparrow}$ table

```
begin
    r ← INIT(0, index_V↑)
    current_case ← Equation 22a()
    if CHOICE(r, current_case) then
        /* Case Equation 22a on page 29                                    */
        for forest ∈ decomposition_3(v) do
            current_case ← Equation 22a(forest)
            if CHOICE(r, current_case) then
                return Tree of type 30c
            end
        end
    end
    UPDATE(r, current_case)
    current_case ← Equation 22a()
    if CHOICE(r, current_case) then
        /* Case Equation 22b on page 29                                    */
        return Tree of type 30d
    end
end
```

6.8 Time and space complexity

The DP scheme defined by Equations 16b on page 29 to 23b is complete (every element of the search space is represented by an execution) and is unambiguous (only one execution leads to an element of the search space) but, most surprisingly, has the same time and space asymptotic complexity than the original DP scheme defined in Equations 12 on page 20 to 15c.

The theoretical time complexity of the DP scheme of Figure 29 on page 29 describded in Algorithm 13 on page 37 is the same as the one stated in 11 on page 19.

The space complexity of the implementation of the DP scheme could be better since the backtrack starts with the *default configuration*: $I|D, \varnothing, \varnothing$ and the study of the DP scheme in Figure 29 shows that not all configuration are accessible starting with the *default configuration*. Therefore, the number of lines in the H table could change from 18 to 12.

7 Conclusion

We managed to produce a PEP8 and Biopython-compliant python package that allows the user to load RNA PDB structures and to proceed to a fair number of computations such as predicting it's secondary interactions, generating a dot-parenthesis sequence by planarizing the predicted arc set or obtaining the dihedral angles of every residue while handling non-typical residues and missing atoms.

Moreover, we obtained a working C++ implementation of the DP scheme for both the matrix population part and the backtrack part that takes as input two dot-parenthesis sequence and a scoring scheme and produces the DP tables and the associated supertree.

However, we cannot at this stage produce a working pipeline that, given two PDB files extracts all the relevant data and uses them to produce the scoring matrix and the dot-parenthesis sequences and loads those information into the C++ implementation. Indeed we did not have enough time to properly define the scoring model we would like to implement.

The lack of a python library to deal with RNA PDB file and the will to fix that issue (and to submit the code to the Biopython team so that no-one has to re-implement that kind of things ever again) lead us to spend more time than predicted on the development and the testing phase of the pprna package.

Moreover, the implementation of the DP scheme also took longer than expected because of unexpected optimization issues. For instance, the H-table had to be redesigned through ranking and unranking process to avoid allocating unnecessary memory slots or the indexation of trees and generation of forests had to be designed so that it could be done in a single tree traversal.

Anyhow, we do have several ideas of models to implement to complete the "pipeline":

1. the DIAL [**Ferre2007**] model that consists of a weighted sum of the cost contributions of the sequence matching, the bracketing matching and the dihedral angles matching. Implementing the exact DIAL model would allow us to observe and quantify the difference between a sequence alignment and a tree alignment of the same RNA with the same scoring scheme.
2. a probabilist model based on dihedral angles

Part IV
References

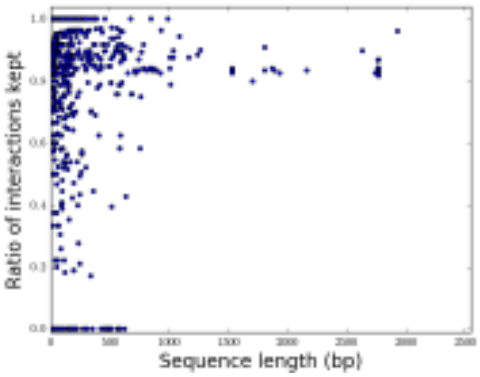
Part V
Appendix

A List of RNA PDB structures used

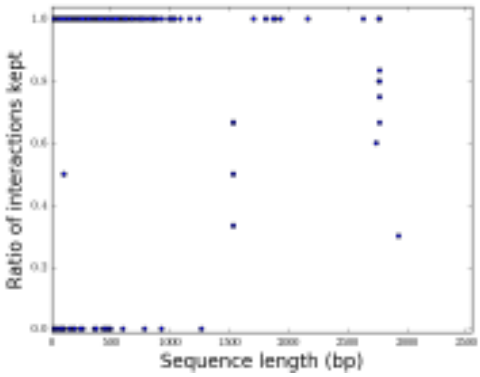
157D, 165D, 17RA, 1A3M, 1A4D, 1A51, 1A60, 1A9L, 1AFX, 1AJF, 1AJL, 1AJT, 1AJU, 1AKX, 1AL5, 1AM0, 1ANR, 1AQO, 1ARJ, 1ATO, 1ATV, 1ATW, 1B36, 1BAU, 1BGZ, 1BJ2, 1BN0, 1BVJ, 1BYJ, 1BZ2, 1BZ3, 1BZT, 1BZU, 1C0O, 1C2W, 1C2X, 1C4L, 1CQ5, 1CQL, 1CSL, 1D0T, 1D0U, 1D4R, 1DDY, 1DQF, 1DQH, 1DUH, 1DUQ, 1E4P, 1E95, 1EBQ, 1EBR, 1EBS, 1EHT, 1EHZ, 1EI2, 1EKA, 1EKD, 1ELH, 1ESH, 1ESY, 1ET4, 1EVV, 1F1T, 1F27, 1F5G, 1F5H, 1F5U, 1F6X, 1F6Z, 1F78, 1F79, 1F7F, 1F7G, 1F7H, 1F7I, 1F84, 1F85, 1F9L, 1FCW, 1FEQ, 1FG0, 1FHK, 1FIR, 1FL8, 1FMN, 1FOQ, 1FQZ, 1FUF, 1FYO, 1FYP, 1G2J, 1G3A, 1GID, 1GRZ, 1GUC, 1H0Q, 1H1K, 1HLX, 1HR2, 1HS1, 1HS2, 1HS3, 1HS4, 1HS8, 1HWQ, 1I3X, 1I3Y, 1I46, 1I4B, 1I4C, 1I7J, 1I9K, 1I9V, 1I9X, 1IDV, 1IE1, 1IE2, 1IK1, 1IK5, 1IKD, 1J4Y, 1J6S, 1J7T, 1J8G, 1J9H, 1JO7, 1JOX, 1JP0, 1JTJ, 1JTW, 1JU1, 1JU7, 1JUR, 1JWC, 1JZC, 1JZV, 1K2G, 1K4A, 1K4B, 1K5I, 1K6G, 1K6H, 1K8S, 1K9W, 1KAJ, 1KD3, 1KD4, 1KD5, 1KFO, 1KH6, 1KIS, 1KKA, 1KKS, 1KOC, 1KOD, 1KOS, 1KP7, 1KPD, 1KPY, 1KPZ, 1KXK, 1L1W, 1L2X, 1L3D, 1L3Z, 1L8V, 1LC4, 1LC6, 1LDZ, 1LMV, 1LNT, 1LPW, 1LU3, 1LUU, 1LUX, 1LVJ, 1M5L, 1M82, 1MDG, 1ME1, 1MFJ, 1MFK, 1MFY, 1MHK, 1MIS, 1MME, 1MNX, 1MSY, 1MT4, 1MUV, 1MV1, 1MV2, 1MV6, 1MWG, 1MWL, 1MY9, 1N53, 1N66, 1N8X, 1NA2, 1NBK, 1NBR, 1NBS, 1NC0, 1NEM, 1NJN, 1NJO, 1NLC, 1NTA, 1NTB, 1NUJ, 1NUV, 1NYI, 1NZ1, 1O15, 1O3Z, 1O9M, 1OQ0, 1OSU, 1OSW, 1OW9, 1P5M, 1P5N, 1P5O, 1P5P, 1P79, 1P9X, 1PBL, 1PBM, 1PBR, 1PJY, 1Q29, 1Q75, 1Q8N, 1Q93, 1Q96, 1Q9A, 1QBP, 1QC0, 1QC8, 1QCU, 1QD3, 1QES, 1QET, 1QWA, 1QWB, 1QZA, 1QZB, 1R2P, 1R3O, 1R4H, 1R7W, 1R7Z, 1RAU, 1RAW, 1RFR, 1RHT, 1RMN, 1RNA, 1RNG, 1RNK, 1ROQ, 1RRR, 1RXA, 1RXB, 1S2F, 1S34, 1S9L, 1S9S, 1SA9, 1SAQ, 1SCL, 1SDR, 1SLO, 1SLP, 1SY4, 1SYZ, 1SZY, 1T0D, 1T0E, 1T1O, 1T28, 1T4X, 1TBK, 1TFN, 1TJZ, 1TLR, 1TN1, 1TN2, 1TOB, 1TRA, 1TUT, 1TXS, 1U2A, 1U3K, 1U9S, 1UTS, 1UUD, 1UUI, 1UUU, 1VOP, 1VTQ, 1WKS, 1WTS, 1WTT, 1WVD, 1X8W, 1X9C, 1X9K, 1XHP, 1XJR, 1XP7, 1XPE, 1XPF, 1XSG, 1XSH, 1XST, 1XSU, 1XV0, 1XV6, 1XWP, 1XWU, 1Y0Q, 1Y26, 1Y27, 1Y3O, 1Y3S, 1Y6S, 1Y6T, 1Y73, 1Y90, 1Y95, 1Y99, 1YFG, 1YFV, 1YG3, 1YG4, 1YKQ, 1YKV, 1YLG, 1YLS, 1YMO, 1YN1, 1YN2, 1YNC, 1YNE, 1YNG, 1YRJ, 1YSV, 1YXP, 1YY0, 1YZD, 1Z2J, 1Z30, 1Z31, 1Z43, 1Z58, 1Z79, 1Z7F, 1ZC5, 1ZCI, 1ZEV, 1ZFT, 1ZFV, 1ZFX, 1ZIF, 1ZIG, 1ZIH, 1ZO3, 1ZX7, 1ZZ5, 205D, 255D, 259D, 280D, 283D, 28SP, 28SR, 299D, 2A04, 2A0P, 2A2E, 2A43, 2A64, 2A9L, 2ADT, 2AGN, 2AHT, 2AO5, 2AP0, 2AP5, 2AU4, 2AWE, 2AWQ, 2B57, 2B7G, 2B8R, 2B8S, 2BCY, 2BCZ, 2BE0, 2BEE, 2BJ2, 2CD1, 2CD3, 2CD5, 2CD6, 2CKY, 2D17, 2D18, 2D19, 2D1A, 2D1B, 2D2K, 2D2L, 2DD1, 2DD2, 2DD3, 2EES, 2EET, 2EEU, 2EEV, 2EEW, 2ES5, 2ESI, 2ESJ, 2ET3, 2ET4, 2ET5, 2ET8, 2EUY, 2EVY, 2F4S, 2F4T, 2F4U, 2F4X, 2F87, 2F88, 2FCX, 2FCY, 2FCZ, 2FD0, 2FDT, 2FEY, 2FGP, 2FQN, 2G1G, 2G1W, 2G32, 2G3S, 2G5K, 2G91, 2G92, 2G9C, 2GBH, 2GCS, 2GCV, 2GDI, 2GIO, 2GIP, 2GIS, 2GM0, 2GPM, 2GQ4, 2GQ5, 2GQ6, 2GQ7, 2GRB, 2GRW, 2GV3, 2GV4, 2GVO, 2H0W, 2H0X, 2H1M, 2H49, 2HEM, 2HNS, 2HO6, 2HO7, 2HOJ, 2HOK, 2HOL, 2HOM, 2HOO, 2HOP, 2HUA, 2I7E, 2I7Z, 2IL9, 2IRN, 2IRO, 2IXY, 2IXZ, 2JLT, 2JR4, 2JRG, 2JRQ, 2JSE, 2JSG, 2JTP, 2JUK, 2JWV, 2JXQ, 2JXS, 2JXV, 2JYF, 2JYH, 2JYJ, 2JYM, 2K3Z, 2K41, 2K4C, 2K5Z, 2K65, 2K66, 2K7E, 2K95, 2K96, 2KBP, 2KD4, 2KD8, 1337, 2KE6, 2KEZ, 2KF0, 2KGP, 2KHY, 2KOC, 2KP3, 2KPC, 2KPD, 2KPV, 2KRL, 2KRP, 2KRQ, 2KRV, 2KRW, 2KRY, 2KRZ, 2KTZ, 2KU0, 2KUR, 2KUU, 2KUV, 2KUW, 2KVN, 2KWG, 2KX8, 2KXM, 2KXZ, 2KY0, 2KY1, 2KY2, 2KYD, 2KYE, 2KZL, 2L1F, 2L1V, 2L2J, 2L3E, 2L5Z, 2L6I, 2L8C, 2L8F, 2L8H, 2L8U, 2L8W, 2L94, 2L9E, 2LA9, 2LAC, 2LBJ, 2LBK, 2LBL, 2LBQ, 2LBR, 2LC8, 2LDL, 2LDT, 2LDZ, 2LHP, 2LI4, 2LJJ, 2LK3, 2LKR, 2LP9, 2LPA, 2LPS, 2LPT, 2LQZ, 2LU0, 2LUB, 2LUN, 2LV0, 2LVY, 2LWK, 2LX1, 2M12, 2M18, 2M1O, 2M21, 2M22, 2M23, 2M24, 2M39, 2M4Q, 2M4W, 2M57, 2M58, 2M5U, 2M8K, 2MEQ, 2MER, 2MFD, 2MHI, 2MI0, 2MIS, 2MIY, 2MN0, 2MNC, 2MQT, 2MS5, 2MTJ, 2MTK, 2MXJ, 2MXK, 2MXL, 2N0R, 2N1Q, 2NOK, 2NPY, 2NPZ, 2O32, 2O33, 2O3V, 2O3W, 2O3X, 2O3Y, 2O43,

2O44, 2O45, 2O81, 2O83, 2OE5, 2OE6, 2OE8, 2OEU, 2OIJ, 2OIU, 2OIY, 2OJ0, 2OJ7, 2OJ8, 2OOM, 2OUE, 2P7F, 2P89, 2PCV, 2PCW, 2PN3, 2PN4, 2PN9, 2PWT, 2Q1O, 2Q1R, 2QBZ, 2QEK, 2QH2, 2QH3, 2QH4, 2QUS, 2QUW, 2QWY, 2R1S, 2R20, 2R21, 2R22, 2RLU, 2RN1, 2RO2, 2RP0, 2RP1, 2RPK, 2RPT, 2RQJ, 2RRC, 2TOB, 2TPK, 2TRA, 2U2A, 2V6W, 2V7R, 2VAL, 2VUQ, 2W89, 2X2Q, 2XEB, 2XNZ, 2XSL, 2Y95, 2YDH, 2YIE, 2YIF, 2Z9Q, 2ZY6, 300D, 301D, 310D, 333D, 353D, 354D, 357D, 359D, 361D, 364D, 377D, 379D, 387D, 397D, 3A3A, 3B31, 3B4A, 3B4B, 3B4C, 3B58, 3B5A, 3B5F, 3B5S, 3B91, 3BBI, 3BBK, 3BBM, 3BBV, 3BNL, 3BNN, 3BNO, 3BNP, 3BNQ, 3BNR, 3BNS, 3BNT, 3BWP, 3C44, 3CGP, 3CGQ, 3CGR, 3CGS, 3CJZ, 3CW5, 3CW6, 3CZW, 3D0M, 3D0U, 3D0X, 3D2G, 3D2V, 3D2X, 3DG0, 3DG2, 3DG4, 3DG5, 3DHS, 3DIG, 3DIL, 3DIM, 3DIO, 3DIQ, 3DIR, 3DIS, 3DIX, 3DIY, 3DIZ, 3DJ0, 3DJ2, 3DS7, 3DVV, 3DVZ, 3DW4, 3DW5, 3DW6, 3DW7, 3E5C, 3E5E, 3E5F, 3EOG, 3EOH, 3F2Q, 3F2T, 3F2W, 3F2X, 3F2Y, 3F30, 3F4E, 3F4G, 3F4H, 3FAR, 3FO4, 3FO6, 3FS0, 3FTM, 3FU2, 3FWO, 3G4M, 3G78, 3GAO, 3GCA, 3GER, 3GES, 3GLP, 3GM7, 3GOG, 3GOT, 3GS1, 3GS5, 3GS8, 3GVN, 3GX2, 3GX3, 3GX5, 3GX6, 3GX7, 3HGA, 3IBK, 3IGI, 3IQN, 3IQP, 3IQR, 3IVN, 3IZD, 3J28, 3J29, 3J2A, 3J2B, 3J2C, 3J2D, 3J2E, 3J2F, 3J2G, 3J2H, 3JQ4, 3JXQ, 3JXR, 3K1V, 3L0U, 3LA5, 3LOA, 3MEI, 3MIJ, 3MJ3, 3MJA, 3MJB, 3ND3, 3ND4, 3NJ6, 3NJ7, 3NPN, 3NPQ, 3OK2, 3OK4, 3OWI, 3OWW, 3OWZ, 3OX0, 3OXB, 3OXD, 3OXE, 3OXJ, 3OXM, 3P22, 3P4A, 3P4B, 3P4C, 3P4D, 3P59, 3PDR, 3PHP, 3Q3Z, 3Q50, 3Q51, 3R1C, 3R1D, 3R1E, 3R4F, 3RG5, 3RKF, 3S49, 3S4P, 3S7C, 3S8U, 3SD3, 3SJ2, 3SKI, 3SKL, 3SKR, 3SKT, 3SKW, 3SKZ, 3SLM, 3SLQ, 3SUH, 3SUX, 3SUY, 3SYW, 3SZX, 3T4B, 3TD0, 3TD1, 3TRA, 3TZR, 3VRS, 3WRU, 402D, 405D, 406D, 409D, 413D, 420D, 422D, 429D, 430D, 433D, 434D, 435D, 437D, 438D, 439D, 462D, 464D, 466D, 468D, 469D, 470D, 471D, 472D, 480D, 483D, 486D, 488D, 4A4R, 4A4S, 4A4T, 4A4U, 4AOB, 4B5R, 4C40, 4C4Q, 4CS1, 4DS6, 4E48, 4E58, 4E59, 4E5C, 4E6B, 4E8K, 4E8M, 4E8N, 4E8P, 4E8Q, 4E8R, 4E8T, 4E8V, 4EN5, 4ENA, 4ENB, 4ENC, 4ERJ, 4ERL, 4F8U, 4F8V, 4FAQ, 4FAR, 4FAU, 4FAW, 4FAX, 4FB0, 4FE5, 4FEJ, 4FEL, 4FEN, 4FEO, 4FEP, 4FNJ, 4FRG, 4FRN, 4G6P, 4G6R, 4G6S, 4GMA, 4GPW, 4GPX, 4GPY, 4GXY, 4IQS, 4J50, 4JAB, 4JAH, 4JF2, 4JIY, 4JRC, 4JRD, 4JRT, 4K27, 4K31, 4K32, 4KQY, 4KYY, 4KZ2, 4L81, 4LVV, 4LVW, 4LVX, 4LVY, 4LVZ, 4LW0, 4LX5, 4LX6, 4MCE, 4MCF, 4MEG, 4MEH, 4MGM, 4MGN, 4MS9, 4MSB, 4MSR, 4NFO, 4NFP, 4NFQ, 4NLF, 4NMG, 4NXH, 4NYA, 4NYB, 4NYC, 4NYD, 4NYG, 4O41, 4OJI, 4OQU, 4P20, 4P5J, 4P8Z, 4P95, 4P97, 4P9R, 4PCJ, 4PDQ, 4PHY, 4PLX, 4PQV, 4QJD, 4QJH, 4QK8, 4QK9, 4QKA, 4QLM, 4QLN, 4R0D, 4RGE, 4RGF, 4RJ1, 4RKV, 4RNE, 4RUM, 4TNA, 4TRA, 4TS0, 4TS2, 4TZX, 4TZY, 4U34, 4U35, 4U37, 4U38, 4WFL, 4WFM, 4XK0, 4XNR, 4Y1I, 4Y1J, 4Y1M, 4YAZ, 4YB0, 4YN6, 5A17, 5A18, 6TNA

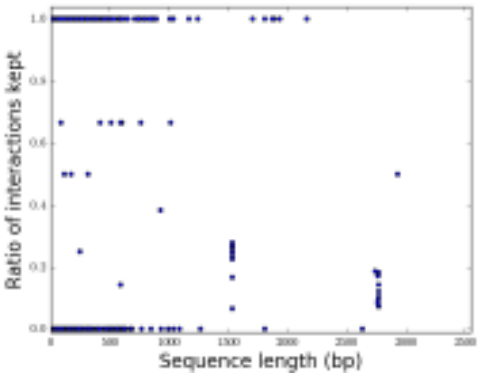
B Supplementary Figures



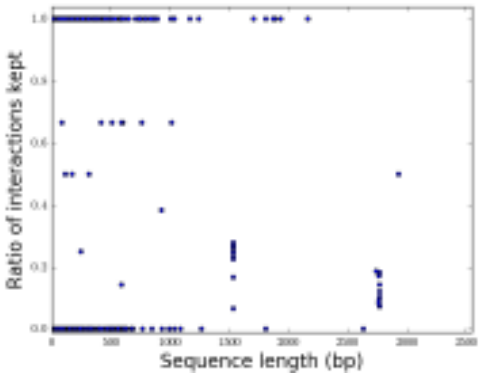
(a) Watson-Crick/Watson-Crick interactions



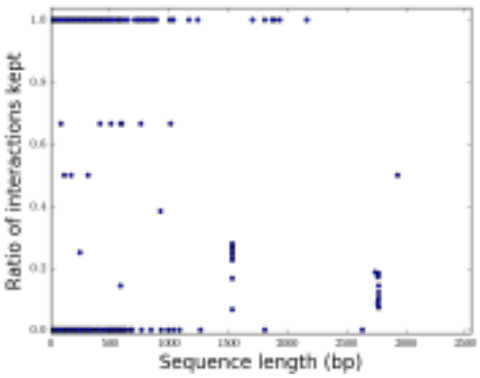
(b) Hoogsteen/Hoogsteen interactions



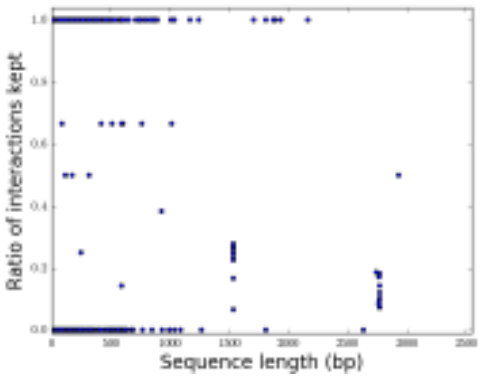
(c) Sugar/Sugar interactions



(d) Watson-Crick/Hoogsten interactions



(e) Watson-Crick/Sugar interactions



(f) Hoogsteen/Sugar interactions

Figure 34: Ratio of interactions kept after planarization of the annotations

C cpp-pprna main features

C.0.1 Tree structure

Kind	Status	Name	Purpose
Attribute	Private	`index`	node indexation 6.3.2
Attribute	Private	`parent`	pointer to parent node
Attribute	Private	`children`	vector of pointers to chidren nodes
Attribute	Private	`position`	integer representing the positions of the current base (pair) in the sequence
Method	Private	`_breadth_first_search`	returns a vector of pointer to nodes in breadth-first order
Constructor	Public	`Tree`	instanciates tree node from various data
Getter	Public	`get_children_forest`	returns the `forest` consisting of all node's children
Getter	Public	`get_position`	returns the `position` attribute
Getter	Public	`get_parent`	returns the `parent` attribute
Getter	Public	`get_children`	returns the `children` attribute
Getter	Public	`get_number_children`	returns the size of `children`
Setter	Public	`set_chilren_forest`	sets the forest consisting of all node's children
Setter	Public	`set_position`	sets the `position` attribute
Setter	Public	`set_parent`	sets the `parent` attribute
Setter	Public	`set_children`	sets the `children` attribute
Operator	Public	`!=`	compares two trees
Method	Public	`size`	returns the size of the tree, using a cache method to avoid repeated computations
Method	Public	`indexation`	indexes the tree and creates an `order_index_struct` with all the generated forests of the tree sorted by size in a single tree traversal (using `_breadth_first_search`)

Table 4: Main attributes and functions of the `tree` structure

D Alignment hierarchy

Table 5: The ALIGN hierarchy complexity summary, for details see [**Blin2010**]

$A \times B \to C$	Model I	Model II	Model III
NEST × NEST → NEST	$\mathcal{O}(n^4)$ [**Jiang1995**]	$\mathcal{O}(n^4)$ [**Blin2006**]	$\mathcal{O}(n^4)$ [**Blin2006**]
NEST × NEST → CROS	$\mathcal{O}(n^3 \log n)$ [**Klein1998**]	NP-hard [**Blin2006**]	
NEST × NEST → UNLIM	$\mathcal{O}(n^3 \log n)$ [**Klein1998**]	NP-hard [**Lin2002**]	NP-hard [**Blin2003**]
CROS × NEST → CROS	$\mathcal{O}(n^3 \log n)$ [**Ma2002**]	NP-hard [**Blin2006**]	
CROS × NEST → UNLIM	$\mathcal{O}(n^3 \log n)$ [**Ma2002**]	NP-hard [**Evans1999**]	MAX SNP-hard [**Jiang2002**]
CROS × CROS → CROS	NP-hard [**Ma2002**]	NP-hard [**Blin2006**]	
CROS × CROS → UNLIM	NP-hard [**Ma2002**]	NP-hard [**Evans1999**]	MAX SNP-hard [**Jiang2002**]
UNLIM × NEST → UNLIM	$\mathcal{O}(n^3 \log n)$ [**Blin2006**]	NP-hard [**Evans1999**]	MAX SNP-hard [**Jiang2002**]
UNLIM × CROS → UNLIM	NP-hard [**Ma2002**]	NP-hard [**Evans1999**]	MAX SNP-hard [**Jiang2002**]
UNLIM × UNLIM → UNLIM	NP-hard [**Ma2002**]	NP-hard [**Evans1999**]	MAX SNP-hard [**Jiang2002**]