DARN! A weighted constraint solver for RNA motif localization

Matthias Zytnicki, Christine Gaspin and Thomas Schiex

October 12, 2007

Abstract

Following recent discoveries about the important roles of non-coding RNAs (ncRNAs) in the cellular machinery, there is now great interest in identifying new occurrences of ncRNAs in available genomic sequences.

In this paper, we show how the problem of finding new occurrences of characterized ncRNAs can be modeled as the problem of finding all locally-optimal solutions of a weighted constraint network using dedicated weighted global constraints, encapsulating pattern-matching algorithms and data structures.

This is embodied in DARN!, a software tool for ncRNA localization, which, compared to existing pattern-matching based tools, offers additional expressivity (such as enabling RNA-RNA interactions to be described) and improved specificity (through the exploitation of scores and local optimality) without compromises in CPU efficiency. This is demonstrated on the actual search for tRNAs and H/ACA sRNA on different genomes.

1 Introduction

Identifying the gene catalogue contained in complete genome sequences is a difficult task in genome sequencing projects. Until now, efforts in the automatic identification of genes were essentially focused on locating protein. In addition to proteins, non-coding RNA (ncRNA) genes are functionally important and their role, in controlling a growing list of processes in the cell, is now well established. More than 503 families are known and listed in the RFAM database (http://www.sanger.ac.uk/Software/Rfam/, [8]). Since 2000, the increasing number and importance of ncRNAs has led to new interest in their computer-based identification as we learn more about them. However, the features used for finding protein genes are essentially ineffective for identifying ncRNA genes. We previously proposed a formulation of the ncRNA localization problem in the context of constraint networks [16] which offers a way of representing and solving the problem of searching for occurrences of ncRNA potentially in interaction with other molecules.

In this paper we propose extending the model and the algorithms with the aim of providing biologists with improved means of exploring candidate solutions. We extend the previous developments in the context of weighted constraint networks in order to rank the potentially large number of solutions, as well as introducing the notion of the locally-optimal solution in order to remove redundant solutions. The tool which was implemented, DARN! —which means *Détection d'ARN*, or "RNA localization" in French—, is used to search for new members of the H/ACA box sRNA family in a recently sequenced genome, *Thermococcus kodakarensis*.

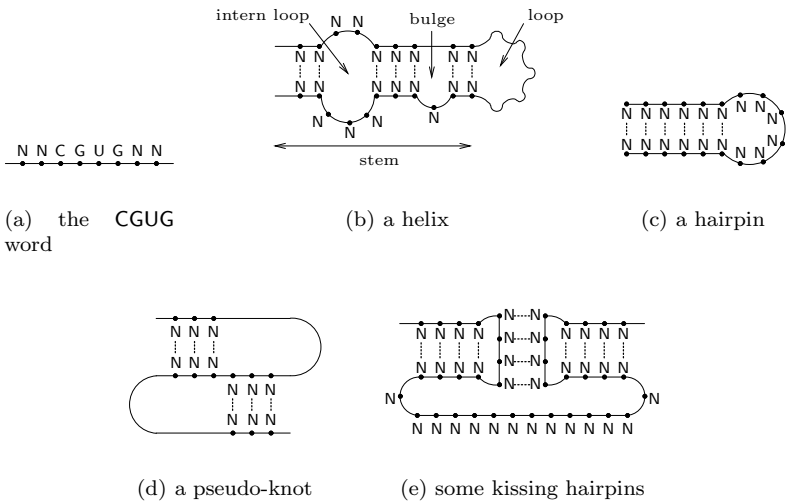
The rest of the paper is organized as follows. Section 2 gives the biological background useful for understanding the remainder of the paper. Section 3 gives a short overview of the formalisms and tools used so far to answer the question of ncRNA localization. Section 4 presents the weighted CSP framework, how ncRNA localization can be modeled within this framework, and the weighted constraints of the tool. Section 5 presents the underlying algorithms of the weighted constraints, a new way of selecting only the most promising candidates, and several additional mechanisms, which make it possible to handle very large sequences. Section 6 gives the results of our tool applied to the search for tRNAs and sRNAs on different genomes. Finally, Section 7 concludes the paper by discussing future developments.

2 Biological background

A ncRNA gene is a functional molecule composed of smaller molecules, called *nucleotides* or *bases* for simplification, linked together by covalent bonds. There are four types of these nucleotides, commonly identified by a single letter: A for adenosine, G for guanosine, U for uridine and C for cytidine. Thus, an RNA is represented as a word built from the four letters A, U, G and C. This sequence (cf. figure 1(a)) defines what is called the primary structure of the RNA molecule. Note that N is traditionally used to denote an arbitrary nucleotide (A, U, G or C).

RNA molecules are single-stranded molecules that have the ability to fold back on themselves by developing interactions between bases, forming pairs. The most frequently interacting pairs are the so-called Watson-Crick pairs where a G interacts with a C, or a U interacts with an A. These pairs are formed from so-called *complementary bases*, G–C pairs being stronger (or more stable) than A–U ones. Less frequently, one may also observe so-called wobble pairs where a G interacts with a U. A sequence of such interactions forms a structure called a *helix* (cf. figure 1(b)). Helices are a fundamental structural element in ncRNA genes and are the basis for more complex structures such as *hairpins* (a helix with a loop, cf. 1(c)), *pseudo-knots* (cf. 1(d)), *kissing hairpins* (cf. 1(e)), etc. Helices can also appear between two different interacting RNA molecules, forming in this case a *duplex*.

The set of interactions is often displayed by a graph where vertices represent bases and edges represent either covalent bonds linking successive nucleotides



(a) the CGUG word (b) a helix (c) a hairpin

(d) a pseudo-knot (e) some kissing hairpins

Figure 1: Some elements of structure. N stands for any nucleotide.

or interacting base pairs. This representation is usually called the molecule's secondary structure.

The actual spatial organization of the entire nucleotide chain at the atomic level, called the tertiary structure, is the relevant level of organization for biological function. The tertiary structure is essential for the function of the ncRNA, and we call a *family* the set of ncRNAs that have a common biological function. However, due to the difficulty of determining tertiary ncRNA structures, the secondary structure is commonly used as a simplified model for most analyses. In this paper, we define the *signature* of a gene family as the set of conserved elements either in the sequence or the secondary structure, including possible duplexes with other ncRNA molecules.

To define the signature of a family, a traditional approach is to rely on an alignment of the sequences of the family's different ncRNA sequences. The alignment brings to light conserved motifs and the common structural characteristics that form the family signature.

To make things more concrete, we will use the H/ACA box sRNA gene family as an example throughout the paper. H/ACA box sRNAs have a specific structure, and interact with other specific ncRNA genes (ribosomal RNAs) in order to chemically modify specific bases in their sequence. To target precisely the bases which should be modified, a duplex between the H/ACA box sRNA gene and the ribosomal RNA is formed. Traditionally, the sequence that contains the modified ncRNA is called the *target sequence*, whereas the other one is called the *main sequence*.

In the group of Archaeal organisms, H/ACA box sRNAs contain one or more hairpins connected by single-stranded regions. Each hairpin exhibits similar

characteristics that we have identified by aligning hairpins contained in known H/ACA box sRNA genes for *Pyrococcus furiosus*, *Pyrococcus abyssi*, *Pyrococcus horikoshii*, *Methanococcus jannaschii* and *Archaeoglobus fulgidus* genomes. Figure 2(a) shows the alignment of different sRNAs of this family for different organisms (due to lack of space, only some sRNAs are shown). The different conserved features in this RNA family can be extracted from this alignment.

First, three conserved motifs (or subsequences) can be identified: a conserved GA (between positions x_7 and x_8), a conserved {A,G}UGA motif (between positions x_9 and x_{10}) and a final conserved ACA motif (between positions x_{17} and x_{18}). Next, the presence of a succession of complementary bases in the regions delimited by x_1 and x_2 on one side, and x_{15} and x_{16} on the other side indicates a conserved helix, involving around seven base pairs. These two interacting regions also have a high ratio of G and C bases, which is usual in helices, for stability reasons. A similar high percentage of G and C bases appears in the regions delimited by x_5 and x_6, and by x_{11} and x_{12} respectively. Finally, the RNA is known to interact with a target sequence through two regions identified here by positions x_3 and x_4, and by positions x_{13} and x_{14} respectively. These two regions form a duplex with the target sequence. The corresponding regions in the target sequence are identified by positions y_6 and y_5, and by y_2 and y_1 respectively. These two regions delimit a two-base long region with a U at the second position, represented here by the conserved UN motif. All these regions appear in a given order and not too far away from each other (the most variable length appearing between x_8 and x_9).

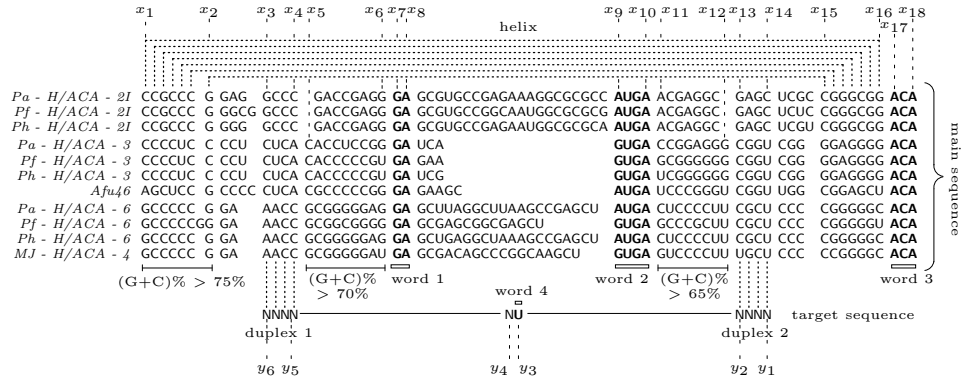
Clearly, the signature of a family can be expressed as a collection of properties that must be satisfied by a set of regions occurring on a sequence.

Figure 2(b) shows the corresponding secondary structure for a solution appearing in the genome of *P. abyssi*, together with the duplex formed with the target sequence.

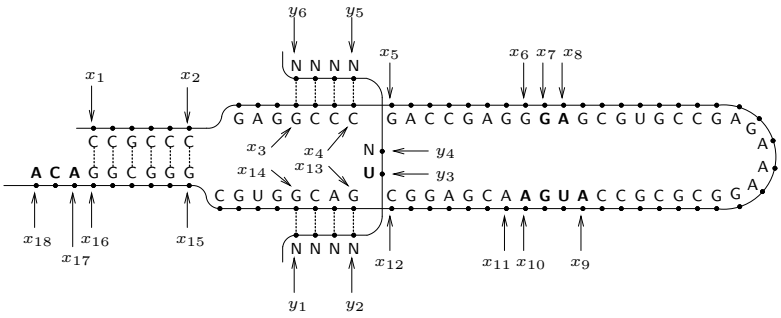
Given the signature of a family, the problem we are interested in involves searching for new members of a gene family in existing genomes, where these members are in fact the set of regions appearing in the genome which satisfy the signature properties. Genomic sequences are themselves long texts composed of nucleotides A, C, G and T, the latter corresponding to U in ncRNA genes. They can be thousand of bases long for the simplest organisms up to several hundred million bases for the more complex ones.

3 Searching ncRNA genes in genomic sequences

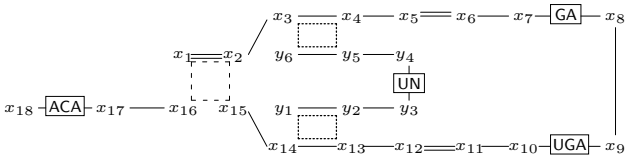
To be able to solve the problem of searching for occurrences of a gene signature in a genomic sequence, one must first decide how to formally represent such a signature. Among the proposed formalisms, one of the most famous ones uses probabilistic information in a stochastic context-free grammar that describes the structures [7] and searches for them using dynamic programming based parsers [15]. However, some complex ncRNA families cannot be described within this formalism and [17] showed that only NP-hard formalisms may correctly describe



(a) an alignment



(b) a solution, on *Pyrococcus abyssi*



(c) a signature represented with cost functions: full lines are spacers, dashed rectangles are helices, double lines are cost functions on the (G+C)%, boxes are patterns and dotted rectangles are duplexes

Figure 2: Different representations of the H/ACA box sRNA family.

them. Also, such models usually involve a very large number of probabilistic parameters which may require a large amount of the family's known RNA genes to be correctly estimated. This may also lead to a lack of generality in the model estimated, which may be specific to some organisms only.

Another approach defines a signature as a set of interrelated motifs. Occurrences of the signature are sought using pattern-matching techniques and an exhaustive tree search. Such programs include RnaMot [9], RnaBob [6], PatScan [5], Palingol [3] and RnaMotif [12]. Although these programs allow pseudo-knots to be represented, they have very variable efficiencies. This favors a CSP model of the problem and such work was done in [16] where the combinatorial aspects are clearly separated from the pattern-matching aspects. A signature is modeled as a weighted constraint network. This model captures the combinatorial features of the problem while the weighted constraints use pattern-matching techniques to enhance their efficiency. This combination offers an elegant and simple way of describing several ncRNAs in interaction and a general-purpose efficient algorithm to search for all their occurrences.

In practice, the ncRNA localization problem has two main characteristics that should be addressed carefully. The first one is that the size of the sequences is usually as large as several hundred million nucleotides. This means that extra care should be taken with the time and space complexities of the algorithms. Second, some queries, depending on the specificity of the signature, give hundreds of thousands of solutions and, in practice, it is impossible for the user to exploit this huge number of solutions. Obviously, by looking more carefully at the solutions, some are better than others and it would be useful to give only the best ones to the user. This is why, in this paper, we have used the weighted CSP formalism —an extension of the CSP framework— to solve the ncRNA detection problem. With far fewer parameters than stochastic context-free grammars, weighted CSP formalism seems to have enough expressive power to properly and efficiently characterize ncRNA families.

4 Formalization

4.1 Weighted CSPs

A WCSP [10] is a tuple $\mathcal{P} = \langle \mathcal{X}, \mathcal{D}, \mathcal{C}, k \rangle$, where:

- $\mathcal{X} = \{x_1, \ldots, x_n\}$ is a set of n variables,
- $\mathcal{D} = \{D(x_1), \ldots, D(x_n)\}$ is the set of the finite domains for each variable and the size of the largest one is d,
- $\mathcal{C}$ is the set of e cost functions, where a cost function c is a function that involves a set of variables $var(c) = x_{i_1} \times x_{i_2} \times \ldots \times x_{i_r}$, and associates an integer called *cost* or *penalty* with every assignment of these variables. In other words, it is a function from $D(x_{i_1}) \times D(x_{i_2}) \times \ldots \times D(x_{i_r})$ to $E = [0..k] \subseteq \mathbb{N}$,

- k is the highest possible cost, representing an inconsistency.

Moreover, we assume the existence of a unary cost function $c : D(x_i) \rightarrow E$, denoted c_i, for every variable x_i (if no such cost function exists, we define $\forall v_i \in D(x_i), c_i(v_i) = 0$). We also define a new cost function, $c_\varnothing \in E$, which does not involve any variable, and is a constant cost payed by all assignments of the variables.

The *projection* of a complete assignment (i.e. an assignment involving all the variables) $t = (x_1 \leftarrow v_1, \ldots, x_n \leftarrow v_n)$ on $X = (x_{i_1}, \ldots, x_{i_m})$ is the partial assignment $t[X] = (x_{i_1} \leftarrow v_{i_1}, \ldots, x_{i_m} \leftarrow v_{i_m})$. Given a cost function c and an assignment t, $c(t[var(c)]) = k$ means that the cost function forbids the corresponding assignment. A cost of 0 means that the assignment is fully accepted by the cost function. Another cost means that the assignment is permitted by the cost function, although not preferred. The cost of a complete assignment $t = (x_1 \leftarrow v_1, \ldots, x_n \leftarrow v_n)$, denoted $\mathcal{V}(t)$, is the sum over all the cost functions:

$$\mathcal{V}(t) = \bigoplus_{c \in \mathcal{C}} c(t[var(c)])$$

where $\oplus$ is the bounded sum: $\forall (a, b) \in E^2, a \oplus b = \min\{a + b, k\}$. A *solution* is a complete assignment, whose cost is less that k.

Note that, if all the costs yielded by a cost function are in $\{0, k\}$, then every assignment of the variables involved is either fully accepted or forbidden, so it can be considered as a hard constraint. Similarly, if $k = 1$, then WCSP reduces to classic CSP.

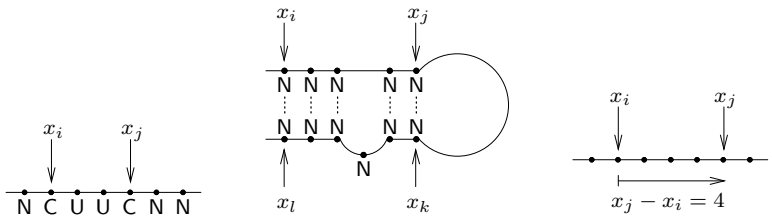
4.2 Model

In order to find ncRNAs, we can build a weighted constraint network that scans a new genome, and detects the regions of the genome where the signature elements are present and correctly positioned. So, the model used is the following:

- variables of the weighted CSP are the positions of the signature elements in the main or target sequences;
- the size of the domains are thus the size of the sequences;
- cost functions enforce the presence of the signature elements, between the positions taken by the variables involved.

When the signature has been specified using cost functions (like in figure 2(c)), a sequence (or two sequences, if there is a target sequence) is provided. Each solution is the set of positions, where each position correctly delimits a bound of an signature element, as described in figure 2(b). A solution is thus a potential biological candidate. Of course, *all* the solutions to the problem (and not only the solution with minimum cost) should be given.

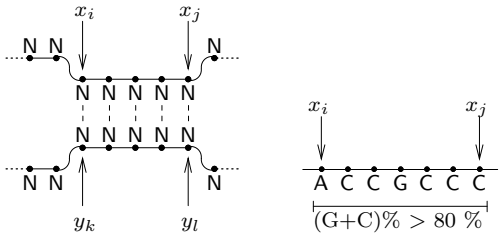
Several weighted global constraints are provided in order to model the signature elements. Each cost function requires a set of parameters, which describe



(a) `pattern`(word = CUUC, errors = 0)(x_i, x_j)

(b) a `helix` cost function

(c) a `spacer` cost function



(d) a `duplex` cost function

(e) a `composition` cost function

Figure 3: Available cost functions.

inherent properties (such as the number of nucleotides in a helix). The variables involved in a cost function give the bounds of the signature element. The cost incurred by the cost function represents a penalty, if the region delimited by the variables does not perfectly match the structural element.

The available soft global cost functions are:

- `pattern`[word, errors](x_i, x_j). For example, we can see that H/ACA box sRNAs usually end with the nucleotides ACA. The fact that some words should be found in any H/ACA box sRNA can be enforced by the cost function `pattern`. This cost function makes sure that the word word starts at position x_i and ends at x_j. The cost computed by this cost function is the Levenshtein distance, i.e. the minimum number of insertions / deletions / substitutions that should be applied to the word word, in order to get the subsequence between the positions x_i and x_j. This cost should not be greater than errors. The parameter word can be given using the four nucleotides A, C, G and U (or T), but also using ambiguous nucleotides. For example, R stands for A or G, and N stands for any nucleotide, following IUPAC notations.

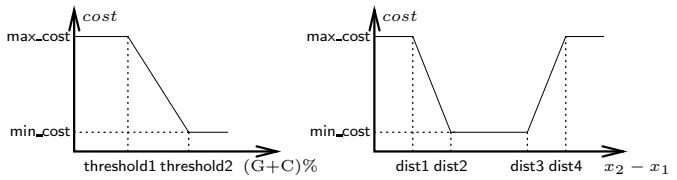
- `helix`[stem_min, stem_max, loop_min, loop_max, errors, indels, wobble] (x_i, x_j, x_k, x_l). It can also be noted that every H/ACA box sRNA contains a helix, and that its size ranges from 8 to 10 nucleotides, with a few possible mismatches. This may be modeled by the cost function `helix`. This cost function states that the nucleotides between the positions x_i and x_j should be able to interact with the nucleotides between positions x_k and x_l. The size of the helix should not be less than stem_min and not more than stem_max. The size of the loop should range from loop_max to loop_min. The cost incurred by this cost function is the Hamming distance between the two strands of the helix, which is the minimum number of substitutions that should be applied to one strand, in order to have a perfect match between the two strands. It should not be greater than errors. Two flags can also be set: if indels is set to true, then the given cost is the Levenshtein distance between the two strands (instead of the Hamming distance). If wobble is set to true, then G–U interactions are also accepted.

- `spacer`[dist1, dist2, dist3, dist4, min_cost, max_cost](x_i, x_j). In the sRNAs, the number of nucleotides between the first and the second word seems to range from 3 to 21 nucleotides. The distance between two different elements of a structure is usually called a *spacer*, possibly modeled by the cost function `spacer`, which gives the relative distance between positions x_i and x_j. The cost incurred by this cost function is a stepwise linear function, described in figure 4(b). If $x_j - x_i$ is less than dist1 or greater than dist4, then the cost function is inconsistent.

- `duplex`[errors, wobble](x_i, x_j, y_k, y_l). The whole region known as *duplex* interacts with another given RNA called the *target sequence*. So, the `duplex` cost function states that a duplex exists between positions x_i and x_j in the main sequence, and y_k, y_l in the target sequence. The cost of this cost function is the Levenshtein distance between the two strands. This cost should not be greater than errors. The wobble parameter has the usual meaning.

- `composition`[nucleotides, relation, threshold1, threshold2, min_cost, max_cost](x_i, x_j). Looking more carefully at the helix, we can see that the number of nucleotides G and C is much higher than the number of nucleotides A and U. This bias towards the nucleotides G and C, linked to the molecule's stability can be modeled by the `composition` cost function. This cost function ensures that the percentage of the two nucleotides specified in nucleotides is greater or less (depending on the value of relation) than a given threshold. The value of nucleotides is usually GC or AU. The cost profile of this cost function is described in figure 4(a), where relation is $\geq$. In this example, if the percentage of nucleotides is less than threshold1, then the cost function is inconsistent.

- Two last cost functions have been implemented: `helix2`, which adds a penalty to shorter helices, and `hairpin`, which can be useful for describing



(a) The cost profile of a `composition` cost function.

(b) The cost profile of a `spacer` cost function.

Figure 4: Cost profiles of some cost functions.

hairpins accurately. Since they are quite similar to `helix`, we will not describe them any further.

5 WCSP algorithms

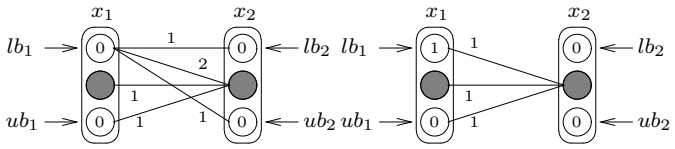
5.1 Algorithms for global cost functions

Notations. To accurately describe the algorithms used in our tool, we will first introduce some notations:

- The domain of each variable x_i (resp. y_i) is a set of possible positions in the main sequence (resp. target sequence). Thus, we can define the *lower bound*, lb_i, of this domain, and the *upper bound*, ub_i.
- The main sequence can be considered as a vector of nucleotides. We will denote this sequence S, and $S[a]$ the a-th nucleotide of the sequence. $S[a..b]$ will represent the subsequence of S composed of the nucleotides $S[a], S[a+1], \ldots, S[b]$. The target sequence is denoted T.

In order to solve the problem of ncRNA localization using a weighted constraint network, we implemented a depth-first branch-and-bound algorithm using binary branching. But since the size of the domain can be as large as several million nucleotides, extra care should be taken about time and space complexities while solving the problem. Thus, at each node of the exploration tree, we maintain *weighted bound arc consistency* (BAC*, cf. [18]), or weighted bound arc consistency with $\varnothing$ *inverse consistency* (BAC* with $\varnothing$IC) for `spacer` cost functions.

BAC* is an extension of 2B-consistency [11] for weighted CSP. It basically propagates costs from binary cost functions (or any cost function with arity higher than 1) to unary cost functions, so that only the unary costs of the bounds of the domain increase. $\varnothing$IC also performs propagations from binary cost functions to $c_\varnothing$. An example of the enforcement of BAC* is given in figure 5.



(a) a WCSP instance

(b) after the enforcement of BAC*

Figure 5: Enforcing BAC*.

Figure 5(a) describes a simple WCSP instance, containing two variables (x_1 and x_2), a binary cost function c_{12} (the binary cost of a pair is written on the edge that joins them; if none is given, it is zero) and two unary cost functions (the cost is written in the circle). Obviously, for any assignment of x_2, assigning x_1 to lb_1 will give a cost of at least 1. BAC* makes this observation more explicit by *projecting* a cost of 1 from the binary cost functions to the unary cost (the result is shown in figure 5(b)). This operation can be broken down into three elementary ones: finding the minimum cost of the binary cost function, when $x_1 = lb_1$; subtracting this cost amount from the binary costs; adding this cost to the unary cost function. As a result of this projection, there exists a value w in the domain of x_2 where $c_{12}(lb_1, w) = 0$. Since this value w is called the *support* of lb_1 the previous projection is usually implemented in a procedure called getSupport, and, in fact, enforcing BAC* on a WCSP instance ultimately reduces to finding supports.

Note that in BAC*, supports for values which are not bounds of the domain are not sought. This property makes BAC* well-suited to the RNA localization problem: as it saves time and space, it can be enforced on very large instances of the problem, whereas the state-of-the-art local property AC* makes the program abort for memory reasons when the sequence is greater than a few dozen thousand of nucleotides [18] (remember that a chromosome may contain millions of nucleotides).

We will now describe the function getSupport, which finds the supports for BAC*, for each cost function. However, due to lack of space, we will only give details of the `pattern` cost function.

The `pattern` cost function For the `pattern` cost function, the support of, say, lb_i, is the position v_j that minimizes the Levenshtein distance between word and all the subsequences of S that start at lb_i, and end at v_j. Finding this position v_i is a typical approximate string matching problem. From the many possible ways of solving it, we chose the following one, described in detail in [13].

We first select the nucleotide $S[lb_i]$ at position lb_i, and generate all the possible matches of this 1-letter word with word. Then, we take the next nucleotide

$S[lb_i + 1]$, and generate all the possible matches of $S[lb_i..lb_i + 1]$ with word, using the matches found between $S[lb_i]$ and word. We continue selecting a letter, and generating the possible matches, until the position of the selected letter is greater than ub_j. At this time, all the work is done, and the support of lb_i is the lowest score given by a match, where the last selected letter was not before the position lb_j (since lb_j is the first position where the pattern may end).

At first glance, it seems that the number of generated matches may exponentially grow. However, it is possible to use a dynamic programming algorithm that keeps the time and space complexities reasonably low. So we used a vector *vector* of $|\mathsf{word}| + 1$ integers, ranging from 0 to $|\mathsf{word}|$. During the search for the support, for example when the letter $S[a]$ is selected, each cell $vector[b]$ stores the number of errors for the best match between $\mathsf{word}[1..b]$ and $S[lb_i..a]$.

The detailed description of getSupport is as follows. The main loop at line **2** reads the nucleotides $S[a], a \in [lb_i..ub_j]$ of interest in the sequence. The inner loop at line **5** updates the *vector* data structure, and computes the set of possible best matches between word and $S[lb_i..a]$. If the condition **12** is true, this means that the selected letter $S[a]$ is in the domain of x_j, and thus the subsequence can end here. So, we have to store the current score (given by $vector[|\mathsf{word}|]$), since a can be a support of lb_i. The function finally returns the support of lb_i, as well as the minimum cost incurred by the cost function.

The update of *vector*, computed on line **11**, works as follows. Let us suppose we select a letter $S[a]$. $vector[b]$ may be (1) the best match between $S[lb_i..a]$ and $\mathsf{word}[1..b-1]$ with an insertion of the nucleotide $\mathsf{word}[b]$, or (2) the best match between $S[lb_i..a-1]$ and $\mathsf{word}[1..b]$ with an insertion of the nucleotide $S[a]$, or (3) the best match between $S[lb_i..a-1]$ and $\mathsf{word}[1..b-1]$ plus the cost of matching $S[a]$ and $\mathsf{word}[b]$. In our implementation, the cost the match between $S[a]$ and $\mathsf{word}[b]$ is computed line **6**. The best match between a suffix of $S[lb_i..a]$ and $\mathsf{word}[1..b-1]$ is stored in $vector[b-1]$ (cf. line **7**), the best match between a suffix of $S[lb_i..a-1]$ and $\mathsf{word}[1..b]$ is stored in $vector[b]$, just after $S[a-1]$ has been selected (cf. line **8**), and the best match between a suffix of $S[lb_i..a-1]$ and $\mathsf{word}[1..b-1]$ is stored in $vector[b-1]$ just before $S[a-1]$ has been selected (cf. line **9**). This latter value is rewritten during the induction process, and a new variable, *prev* is needed, that stores $vector[b-1]$, just before its update (on lines **3** and **10**).

This induction process has two base cases. The first one deals with the values of *vector*, when no nucleotide in S has been selected. In this case, $vector[b]$ is the cost of matching $\mathsf{word}[1..b]$ with a 0-letter word, so $vector[b]$ gets the value b (line **1**). The second base case deals with the values of $vector[0]$ during the run of an algorithm. When the letter $S[a]$ is selected, $vector[0]$ is the cost of matching $S[lb_i..a]$ with a 0-letter word, so $vector[0]$ gets the value $a - lb_i + 1$ (line **4**).

In our implementation, we used two vectors. The first one reads the sequence and the word from left to right, and thus updates the supports of the bounds of x_i. The second one reads the sequence and the word from right to left, and updates the supports of the bounds of x_j.

Function getSupport(lb_i): (position × cost)

Input : a bound of x_i
Output: the support of this bound
$minCost \leftarrow k$; $support \leftarrow -1$;
1 **foreach** $b \in [0..|\mathsf{word}|]$ **do** $vector[b] \leftarrow b$;
2 **foreach** $a \in [lb_i..ub_j]$ **do**
3 $prev \leftarrow vector[0]$;
4 $vector[0] \leftarrow a - lb_i + 1$;
5 **foreach** $b \in [1..|\mathsf{word}|]$ **do**
6 $match \leftarrow \begin{cases} 0 & \text{if } S[a] = \mathsf{word}[b], \\ 1 & \text{otherwise.} \end{cases}$;
7 $val_1 \leftarrow vector[b-1] + 1$;
8 $val_2 \leftarrow vector[b] + 1$;
9 $val_3 \leftarrow prev + match$;
10 $prev \leftarrow vector[b]$;
11 $vector[b] \leftarrow \min\{val_1, val_2, val_3\}$;
12 **if** $((a \geq lb_j) \wedge (vector[|\mathsf{word}|] < minCost)$ **then**
$minCost \leftarrow vector[|\mathsf{word}|]$;
$support \leftarrow a$;
return $(support, minCost)$;

The `helix` cost function We shall suppose here that we are also looking for the support of lb_i, the lower bound of x_i. The algorithm first locates all possible positions of the three other variables, given current domains and the stem_min, stem_max, loop_min and loop_max parameters. For each possible position of the variables, it computes the cost of the cost function. Notice here that the time complexity of the function getSupport heavily relies on the parameters of the cost function, since the number of possible positions of the variables increases if the differences stem_max − stem_min and loop_max − loop_min grow.

The cost incurred by the cost function can be computed by two different algorithms, depending on the value of indels. If insertions and deletions are not allowed, then the cost is the Hamming distance, which can be computed very easily in time and space linear to stem_max: if the strands do not have the same size, then the helix is not accepted; otherwise, we take the leftmost nucleotide of the first strand, and the rightmost nucleotide of the second strand, if they do not match, then the score increases by one, and we repeat the same test by reading the first strand from left to right, and the second strand from right to left.

If insertions and deletions are allowed, we use an alternative form of the Needleman-Wunsch algorithm [14] where only the bands around the diagonal are explored in the dynamic programming matrix. This optimization comes from the observation that a cell which is on the diagonal can represent the score of an alignment which contains only substitutions (and no insertion nor

deletion), whereas a cell which is not on the diagonal cannot. Thus, a cell which is on the i-th diagonal contains at least i deletions or insertions, and yields a cost of at least i. Since the maximum allowed number of errors is errors, then only the $2 \times \mathsf{errors} + 1$ bands around the diagonal should be filled. This Needleman-Wunsch-like algorithm takes time and space proportional to $\mathsf{stem\_max} \times \mathsf{errors}$, and the function getSupport runs in time proportional to errors, loop_max and $\mathsf{stem\_max}^2$.

The `spacer` cost function Since this cost function is a piecewise linear function, BAC* and ∅IC can be enforced in time proportional to d. This is a direct consequence of the corollary 1 in [18].

The `duplex` cost function The simple algorithm used for `helix` would be impractical here, since we may have no bound on the positions of the variables in the target sequence. To avoid this, we designed a new algorithm, that can get the supports more efficiently.

No propagation occurs until the variables x_i and x_j are assigned. When both variables have been assigned, then the problem reduces to finding a subsequence in the target sequence that matches the subsequence in the main sequence between positions x_i and x_j, which is very similar to the `pattern` cost function. In order to accelerate the search in the potentially long target sequence, we used an *enhanced suffix array* [1], that stores the set of all the suffixes of the target sequence, lexicographically ordered. Equipped with some additional information, the enhanced suffix array can be used like a suffix tree (and thus, provided that no error is allowed, a word can be found in time proportional to the size of this word, whatever the size of the sequence). Using the array structure, it can be encoded in less space than a suffix tree, and yields less cache misses.

To find the best matches in the suffix array, we also developed an original depth-first branch-and-bound algorithm on the enhanced suffix array that makes it possible to explore only the most promising regions of the target sequence. As a result, the time complexity of the cost function is independent of the size of the target sequence, although exponential to the size of the subsequence of the target sequence, and the number of allowed errors [19].

The `composition` cost function The `composition` cost function remains idle until one variable is assigned. If x_j is assigned, then scanning the nucleotides between lb_i and x_j to find whether x_j actually is a support of lb_i is straightforward.

5.2 Removing non-locally optimal solutions

For one biological candidate, several equivalent solutions may exist and are predicted by our approach. If an ncRNA occurrence is found at a given position in the sequence, it is easy to see that several other solutions obtained by slight modifications of the position of a structural element may also exist. To avoid

reporting all these phantom solutions, we introduced an additional mechanism that removes all the unwanted solutions. This is an important mechanism, since the number of irrelevant solutions may grow exponentially when the number of variables increases. To do so, we defined the *dominance* between solutions. We say that a solution A *dominates* the solution B if:

- A and B overlap on the main sequence, and $\mathcal{V}(A) < \mathcal{V}(B)$, or
- A and B overlap on the main sequence, and $\mathcal{V}(A) = \mathcal{V}(B)$, and A is the leftmost solution.

In other words, a solution A dominates B if they overlap and the cost of A is less than the cost of B; if the costs are the same, the leftmost solution is chosen for domination. A locally-optimal solution is a solution that is not dominated by any other other solution. Only these solutions are displayed to the user. This mechanism proved to be efficient and only gives the relevant solutions.

To compute the set of locally-optimal solutions, we have to check whether each newly found solution A is dominated, and whether any already found solution B_i is dominated by A. Since both problem are quite similar, we will only give details of how to detect whether a newly found solution is dominated.

In order to be sure that A is locally-optimal, we have to compare its cost with all the already known solutions B_i that may overlap with it. Thus, we have to store all the dominated and non-dominated solutions found so far. Moreover, we have to scan all the stored solutions that may overlap with A as quickly as possible. To do this, we implemented a two-dimensional linked list L_{ij}. This list stores the start positions $L_i.start$ of all the solutions. Then, each element L_i points to another list, that stores the end positions $L_{ij}.end$ and the costs $L_{ij}.cost$ of the solutions that start at position $L_i.start$. To speed up the search, we added some extra information in each L_i cell of the list: $L_i.end \leftarrow \max_j\{L_{ij}.end\}$.

The algorithm isDominated describes how to find whether A is dominated. It is used in DARN! as soon as the solution A is found. The line **13** scans all the start positions of the already stored solutions. Since the elements of L are sorted by the increasing value of $L_i.start$, and the sequence is scanned from left to right, the last L_i elements are the only ones to store solutions that may overlap with A. This is why L is scanned by the decreasing value of $L_i.start$, and, if the difference between $L_i.start$ and the beginning of A is larger than an *a priori* upper bound of the size of the largest possible ncRNA (stored in MaxRnaSize), the algorithm returns (line **14**). Then, we test whether A may overlap with a solution stored in the list L_i (line **15**). If it does, then we look for the solutions that A overlaps with (lines **16** and **17**). Finally, we test whether the already found solutions dominate A (line **18**).

Note that this data structure does not reduce the space complexity of the storage of the solutions, but it makes it possible to find whether a recently found solution is dominated in reasonable time, and whether already found solutions are dominated by a recently found solution. Moreover, this mechanism is used to prune the branch-and-bound tree. DARN! stores the cost and the positions of every dominated and non-dominated solution found so far. During

Function isDominated(A)

Input : a recently found solution A
Output: true if A is dominated, false otherwise

```
13 for i ← |L| down to 1 do
14     if (A.start − L_i.start > MaxRnaSize) then
           return false ;
15     if ([L_i.start..L_i.end] ∩ [A.start..A.end] ≠ ∅) then
16         for j ← 1 to |L_i| do
17             if ([L_i.start..L_ij.end] ∩ [A.start..A.end] ≠ ∅) then
18                 if ((L_ij.cost < A.cost) ∨ ((L_ij.cost = A.cost) ∧ (L_i.start ≤
                   A.start))) then
                       return true ;
   return false ;
```

the exploration of the branch-and-bound tree, DARN! checks whether a solution has been found in the scanned region. If this the case, then the value k is set to $c + 1$, where c is the cost of the solution already found in the region. This feature saves much time in practice.

5.3 Additional WCSP mechanisms

State-of-the-art weighted constraint solvers always implement several mechanisms that speed up the search during the exploration. Although implementing all of them is out of the scope of our work, we tried to implement the most promising ones.

An initial well-known mechanism implemented in many solvers is the fail-first dynamic ordering of the variables. Our heuristics work as follows. A high score is given to the cost functions that are unlikely to be satisfied (for instance, patterns with long words and few errors, long helices, etc.), whereas time-consuming cost functions, which are easily satisfied are given a negative score (the duplex cost function is a good example). Then, we give to each variable the sum of the score of the cost functions in which it is involved. At each node of the tree, the variable with the highest score is chosen for assignment. This favors the rapid search for *anchors* that heavily discriminate a signature, and dramatically speeds up the exploration.

A final useful mechanism is the use of prioritized queues to propagate events. For example, every cost function is divided into its crisp and soft parts, and the crisp part is always handled first. Then, we observed that `pattern` and `spacer` cost functions should be checked before `helix` cost functions, since the latter kind of cost function takes much longer to be propagated. This is why the set of cost functions is split into *simple* and *complicated* cost functions, and simple cost functions are handled first. Finally, the cause of the event is also taken into consideration: assignments are handled before lower or upper bound updates.

6 Results

The aim of the first set of experiments was to test whether our tool was competitive with respect to the ncRNA localization tools that are specific to a given ncRNA family. Since it is dedicated to a ncRNA family, this kind of tool can usually describe a signature in great detail. Here we would like to know if it is possible to make a signature that could be as detailed as the ones embedded into specific localization tools.

A typical benchmark for the ncRNA localization problem is the transfer RNA (tRNA) localization. This ncRNA has a very characteristic structure, members of this family are usually known, and there exists a very well-known program called tRNAscan-SE [7] that specifically finds the tRNAs in different genomes. This tool has been trained to localize tRNAs in many genomes, using the stochastic context-free grammar formalism. We compared DARN! with tRNAscan-SE on the genomic sequences of the yeast [4] provided by RFAM [8], which is the reference database for ncRNAs. This database has collected all the tRNAs that are located on chromosome 3 (containing 316,613 nucleotides) and on the first 165,534 nucleotides of chromosome 16 of the yeast genome.

We made a signature so that it could match as closely as possible the alignment of the tRNA of the yeast. This is a major difference with tRNAscan-SE, since the latter can localize tRNAs in many other genomes, whereas our descriptor is tuned to the yeast genome. As described in figure 6, we modeled it using sixteen variables, four `helix` cost functions, two `pattern` cost functions and six `spacer` cost functions. The descriptor is available on the web site at http://carlit.toulouse.inra.fr/Darn/index.php.

The results of the comparison, performed on a 2.4 GHz computer with 8 GB RAM running Linux, are displayed in table 1. Three main observations can be made. First, DARN! was not slower than the state-of-the-art tool. This indicates that our approach is competitive with respect to the other tools. Second, all the solutions given by DARN! were also given by tRNAscan-SE. However, since the solutions given by DARN! are exactly the ones recorded in RFAM, the two remaining solutions on chromosome 3 predicted by tRNAscan-SE are very likely to be false positive candidates (i.e. regions that were not actually tRNAs), whereas DARN! only gave the correct regions. This may show that our tool accepts quite accurate descriptions of ncRNA families. Third, the number of dominated solutions found by DARN! on chromosome 16 is almost as large as the number of non-dominated solutions. It tends to show that the mechanism is crucial, and since all the non-dominated solutions are the tRNAs that we were looking for, it also seems accurate.

Concerning H/ACA sRNAs, several studies have recently identified new members of this family in Archaea. In order to test the ability of a constraint network to model interactions between different molecules, we performed in [16] a first computational screen of Archaea *M. jannaschii*, *P. abyssi*, *P. furiosus* and *P. horikoshii* genomes for H/ACA box sRNAs. Results showed the model's value in finding and annotating orphan sRNAs as well as the significance of a weighted model in order to make it easier to analyze a potentially very large

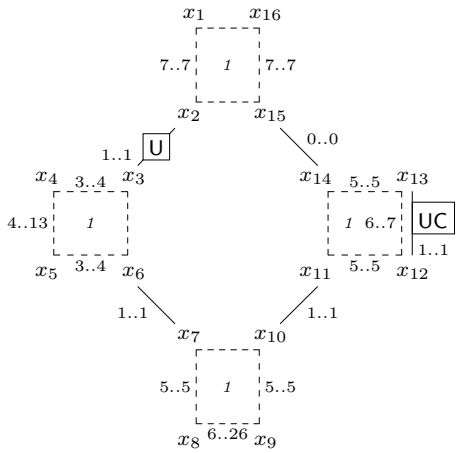


Figure 6: A tRNA signature with cost functions. Integer intervals near full lines give the allowed size of the spacer. Intervals near dotted squares give the sizes of the helices or the sizes of their loops. Italic numbers inside these squares are the number of possible errors in the helices. Only one error is allowed overall.

chromosome 3				
	time	# solutions	# dominated solutions	specificity
DARN!	0.8 s.	8	4	100%
tRNAscan-SE	1.3 s.	10	-	80%
chromosome 16, first 165,534 nucleotides				
	time	# solutions	# dominated solutions	specificity
DARN!	0.5 s.	5	4	100%
tRNAscan-SE	0.7 s.	5	-	100%

Table 1: Comparison between DARN! and tRNAscan-SE.

T. kodakarensis (> 20 million nucleotides) Time: 65s Number of solutions: 6941 Number of non-dominated solutions: 22		
H/ACA gene (#hairpins)	DARN! (cost)	BLAST
H/ACA-1 (1)	yes (0)	yes
H/ACA-2 (2)	yes (0, 1)	no
H/ACA-3 (1)	yes (0)	yes
H/ACA-4 (2)	yes (0, 1)	yes
H/ACA-5 (3)	yes (0, 0, 2)	yes
H/ACA-6 (1)	yes (0)	yes

Table 2: H/ACA candidate genes in *T. kodakarensis*. In the first column, H/ACA sRNA genes are numbered as in [16]. In brackets, #hairpins gives the number of hairpins in the gene. The second column gives results using DARN!. The costs of solutions for each hairpin of the ncRNA gene are given in brackets. The third column gives results using BlastN [2], a standard software for comparing genes on the basis of their sequence only, with default parameter values.

number of solutions. Another possible improvement was the elimination of the redundant solutions. These latter developments, implemented in DARN! gave promising results on H/ACA sRNA gene finding.

The descriptor used for these experiments is given in figure 7. The weighted framework made it possible to give a penalty to degenerated words (GA and ACA) on the one hand, and to favor a high GC content on the other hand. The maximum score was set to 2 so that DARN! gives all the known ncRNAs, with the least number of unknown solutions possible.

All the H/ACA sRNA genes similar to known Archaeal H/ACA sRNAs, representing ten hairpins, were found by using DARN! (cf. results in table 2). For a total of twenty-two solutions, ten of them represented hairpins present in at least one of the H/ACA sRNA known in other species. Nine solutions were found with a cost of 0. Among these nine solutions, two were dissimilar to known H/ACA sRNA genes and could represent new H/ACA sRNA candidates. Six solutions were found with a cost of 1. Among these six solutions, two represented hairpins present in H/ACA-2 and H/ACA-4. The other solutions of cost 1 were dissimilar to known H/ACA sRNA genes and could represent new H/ACA sRNA candidates. The remaining seven solutions had a cost of 2. One of them represented one of the three hairpins present in H/ACA-5. The other ones were not similar to known H/ACA sRNA genes and could also represent new H/ACA sRNA candidates.

```
TOP_VALUE = 3
VARIABLES X_VAR = 1...18, Y_VAR = 1...6

HARD_PART

PATTERN[word="RA",err=0](X7,X8)
PATTERN[word="RUGA",err=0](X9,X10)
PATTERN[word="ANA",err=0](X17,X18)
PATTERN[word="UN",err=0](Y3,Y4)

HELIX[stem=7..10,loop=30..65,err=1](X1,X2,X15,X16)

DUPLEX[err=1](X3,X4,Y1,Y2)
DUPLEX[err=1](X13,X14,Y5,Y6)
COMPOSITION[nucleotides="GC",threshold>=75%](X1,X2)
COMPOSITION[nucleotides="GC",threshold>=70%](X5,X6)
COMPOSITION[nucleotides="GC",threshold>=65%](X11,X12)

SPACER[lenmin=0,lenmax=8](X2,X3)
SPACER[lenmin=3,lenmax=6](X3,X4)
SPACER[lenmin=1,lenmax=1](X4,X5)
SPACER[lenmin=6,lenmax=8](X5,X6)
SPACER[lenmin=1,lenmax=1](X6,X7)
SPACER[lenmin=3,lenmax=30](X8,X9)
SPACER[lenmin=1,lenmax=1](X10,X11)
SPACER[lenmin=4,lenmax=8](X10,X12)
SPACER[lenmin=1,lenmax=1](X12,X13)
SPACER[lenmin=3,lenmax=6](X13,X14)
SPACER[lenmin=0,lenmax=8](X14,X15)
SPACER[lenmin=0,lenmax=1](X16,X17)
SPACER[lenmin=3,lenmax=6](Y1,Y2)
SPACER[lenmin=1,lenmax=1](Y2,Y3)
SPACER[lenmin=1,lenmax=1](Y4,Y5)
SPACER[lenmin=3,lenmax=6](Y5,Y6)
SPACER[lenmin=10,lenmax=12](Y1,Y6)

SOFT_PART

PATTERN[word="GA",err=1](X7,X8)
PATTERN[word="ACA",err=1](X17,X18)
COMPOSITION[nucleotides="GC",threshold>=75..100%,costs=0..2](X1,X2)
COMPOSITION[nucleotides="GC",threshold>=70..100%,costs=0..2](X5,X6)
COMPOSITION[nucleotides="GC",threshold>=65..100%,costs=0..2](X11,X12)
```

Figure 7: Descriptor of the H/ACA sRNA. `TOP_VALUE` is the value k. Cost functions and hard constraints are described separately.

7 Conclusions and future work

We have presented a new tool called DARN!, that can find new occurrences of characterized non-coding RNAs. This tool efficiently combines weighted CSP techniques with pattern-matching algorithms and data structures. It also offers a large diversity of components that can be useful for recognizing even complicated non-coding RNAs: several helices, patterns, spacers, composition in nucleotides, and, most importantly, duplexes, which model the ability of a non-coding RNA to interact with another RNA sequence.

Results on the tRNA indicated that the weighted CSP framework is able to give quite accurate signatures, even compared to specific ncRNA localization tools, and results on the H/ACA box sRNA suggest that the tool could be very useful for discovering new ncRNAs. Another useful mechanism, the selection of locally-optimal solutions, improves the tool's specificity even further.

Work is underway to design a method that would automatically infer the signature, given an alignment, and find the best costs for the cost functions. Finally, we will also implement several new cost functions that would make it possible to describe ncRNA families even more accurately: repetitions, duplex with one among several sequences, etc.

References

[1] M. Abouelhoda, S. Kurtz, and E. Ohlebusch. Replacing suffix trees with enhanced suffix arrays. *Journal of Discrete Algorithms*, 2:53–86, 2004.

[2] S. Altschul, W. Gish, W. Miller, E. Myers, and D. Lipman. Basic local alignment search tool. *J. Mol. Biol.*, 215:403–410, 1990.

[3] B. Billoud, M. Kontic, and A. Viari. Palingol: a declarative programming language to describe nucleic acids' secondary structures and to scan sequence database. *Nucleic Acids Research*, 24(8):1395–1403, 1996.

[4] J. Cherry, C. Ball, S. Weng, G. Juvik, R. Schmidt, C. Adler, B. Dunn, S. Dwight, L. Riles, R. Mortimer, and D. Botsein. Genetic and physical maps of *Saccharomyces cerevisiae*. *Nature*, 387(6632 Suppl):67–73, 1997.

[5] M. Dsouza, N. Larsen, and R. Overbeek. Searching for patterns in genomic data. *Trends in Genetics*, 13(12), 1997.

[6] S. Eddy. *Rnabob: a program to search for RNA secondary structure motifs in sequence databases*, 1996.

[7] S. Eddy and R. Durbin. RNA sequence analysis using covariance models. *Nucleic Acids Research*, 22(11):2079–88, 1994.

[8] S. Griffiths-Jones, S. Moxon, M. Marshall, A. Khanna, S. Eddy, and A. Bateman. Rfam: annotating non-coding RNAs in complete genomes. *Nucleic Acids Research*, 33:D121–D124, 2005.

[9] A. Laferrière, D. Gautheret, and R. Cedergren. An RNA pattern matching program with enhanced performance and portability. *Comp. Appl. Biosc.*, 10(2):211–212, 1994.

[10] J. Larrosa and T. Schiex. Solving Weighted CSP by Maintaining Arc-consistency. *Artificial Intelligence*, 159(1-2):1–26, 2004.

[11] O. Lhomme. Consistency techniques for numeric CSPs. In *IJCAI-93*, pages 232–238, 1993.

[12] T. Macke, D. Ecker, R. Gutell, D. Gautheret, D. Case, and R. Sampath. Rnamotif, an RNA secondary structure definition and search algorithm. *Nucleic Acids Research*, 29(22):4724–4735, 2001.

[13] G. Navarro and M. Raffinot. *Flexible Pattern Matching in Strings – Practical on-line search algorithms for texts and biological sequences*. Cambridge University Press, 2002.

[14] S. Needleman and C. Wunsch. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *Journal of Molecular Biology*, 48(3):443–453, 1970.

[15] Y. Sakakibara, M. Brown, R. Hughey, I. Mian, K. Sjölander, R. Underwood, and D. Haussler. Recent methods for RNA modeling using stochastic context-free grammars. In *CPM'94*, pages 289–306, 1994.

[16] P. Thébault, S. de Givry, T. Schiex, and C. Gaspin. Searching RNA motifs and their intermolecular contacts with constraint networks. *Bioinformatics*, 22(17):2074–2080, 2006.

[17] S. Vialette. On the computational complexity of 2-interval pattern matching problems. *Theoretical Computer Science*, 312(2-3):223–249, 2004.

[18] M. Zytnicki, C. Gaspin, and T. Schiex. A new local consistency for weighted CSP dedicated to long domains. In *SAC'06: Proceedings of the 2006 ACM symposium on Applied computing*, pages 394–398, 2006.

[19] M. Zytnicki, C. Gaspin, and T. Schiex. Suffix arrays and weighted CSPs. In A. Dal Palu, A. Dovier, and S. Will, editors, *Workshop on Constraint Based Methods for Bioinformatics*, pages 69–74, Nantes, 2006.