Approximate Matching of Structured Motifs in DNA Sequences

Nadia El-Mabrouk * Mathieu Raffinot † Jean-Eudes Duchesne ‡
Mathieu Lajoie § Nicolas Luc ¶

Abstract

Several methods have been developed for identifying more or less complex RNA structures in a genome. All these methods are based on the search for conserved primary and secondary sub-structures. In this paper, we present a simple formal representation of a helix, which is a combination of sequence and folding constraints, as a constrained regular expression. This representation allows us to develop a well-founded algorithm that searches for all approximate matches of a helix in a genome. The algorithm is based on an alignment graph constructed from several copies of a pushdown automaton, arranged one on top of another. This is a first attempt to take advantage of the possibilities of pushdown automata in the context of approximate matching. The worst time complexity is $O(krpn)$, where k is the error threshold, n the size of the genome, p the size of the secondary expression, and r its number of union symbols. We then extend the algorithm to search for pseudo-knots and secondary structures containing an arbitrary number of helices.

1 Introduction

Following the complete or partial sequencing of a large variety of genomes, one of the major challenges is to decode this huge amount of information by identifying the different genes in the new sequenced genomes. According to different empirical methods, biochemical techniques, multiple sequence alignment and dynamic programming algorithms, some common secondary structures of certain RNA families and other genetic elements have been determined. Several methods have been developed to search for such structures in a genome. Some of them are tailor-made for searching specific families, like FAStRNA [5] and tRNAscan [7, 13] for tRNAs, CITRON [12] for group I introns, SNOSCAN and others for snoRNAs [14, 19, 8]. Other methods are more general, such as RNAMOTIF [15], RNABOB [4] and PALINGOL [1]. Stochastic context-free grammars (SCFGs) [3, 22] have also been used to represent RNA structures. Whatever the method, it is always based on the identification of various conserved primary and secondary sub-structures. The primary sub-structures are usually deduced from

*Corresponding author: Département d'informatique et de recherche opérationnelle, Université de Montréal, CP 6128 Succursale Centre-ville, Montréal, Québec H3C 3J7. Tel: 1-514-343-7481. E-mail: mabrouk@iro.umontreal.ca.

†CNRS - Equipe Génome et Informatique, Evry, and ENS, 46 rue d'Ulm, Paris, France. E-mail: raffinot@genopole.cnrs.fr.

‡Département d'informatique et de recherche opérationnelle, Université de Montréal

§Département d'informatique et de recherche opérationnelle, Université de Montréal

¶CGI

a preliminary alignment of homologous sequences in different genomes. They represent a consensus for motifs approximately repeated in all the sequences of the alignment. For example, a primary representation of the $T\Psi C$ region by the consensus "YVNNRGTTCRADYCY" has been deduced from an alignment of about 500 tRNA sequences [5]. Each letter denotes a particular subset of $\{A, C, G, T\}$ (see Table 1, [2]). The meaning of the consensus is: the first position is either a C or a T in all the tRNA sequences of the alignment, the second position can be anything except a T, etc. Such consensi are network expressions (regular expressions not containing a Kleene closure). Myers and Miller [18] developed an $O(np)$ algorithm for approximately matching a sequence G of size n to a regular expression R of size p. This algorithm is based on an alignment graph obtained by concatenating $n + 1$ copies of a non-deterministic finite automaton recognizing R.

Symbol	Significance	Symbol	Significance
A	A	N	A | C | G | T
B	C | G | T	R	A | G (purine)
C	C	S	C | G
D	A | G | T	T	T
G	G	V	A | C | G
H	A | C | T	W	A | T
K	G | T	Y	C | T (pyrimidine)
M	A | C		

Table 1: Symbols used to define sets of nucleotides. The standard IUPAC code.

However, for RNA molecules, consensi are usually defined by a combination of spatial structure and sequence motif, and folding constraints can be stronger than sequence constraints. The most common type of such secondary sub-structures is a helix, that is a sequence of stacked pairs, bulges and internal loops followed by a loop (see Figures 1 and 2). In the literature, this kind of structure is sometimes referred to as a stem-loop or a palindrome.

In contrast with secondary structure prediction, few algorithms have been dedicated to helix search. One of the most adapted algorithms for searching helices with indels and mismatches is that of Sagot-Viari [21]. However, as it is designed to identify all possible folding regions in a genomic sequence, it does not allow for complex definitions of helices. Consider, for example, the $T\Psi C$ region of the tRNA, for which primary structure constraints have been described above. This region has a folded part, and it would have been more appropriate to represent it by the *secondary expression* shown in Figure 2.(a). This expression accounts for correlated constraints due to base-pairings. For example, the nucleotide numbered 2 (Figure 2) should be a T if nucleotide 1 is an A, a C if nucleotide 1 is a G, and a G if nucleotide 1 is a C. HyPaLib database project [10] aims to enumerate many of such secondary expression patterns in a specific description grammar. A secondary expression can be represented by a context free grammar. In [16], Myers gave an $O(Pn^3)$ algorithm for approximately matching a string of length n and a context-free language specified by a grammar of size P.

Despite all the existing methods described above, a rigorous formalisation of a helix is missing. In this paper, our aim is to provide a simple formal representation of a helix allowing for a flexible search with errors in a nucleic acid sequence. We present a specific algorithm to search for a secondary expression S that is extensible to pseudo-knots and general secondary structures. This algorithm reports the final positions of all the approximate occurrences of S up to k errors. The allowed errors are insertion, deletion and substitution of single

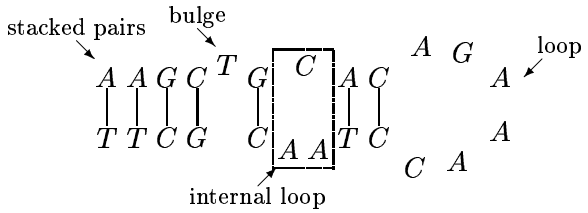
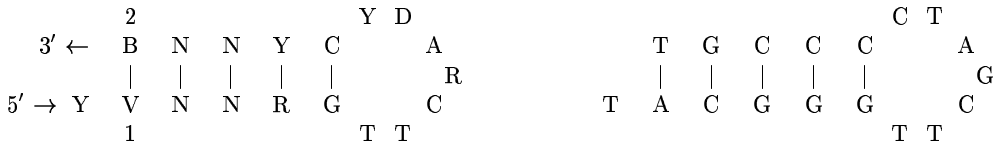


Figure 1: An example of a helix. A, C, G, T are the four nucleotides (or bases) and | is a Watson-Crick (A-T or C-G) base-pairing. The helix contains a loop of 6 nucleotides, a stem of 7 stacked pairs, a bulge (unpaired nucleotides at one side of the stem) and an internal loop (unpaired nucleotides at both sides of the stem).



(a) $T\Psi C$ (b) An occurrence

Figure 2: (a): A secondary expression representing a consensus for the $T\Psi C$ region of tRNAs; (b): An occurrence of the secondary expression (a).

nucleotides (edit distance). We have extended the method of Myers and Miller [18, 17]. The algorithm is based on an alignment graph constructed from several copies of a pushdown automaton, arranged one on top of another. As in [18], the approximate search problem is reduced to finding least error paths in this graph. However, the problem we face is more complex. Indeed, as secondary expressions are constrained network expressions, we need to restrict the least error computation to a specific set of paths, that can not be defined *a priori*. Therefore, instead of using a non-deterministic finite automaton as in [18], we use a pushdown automaton. To our knowledge, this paper is a first attempt to take advantage of the possibilities of these pushdown automata in the context of approximate matching. An array of $k+1$ sets of stacks is associated to each node of the alignment graph to control the different paths leading to an acceptance. The worst time complexity of our new algorithm is $O(krpn)$, where k is the error threshold, n is the genome size, p the size of the secondary expression S, and r the number of | (union symbol) in S.

The approach easily generalizes to pseudo-knots and more complex structures containing an arbitrary number of helices. This allows searching for large and complete secondary structures. We define such structures and explain these extensions.

This paper is organized as follows. We formally define a secondary expression in Section 2. In Section 3, we describe our pushdown automaton, and in Section 4, we introduce the notion of alignment graph. In Section 5, we present our approximate matching algorithm, and we analyze its complexity. We also specify how to actually report the occurrences. We then extend our approach to pseudo-knots and generalized secondary expressions in Section 6. Finally, in Section 7, we report experimental results for the search for tRNAs and RNase P RNA sub-structures in two genomes.

A preliminary extended abstract of this work has been published in [6]. The present version contains important corrections, modifications and extensions.

2 Problem and definitions

A string is a sequence of nucleotides, i.e. of characters from $N = \{\texttt{A,C,G,T}\}$. A **genome** is a long string on this alphabet. Given a secondary expression S and a genome G, the problem is to find all the approximate occurrences of S in G. This formulation hides different concepts that have to be defined formally: what do we mean by a secondary expression? What do we mean by an approximate match? To define our concept of a secondary expression, we need the following preliminary definitions.

Definition 1 (network expression) *The set* **NetSet** *of network expressions over the alphabet N is defined recursively by:* **(1)** *The empty pattern ε is a network expression;* **(2)** *Any character of $N \cup \{\epsilon\}$ is a network expression (called an* **atomic expression***);* **(3)** *If E_1 and E_2 are two network expressions, then $E_1|E_2$ and E_1E_2 are network expressions.*

For example, `RM` (Table 1) represents the network expression `(A|G)(A|C)`. A network expression E is formally a specification of a set of sequences, the language $\mathcal{L}(E)$. For instance, the sequences matched by $E = \texttt{RM}$ are those of the set $\mathcal{L}(E) = \{\texttt{AA,AC,GA,GC}\}$.

Definition 2 (complement) *The* **complement** *of the network expression E over the alphabet N, denoted $\overline{E}$, is the network expression defined recursively by:* **(1)** $\overline{\varepsilon} = \varepsilon$; **(2)** $\overline{A} = T$, $\overline{T} = A$, $\overline{C} = G$ *and* $\overline{G} = C$; **(3)** *If $E = E_1E_2$, then $\overline{E} = \overline{E_2}\ \overline{E_1}$;* **(4)** *If $E = E_1|E_2$, then $\overline{E} = \overline{E_1}|\overline{E_2}$.*

For example, if $E = \texttt{RM}$, then $\overline{E} = \texttt{KY}$. This definition considers only the more stable base pairs, that is the Watson-Crick base pairs. To allow for wobble pairs $G-T$ as well ($G-U$ in RNAs), it suffices to modify the rule **(2)** by **(2')**: $\overline{A} = T$, $\overline{T} = (A|G)$, $\overline{C} = G$ and $\overline{G} = (C|T)$.

A **secondary expression** is a formal description of a helix containing paired and unpaired regions. Unpaired regions can be modeled by network expressions, and paired regions by pairs formed by a network expression and its complement. To distinguish between the different parts of a secondary expression, we need to specify the type of each network expression by an element of $\{p, sl, sr\}$. A network expression marked by a p represents an unpaired region, and an expression marked by an sl (resp. sr) represents a left strand (resp. a right strand) of a paired region.

Definition 3 (secondary expression) *A secondary expression is a sequence of elements of $NetSet \times \{p, sl, sr\}$. The set of secondary expressions is defined recursively by:*

- *If E is a network expression (possibly ε), then $S = (E, p)$ is a secondary expression;*
- *If E_1, E_2, E_3 are network expressions (possibly ε) and S' is a secondary expression, then the sequence $S = (E_1, p)(E_2, sl)\ S'\ (\overline{E_2}, sr)(E_3, p)$ is a secondary expression.*

For simplicity, successive network expressions specifying unpaired regions are reduced to only one unpaired region. Similarly, paired regions specified by successive $E_i, E_{i+1} \cdots E_j$ on the left strand and successive $\overline{E_j} \cdots \overline{E_{i+1}}\,\overline{E_i}$ on the right strand are reduced to only one paired region.

The structure in Figure 3 is an example of a secondary expression. The loop, bulges and internal loops are unpaired regions. Our definition of a secondary expression allows for an empty loop. Notice also that, from the definition of a secondary expression, there is no

network expression marked sl after a network expression marked sr in the sequence defining a secondary expression S. In other words, a secondary expression represents a single helix. We generalize our definitions and algorithms to secondary structures containing more than one helix in Section 6.

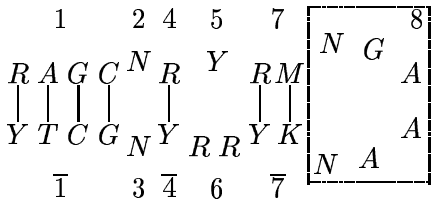


Figure 3: A rough representation of group II intron's domain V. The formal representation of this secondary expression is: $S = (1, sl)(2, p)(4, sl)(5, p)(7, sl)(8, p)(\overline{7}, sr)(6, p)(\overline{4}, sr)(3, p)(\overline{1}, sr)$, where each number represents the network expression under (or over) it. For example, the number 1 represents RAGC.

Let $S = (E_i, t_i)_{1 \leq i \leq n}$ be a secondary expression such that for each i, $E_i \in$ NetSet and $t_i = p$, sl or sr. The network expression $E_1 E_2 \cdots E_n$ obtained by concatenating all E_is is **the network expression of** S denoted $NetExp(S)$. Notice that S does not have the same conceptual meaning as $NetExp(S)$. The formal definition of a language described by a secondary expression is given below.

Definition 4 *The language $\mathcal{L}(S)$ specified by a secondary expression S is defined recursively by:*

- *If $S = (E, p)$, then $\mathcal{L}(S) = \mathcal{L}(E)$;*
- *If $S = (E_1, p)(E_2, sl)\, S'\, (\overline{E_2}, sr)(E_3, p)$, such that E_1, E_2, E_3 are network expressions and S' is a secondary expression, then $\mathcal{L}(S) = \{u \in N^* \ / \ u = vwx\overline{w}z$ for $v \in \mathcal{L}(E_1)$, $w \in \mathcal{L}(E_2)$, $z \in \mathcal{L}(E_3)$ and $x \in \mathcal{L}(S')\}$.*

For example, in Figure 2, the helix (b) is an occurrence of the secondary expression (a).

Given a genome G and a secondary expression S, the "approximate matching problem" is to find all positions in G corresponding to an approximate occurrence of a pattern in $\mathcal{L}(S)$. The notion of "approximate occurrence" is tightly related to a distance δ, and to a threshold k. Let $A = a_1 \cdots a_p$ and $B = b_1 \cdots b_q$ be two sequences over N. For $a, b \in N$, $\delta(a, b)$ gives the value of aligning a with b, $\delta(\varepsilon, b)$ that of leaving b unaligned in B, and $\delta(a, \varepsilon)$ that of leaving a unaligned in A. The value of an alignment of A with B is the sum of the values assigned by δ to its aligned pairs and unaligned symbols. We denote by $\delta(A, B)$ the score of a minimal-cost alignment of A with B.

In this paper, we consider the edit distance, that is:

$$\begin{cases} \delta(a, b) = 1 & \text{if } a, b \in N,\ a \neq b \quad \text{(substitution)} \\ \delta(a, a) = 0 & a \in N \quad \text{(match)} \\ \delta(a, \varepsilon) = \delta(\varepsilon, a) = 1 & a \in N \quad \text{(insertion, deletion)} \end{cases}$$

Formally, the set of sequences approximately matching a secondary expression S within k is $\mathcal{L}_\delta(S, k) = \{A \ / \ \exists\, B \in \mathcal{L}(S),\ \delta(A, B) \leq k\}$. The problem of approximately matching S to G within a threshold k is to find, in G, all occurrences of sequences of $\mathcal{L}_\delta(S, k)$.

Remark : Although scores are assigned to pairs of symbols of the alphabet (not to helix pairings or bonds), they allow to account, not only for primary structure divergence by for secondary structure mispairings as well. Consider, for example, the first pairing $(R-Y)$ of the structure of Figure 3. Here the user requires to have a purine on the left strand, a pyrimidine on the right strand, and the two nucleotides should be correctly paired ($A-T$ or $G-C$). If the purine/pyrimidine restriction is respected but with an uncorrect pairing, this accounts for a secondary structure error (score of 1). Conversaly, if the purine/pyrimidine restriction is not respected, this accounts for a primary structure error (score of 1 or 2). If the primary structure constraint is not important (allow for a pyrimidine$-$purine pairing as well), then this pairing should be defined as an $N-N$ pairing.

Though the notion of an approximate occurrence given in this paper may seems uncomplete, the goal is to use a score that provides a simple and intuitive way to define a helix and search it in a database.

3 A pushdown automaton recognizing a secondary expression

The language generated by a secondary expression S is a regular language recognized by a finite automaton. However, the size of this automaton is exponential in r (number of $|$ in S). Using a pushdown automaton allows a condensed representation of the paths in the automaton and a more efficient approximate matching algorithm.

We use a particular non-deterministic, state-labeled, representation of a **finite pushdown automaton**. Such an automaton will be referred to as an ε-NFPA. Formally, an ε-NFA *PushA* consists of:

- an input alphabet N;
- a stack alphabet Γ;
- a set V of vertices called states;
- a set E of directed edges between states;
- a function λ assigning a label $\lambda(s) \in N \cup \{\varepsilon\}$ to each state s;
- a function γ from $V \times (N \cup \{\varepsilon\}) \times \Gamma$ to a finite subset of $V \times \Gamma^*$;
- an initial state $\theta \in V$;
- a final state $\phi \in V$;
- a particular stack symbol $I \in \Gamma$ called the start symbol.

The interpretation of $\gamma(t, a, Z) = (s, \alpha)$ where t and s are states, $a \in N \cup \{\varepsilon\}$, Z is a stack symbol and $\alpha \in \Gamma^*$, is that the automaton in state t, with input symbol a and Z the top symbol on the stack, can enter state s and replace symbol Z by α. We adopt the convention that the leftmost symbol of α will be placed highest on the stack.

The automaton *PushA* is a vertex-labeled directed graph with a stack associated to each state. The notation $t \rightarrow s$ means that there is an edge in *PushA* from state t to state s. For a state t, $w \in N^*$ and $\alpha \in \Gamma^*$, we say $(t, aw, Z\alpha) \vdash\!\!\rightarrow (s, w, \beta\alpha)$ if $\gamma(t, a, Z) = (s, \beta)$. For simplicity, if the stack content is not required, we just denote a transition by $(t, aw) \vdash\!\!\rightarrow (s, w)$.

We use $\stackrel{*}{\longmapsto}$ for the transitive closure of $\longmapsto$. The **language accepted by *PushA*** is defined as:

$$\mathcal{L}(PushA) = \{w \ / \ (\theta, w, I) \stackrel{*}{\longmapsto} (\phi, \varepsilon, \alpha)\}.$$

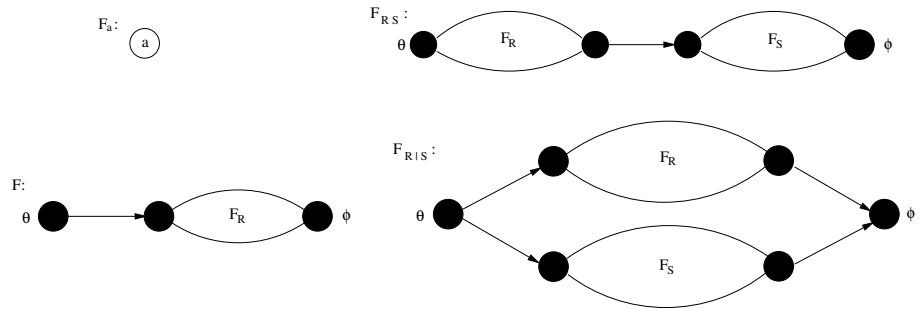


Figure 4: Constructing the ε-NFA recognizing a network expression. Black states are ε-states.

The construction of the ε-NFPA recognizing a secondary expression S is based on the construction of the ε-NFA (non-deterministic finite automaton) denoted $\mathcal{A}(S)$, recognizing the network expression $NetExp(S)$. Figure 4 depicts the inductive construction (originally from Thompson [25]) of an ε-NFA recognizing a network expression (atomic, of form RS or $R|S$). Figure 5 gives an example of an automaton $\mathcal{A}(S)$.

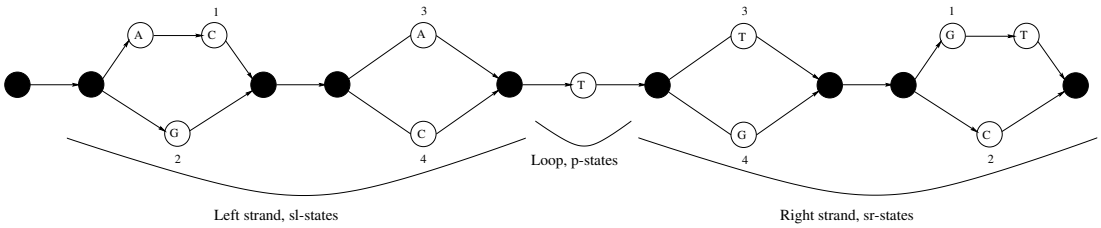


Figure 5: The ε-NFA recognizing $NetExp(S)$, for $S = (E_1, sl)(E_2, p)(\overline{E_1}, sr)$, with $E_1 = ((AC)|G)(A|C)$ and $E_2 = T$. Numbers corresponding to marked states are shown. States 1 and 3 are up-states, whereas states 2 and 4 are down-states.

The language specified by S is a subset of that specified by $NetExp(S)$. Indeed, the right strand of a secondary structure depends on its left strand. Therefore, the pushdown automaton recognizing S should keep track of the paths traversed in the left strand. To do so, we use an appropriate numbering of the states of S, that we define after some preliminary notations.

Let s be a state of $\mathcal{A}(S)$. The state s corresponds to an atomic expression of a network expression E of $NetExp(S)$. If E is marked sl, then s is an **sl-state**; if E is marked p, then s is a **p-state**; if E is marked sr, then s is an **sr-state**. If s is an sl-state, we denote by $\overline{s}$ the state corresponding to the complement atomic expression in $\overline{E}$. By extension, the complementary of a subpath $\xi = s_1 s_2 \dots s_d$ is $\overline{\xi} = \overline{s_d} \dots \overline{s_2 s_1}$.

Let r be the number of symbols $|$ in all the network expressions of S labeled sl. Let $(|_1, \cdots, |_r)$ be the sequence obtained by annotating them from left to right. As each $|_i$ is a binary operator, it is applied to two expressions: an "up" expression $E_{i,1}$, and a "down" expression $E_{i,2}$ (by reference to Thompson representation, see Figure 4). Any state of $E_{i,1}$

or $\overline{E_{i,1}}$, for any i, is called an **up-state**, and any state of $E_{i,2}$ or $\overline{E_{i,2}}$ is called a **down-state**. For each i, $1 \leq i \leq r$, let $s_{i,1}$ be the state corresponding to the last atomic expression of $E_{i,1}$, $s_{i,2}$ the one corresponding to the last atomic expression of $E_{i,2}$. For each $s_{i,j}$, let $\nu(s_{i,j}) = \nu(\overline{s_{i,j}}) = 2(i-1)+j$. A state is said **marked** if it belongs to $\{s_{1,1}, s_{1,2}, \cdots s_{r,1}, s_{r,2}\} \cup \{\overline{s_{1,1}}, \overline{s_{1,2}}, \cdots \overline{s_{r,1}}, \overline{s_{r,2}}\}$, and **unmarked** otherwise (see Figure 5). The stack alphabet is the set of marks $\Gamma = \{0, 1, \cdots 2r-1\}$, where $I = 0$ is the stack start symbol.

We have now all the notations and concepts to define the ε-NFPA *PushA*(S) that recognizes $\mathcal{L}(S)$. It is obtained from $\mathcal{A}(S)$ by defining the γ transition function as follows.

Definition 5 *Let Z be the top symbol on the stack, a be any character of $N \cup \{\varepsilon\}$, s any state, and $t \to s$ any edge leading to s. The transition $\gamma(t, a, Z)$ is defined if and only if $a = \lambda(s)$. In that case:*

Tr_1 *If s is a p-state or an unmarked state, then $\gamma(t, \lambda(s), Z) = (s, Z)$.*

Tr_2 *If s is a marked sl-state, then $\gamma(t, \lambda(s), Z) = (s, \nu(s)Z)$.*

Tr_3 *If s is an sr-state such that $\nu(s) = Z$, then $\gamma(t, \lambda(s), Z) = (s, \varepsilon)$.*

Roughly, if we enter a marked *sl*-state s, then we push $\nu(s)$ on the stack (Tr_2); if we enter a marked *sr*-state s, then we remove the top symbol of the stack (Tr_3); if we enter any other state, we do not change the stack (Tr_1). Notice that, as soon as the stack begins to be emptied, it can not grow again. This is due to the fact that an *sr*-state can not be followed by an *sl*-state.

To prove that our automaton *PushA*(S) recognizes $\mathcal{L}(S)$, we need to introduce a specific subpath decomposition. The decomposition of a marked *sl*-part is built by subpath extension using the following rules:

(a) at the beginning, each marked state s represents a distinct subpath $\xi_s = [s_{left} = s, s_{right} = s]$;

(b) each subpath is extended to the left by adding one after the other the state s' having at most a single incoming edge and an outgoing edge leading to s_{left}. The automaton construction insures that s', if it exists, is unique. Then, s_{left} becomes s' and so on;

(c) each subpath is extended to the right by concatenating one after the other the state s'' having at most a single outgoing edge and an incoming edge from s_{right}. The rightest state s_{right} then becomes s'' and so on.

The decomposition of an unmarked *sl*-part is the unique path from the initial state to the final one. Figure 6 is an illustration of this decomposition. Notice that subpaths can overlap.

The decomposition of an *sl*-part of a secondary expression S induces a symmetric decomposition for the corresponding *sr*-part. Figure 7 shows the symmetric decomposition of that in Figure 6.

Lemma 1 *Let ξ be a path from θ to ϕ in the ε-NFPA* PushA(S). *Let $(\xi_1, \ldots, \xi_r)$ be the sequence of sl-subpaths of ξ. Then $(\overline{\xi_r}, \ldots, \overline{\xi_1})$ is the sequence of sr-subpaths of ξ.*

Proof. The path ξ goes through the automaton in two steps: 1. ξ goes through *sl*-states (marked or unmarked) or *p*-states (corresponding to the automaton built on the left part of

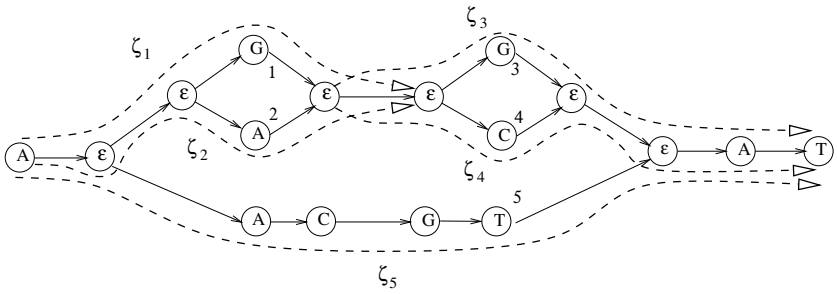


Figure 6: Subpaths decomposition of the *sl*-part of $PushA(S)$, $S = A((G|A)(G|C)|ACGT)AT$.

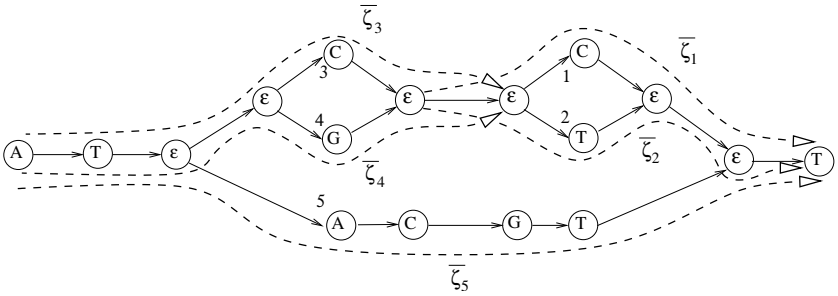


Figure 7: Symmetric subpaths decomposition of the *sr*-part of $PushA(S)$, for S of Figure 6.

$S)$; 2. ξ goes through *sr*-states (marked or not) or *p*-states (corresponding to the automaton built on the right part of S).

At each subpath ξ_i of ξ, the mark z_i of the last state of ξ_i is pushed on the stack Z by the transition rule Tr_2, leading to the stack $Z = z_1, z_2, \ldots, z_r$ at the end of step 1. Notice that these marks are the only ones pushed in the stack. The only way to reach the final state ϕ with an empty stack is to pop out each z_l using the transition rule Tr_3, which means entering into the *sr*-subpath numbered z_l, the complementary of the *sl*-subpath numbered z_l.

By induction, emptying Z means that we went through the complementaries of all the subpaths ξ_i that led to the stack elements, in reverse order □

Lemma 2 *The ε-NFPA* PushA(S) *recognizes the language generated by the secondary expression S.*

Proof. '⇒'. Let u be a word recognized by $PushA(S)$ and ξ_u be the corresponding path. As $PushA(S)$ is based on $\mathcal{A}(\mathcal{S})$, the Thompson automaton recognizing $NetExp(S)$, $u \in \mathcal{L}(NetExp(S))$. To prove that u is in the language $\mathcal{L}(S)$, it only remains to verify that the secondary constraints are satisfied. Let us prove the lemma by induction. $S = (E_1, p)(E_2, sl)\, S'\, (\overline{E_2}, sr)(E_3, p)$. The path ξ_u is of form $\xi_u = \xi_{1,p}\xi_{2,sl}\xi_{S'}\xi_{2,sr}\xi_{3,p}$. But from Lemma 1, $\xi_{2,sr} = \overline{\xi_{2,sl}}$. $\xi_u = \xi_{1,p}\xi_{2,sl}\xi_{S'}\overline{\xi_{2,sl}}\xi_{3,p}$. By induction, the secondary constraints are satisfied along all the path ξ_u, proving that $u \in \mathcal{L}(S)$.
'⇐'. Let u be a word in $\mathcal{L}(S)$. As $\mathcal{L}(S) \subset \mathcal{L}(NetExp(S))$, u labels a path in the Thompson automaton built on $NetExp(S)$, and thus u is recognized by $PushA(S)$ □

4 Alignment graph

For the problem of aligning a network expression E to a sequence G of size n within a threshold k, Myers and Miller showed in [18] that it is easier to reduce the problem to one of finding a shortest source-to-sink path in a weighted and directed **alignment graph** depending on E and G. The graph is constructed from $n+1$ copies of the ε-NFA recognizing E, arranged one on top of another. We use a similar representation for secondary expressions. Let S be a secondary expression and $NetExp(S)$ be its network expression. The alignment graph $AlGraph(S, G)$ is constructed from $n+1$ copies of the ε-NFA $PushA(S)$ recognizing $NetExp(S)$ (Figure 8). Formally, the vertices of the graph are all the pairs (i, s) for $0 \leq i \leq n$ and $s \in V$. For every (i, s), there are up to 5 edges directed into it:

- If $i > 0$, then there is a **deletion edge** from $(i-1, s)$ leading to 1 error;
- If $s \neq \theta$, then for each state t such that $t \to s$, there is an **insertion edge** from (i, t) leading to 1 error;
- If $i > 0$ and $s \neq \theta$, then for each state t such that $t \to s$, there is a **substitution edge** from $(i-1, t)$ leading to 0 or 1 error, depending on the matched characters.

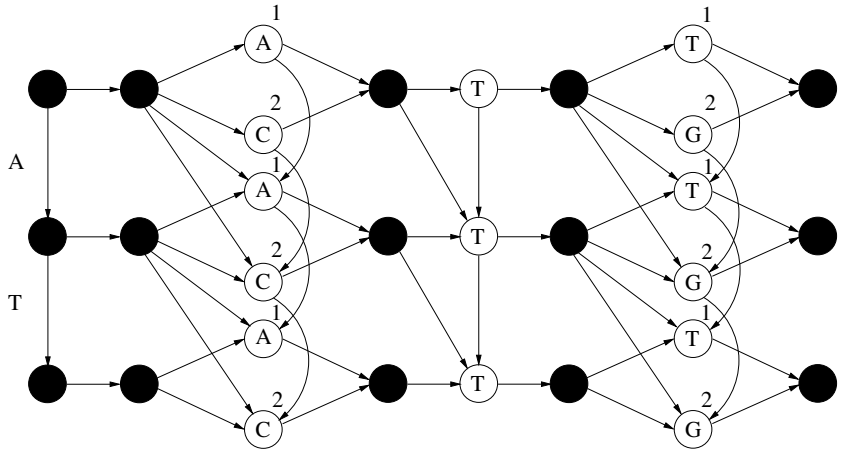


Figure 8: The alignment graph for $G = \text{AT}$ versus $NetExp(S) = (A|C)T(T|G)$.

Myers and Miller [18] show that the problem of aligning G to $NetExp(S)$ is equivalent to finding a least cost path between source vertex $(0, \theta)$ and sink vertex (n, ϕ). The cost of a path is the sum of the costs of its edges. Moreover, all substitution and deletion edges entering ε-labeled vertices except θ can be removed without destroying the property of there being a path corresponding to every alignment. With this simplification, for every vertex (i, s) of the graph, the maximal number of edges directed into it is reduced to 3: at most two insertion edges and no others for ε-states, and one of each kind of edge for any other state.

For any state s of $PushA(S)$, let $\mathcal{L}_{PA}(s)$ be the set of words recognized by $PushA(S)$ from θ to s. For any path from θ to s, we denote by $\Pi(s)$ the sequence obtained by concatenating the labels λ of the states on this path. Let $\Pi(i, s) = (i_1 = 0, s_1 = \theta)(i_2, s_2) \cdots (i_p = i, s_p = s)$ be any path from $(0, \theta)$ to (i, s), and $\Pi(s)$ be the corresponding path in $PushA(S)$ (each maximal sequence of identical states $s_l = s_{l+1} = \cdots = s_m$ is replaced by the unique state s_l). Then $\Pi(i, s)$ spells an alignment between the prefix $G[1, i]$ of size i of G and $\Pi(s)$. A path

modeling such an alignment is a **valid path** iff $\Pi(s)$ is a word of $\mathcal{L}_{PA}(s)$. Now, every **valid alignment**, that is an alignment between $G[1,i]$ and a pattern of $\mathcal{L}_{PA}(s)$, is spelled by at least one valid path. Thus, the problem of approximately matching a prefix of size i of G to a prefix $\Pi(s)$ of a word of $\mathcal{L}(S)$ is equivalent to finding a least cost valid path between source vertex $(0,\theta)$ and (i,s).

In order to select valid paths, we associate an array of $k+1$ sets of stacks to each node (i,s) of $AlGraph(S,G)$. More precisely, for any state (i,s) we consider, for each e, $0 \leq e \leq k$, the set of stacks:

$$\begin{aligned} Stack\,(i,s,e) \quad = \quad & \{\alpha \in \Gamma^* \mid \exists \text{ a path } \Pi(i,s) \text{ of cost } e \\ & \text{such that } (\theta, \Pi(s), I) \stackrel{*}{\longmapsto} (s, \varepsilon, \alpha)\} \end{aligned}$$

Note that $Stack\,(i,s,e)$ can be empty, which is different from a set of stacks containing only the empty stack ε.

To compute the set of stacks of error level e at each node (i,s), we have to consider all possible transitions (among three transitions) that are likely to give rise to e errors at node (i,s). We define the set of valid tails to take into account this consideration.

Definition 6 *The set of* **valid tails** $VT(i,s,e)$*, for a node* (i,s) *and a cost* e*, is a set of triplets defined as follows:*

- *s is an sl-state or a p-state;*
  - *if* $(j,t) \rightarrow (i,s)$ *is an insertion or deletion edge and* Stack $(j,t,e-1) \neq \emptyset$*, then* $(j,t,e-1)$ *is a valid tail;*
  - *if* $(j,t) \rightarrow (i,s)$ *is a substitution edge then: if* $a_i = \lambda(s)$ *and* Stack $(j,t,e) \neq \emptyset$*,* (j,t,e) *is a valid tail; else if* $a_i \neq \lambda(s)$ *and* Stack $(j,t,e-1) \neq \emptyset$*,* $(j,t,e-1)$ *is a valid tail;*
- *s is an sr-state;*
  - *if* $(j,t) \rightarrow (i,s)$ *is a deletion edge and* Stack $(j,t,e-1) \neq \emptyset$*, then* $(j,t,e-1)$ *is a valid tail;*
  - *if* $(j,t) \rightarrow (i,s)$ *is an insertion edge and* Stack $(j,t,e-1) \neq \emptyset$ *then* $(j,t,e-1)$ *is a valid tail if s is unmarked, or s is marked and there is a stack in* Stack $(j,t,e-1)$ *with top symbol* $\lambda(s)$*.*
  - *if* $(j,t) \rightarrow (i,s)$ *is a substitution edge then:*
    - *if* $a_i = \lambda(s)$ *and* Stack $(j,t,e) \neq \emptyset$*, then* (j,t,e) *is a valid tail if s is unmarked, or s is marked and there is a stack in* Stack (j,t,e) *with top symbol* $\lambda(s)$*.*
    - *if* $a_i \neq \lambda(s)$ *and* Stack $(j,t,e-1) \neq \emptyset$*, then* $(j,t,e-1)$ *is a valid tail if s is unmarked, or s is marked and there is a stack in* Stack $(j,t,e-1)$ *with top symbol* $\lambda(s)$*.*

We denote by $VT^D(i,s,e), VT^I(i,s,e)$ and $VT^S(i,s,e)$ the subsets of $VT(i,s,e)$ built, respectively, from the deletion, insertion and substitution edges. Note that each of these subsets contain at most one "Stack" each.

4.1 Representing the stacks

Before describing how to update the sets of stacks, we give a condensed and useful representation of a set of stacks as a binary tree.

An empty set of stacks is represented by $\emptyset$. Otherwise it is represented by a binary tree P, that is a linked structure in which each node has one integer value, contained in the field $P.val$, and two link fields $P.left$ and $P.right$, representing its left and right children, respectively, or are null pointers, denoted $NULL$. A leaf node is characterized by having null values for both $left$ and $right$. Note that an empty tree $NULL$ represents the set of stacks $\{\varepsilon\}$.

We define the following operations on these trees.

- **INSERT**(P, num) creates a new node P' such that: (1) $P'.val = num$; (2) $P'.left = P$ and $P'.right = NULL$. The procedure sets $P = P'$.
- **REMOVE**(P) removes the top element of P. It can be used only when $P.right = NULL$. Set $P = P.left$.
- **COMBINE**(P_1, P_2) creates a new note P such that $P.val = 0$, $P.left = P_1$ and $P.right = P_2$. This function returns the address of the node P.
- **MERGE**(P_1, P_2) creates a new binary tree P by recursively merging pairs of left nodes and pairs of right nodes. At each step the trees P_1, P_2 being merged should verify the property: one of the two trees is empty, or $P_1.val = P_2.val$. In the first case, P is the non-empty tree (P_1 or P_2), and in the second case, $P.val = P_1.val$, and left and right nodes of P are created by MERGE$(P_1.left, P_2.left)$ and MERGE$(P_1.right, P_2.right)$ (see Figure 9).

To simplify the description of the algorithm, we extend these operations to take as input an empty set. In that case, all that is done is to return an empty set.

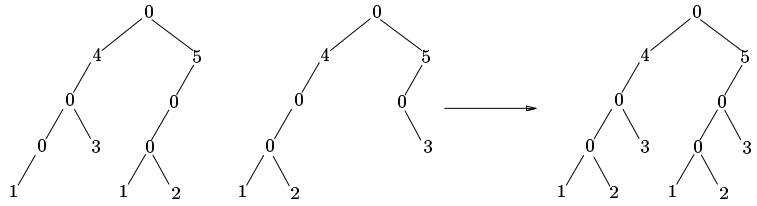


Figure 9: Merging two trees by recursively merging pairs of left nodes and pairs of right nodes.

The trees are initialized with the node $NULL$. For the source vertex $(0, \theta)$ and $0 \leq e \leq k$, $P(0, \theta, e) = NULL$.

4.2 Updating the stacks

We denote by $P(i, s, e)$ the binary tree representing $Stack\,(i, s, e)$. Roughly, $P(i, s, e)$ is obtained by merging the binary trees associated to the triplets of $VT(i, s, e)$. To simply the notation and the description of the algorithm, these binary trees will simply be denoted by $VT(i, s, e)$ (and similarly for the binary trees associated to $VT^D(i, s, e), VT^I(i, s, e)$ and $VT^S(i, s, e)$).

For any ε-state (i,s), as no deletion or substitution edge leads to (i,s), for each e, $0 \leq e \leq k$, $VT(i,s,e)$ contains at most two triplets, and the tree $P(i,s,e)$ is computed by the following procedure:

```
Procedure Update-ε-Stack(i,s)
1.  For  e = 0 ... k Do
2.      If  VT(i,s,e) = {(i,t,e)} Then
3.          P(i,s,e) = P(i,t,e)
4.      Else /* VT(i,s,e) = {(i,t1,e),(i,t2,e)}, with
                (i,t1) up-state, and (i,t2) down-state */
5.          P(i,s,e) =COMBINE(P(i,t1,e),P(i,t2,e));
6.      End of if
7.  End of for
```

Consider now a non ε-state (i,s). The triplets of $VT(i,s,e)$ give rise to at most one deletion edge $(i-1,s) \rightarrow (i,s)$, one insertion edge $(i,t) \rightarrow (i,s)$, and one substitution edge $(i-1,t) \rightarrow (i,s)$. Therefore, $P(i,s,e)$ is obtained by combining at most three binary trees $P(i,s,e)^D$, $P(i,s,e)^I$ and $P(i,s,e)^S$, corresponding respectively to the potential insertion, deletion, and substitution edge (possibly coming from other error levels). Each of these trees is considered if and only if the tail of the corresponding egde belongs to $VT(i,s,e)$. Procedures *Update-p-u-Stack*, *Update-sl-Stack* and *Update-sr-Stack* shown in Figure 10 describe the construction of these trees, and the construction of the resulting $P(i,s,e)$ trees for $0 \leq e \leq k$.

Lemma 3 *Let $P(i,s,e)$, for $0 \leq e \leq k$, be the binary tree obtained by procedures* Update-$\{\varepsilon, p-u, sl, sr\}$-Stack *at node (i,s) for the error level e. Then $\alpha \in P(i,s,e)$ if and only if there exists an e-error path $\Pi(i,s)$ such that $(\theta, \Pi(s), I) \stackrel{*}{\longmapsto} (s, \varepsilon, \alpha)$.*

Proof. "$\Leftarrow$" Let $\Pi(i,s) : (\theta, \Pi(s), I) \stackrel{*}{\longmapsto} (s, \varepsilon, \alpha)$ be an e-error path from $(0,\theta)$ to (i,s). This path models an alignment between the prefix of size i of G and $\Pi(s)$, a word of $\mathcal{L}_{PA}(s)$. We proceed by recursion on the size of $\Pi(i,s)$.

Consider a path of size 1, that is a path containing an unique edge $E = (0,\theta) \longrightarrow (i,s)$, where $i = 0$ or 1, and s is a state of $PushA(S)$ reachable from θ in one step. In this case, $e \in \{0,1\}$. The state s should be either a p-state or a marked or an unmarked sl-state.

If s is a p-state or an unmarked state, from Tr_1 (Definition 5), $\alpha = I$. The e-level tree of a p-state is computed by *Update-p-u-Stack*, or by *Update-ε-Stack* if s is an ε-state. As $P(0,\theta,e)$ is initialized with the "empty tree" I, these two procedures combine or merge at most three empty trees, and thus give rise to an empty stack for state (i,s,e).

If s is an sl-state, the edge E can not be a deletion, and thus $\alpha = \nu(s)$. The e-level set of stacks of a marked sl-state is computed by *Update-sl-Stack*. Lines 6 and 7 of *Update-sl-Stack* insert $\nu(s)$ on top of the empty trees $VT^I(i,s,e), VT^S(i,s,e)$. These trees are then merged in Lines 8, leading to a binary tree restricted to $\nu(s)$.

Consider now an e-error path $\Pi(i,s) = \Pi(j,t)E$, where $\Pi(j,t) = (0,\theta) \stackrel{*}{\longmapsto} (j,t)$ and $E = (j,t) \stackrel{v}{\rightarrow} (i,s)$. $Pi(j,t)$ should be an e' error-path, with $e' = e$ or $e-1$. Let $(\theta, \Pi(t), I) \stackrel{*}{\longmapsto} (t, \varepsilon, \alpha_t)$, and $(t, \lambda(s), \alpha_t) \longmapsto (s, \varepsilon, \alpha_s)$. From the recurrence hypothesis, α_t is in $P(j,t,e')$.

1. If s is a p-state or an unmarked state, then from Tr_1 (Definition 5), $\alpha_s = \alpha_t$. The e-level tree of a p-state is calculated either by *Update-p-u-Stack* or by *Update-ε-Stack*. These

Procedure Update-p-u-Stack(i,s) /* If s is a p-state or an unmarked state */

1. **For** $e = 0 \ldots k$ **Do**
2. $P(i,s,e)$ =MERGE($VT(i,s,e)$)
3. **End of for**

Procedure Update-sl-Stack(i,s) /* If s is an sl-state */

4. **For** $e = 0 \ldots k$ **Do**
5. $P(i,s,e)^D = VT^D(i,s,e)$
6. $P(i,s,e)^I$ =INSERT($VT^I(i,s,e), \nu(s)$);
7. $P(i,s,e)^S$ =INSERT($VT^S(i,s,e), \nu(s)$);
8. $P(i,s,e)$ =MERGE($P(i,s,e)^D, P(i,s,e)^I, P(i,s,e)^S$)
9. **End of for**

Procedure Update-sr-Stack(i,s) /* If s is an sr-state */

10. **For** $e = 0 \ldots k$ **Do**
11. $P(i,s,e)^D = VT^D(i,s,e)$
12. $P(i,s,e)^I = \emptyset$; $P(i,s,e)^S = \emptyset$;
13. **If** s is an up-state **Then**
14. $P(i,s,e)^I = VT^I(i,s,e).left$;
15. $P(i,s,e)^S = VT^S(i,s,e).left$;
16. **Else**
17. $P(i,s,e)^I = VT^I(i,s,e).right$;
18. $P(i,s,e)^S = VT^S(i,s,e).right$;
19. **End of if**
20. **If** $P(i,s,e)^I.val \neq 0$ **Then**
21. $P(i,s,e)^I$ =REMOVE($VT^I(i,s,e)$);
22. **End of if**
23. **If** $P(i,s,e)^S.val \neq 0$ **Then**
24. $P(i,s,e)^S$ =REMOVE($VT^S(i,s,e)$);
25. **End of if**
26. $P(i,s,e)$ =MERGE($P(i,s,e)^D, P(i,s,e)^I, P(i,s,e)^S$)
27. **End of for**

Figure 10: Updating the stacks. Each type of state needs its own updating procedure.

procedures build the e-level tree $P(i,s,e)$ by merging the trees of $VT(i,s,e)$. From the definition of $VT(i,s,e)$, $(j,t,e') \in VT(i,s,e)$. As $\alpha_s = \alpha_t$ and $\alpha_t \in P(j,t,e')$, then we have $\alpha_s \in P(i,s,e)$.

2. Suppose now that s is a marked sl-state. **(i)** If E is a deletion edge, then $\alpha_s = \alpha_t$. By the recurrence hypothesis, $P(j,t,e')$ contains α_t. As $VT^D(i,s,e)$ contains (j,t,e'), Line 5 of *Update-sl-Stack* insures that α_t is in the tree $P(i,s,e)^D$. Therefore, after the merging of Line 8, α_t is in the tree $P(i,s,e)$.

   (ii) If E is an insertion or a substitution edge, then from Tr_2, $\alpha_s = \nu(s)\,\alpha_t$. By the recurrence hypothesis, $P(j,t,e')$ contains α_t. As $VT^I(i,s,e)$ or $VT^S(i,s,e)$ contains (j,t,e'), Lines 6 and 7 of *Update-sl-Stack* insert $\nu(s)$ at the top of $P(j,t,e')$. Therefore, after the merging of Line 8, $\nu(s)\,\alpha_t$ is in the tree $P(i,s,e)$.

3. Suppose now that s is a marked sr state. **(i)** If E is a deletion edge, then $\alpha_s = \alpha_t$. In that case, *Update-sr-Stack* is equivalent to *Update-sl-Stack*, and thus the proof is identical to (2.i).

   (ii) If E is an insertion or a substitution edge, then from Tr_3, if $\alpha_t = \nu(s)\,\alpha'$, $\alpha_s = \alpha'$. By the recurrence hypothesis, $P(j,t,e')$ contains α_t. As $VT^I(i,s,e)$ or $VT^S(i,s,e)$ contains (j,t,e'), Lines 12 to 18 keeps α_t in $P(i,s,e)^I$ or $P(i,s,e)^S$, and then Lines 19 to 24 remove $\nu(s)$ out of the top of the stack α_t. Therefore, after the merging of Line 25, α' is in the tree $P(i,s,e)$.

"$\Rightarrow$" Let $\alpha \in P(i,s,e)$. The α comes from a set of alignments, all corresponding to a single path $\Pi(s)$ in the ε-NFA PushA(s). This path is obtained by removing all the insertion edges, and considering substitution and deletion edges as valid transitions of PushA(s), without considering the error level. By Lemma 2, this path is valid. As $VT(i,s,e)$ is defined in order to increase the number of errors according to the edit distance, there exists an alignment, that reduces to $\Pi(s)$ in PushA(s), with e-errors □

5 Approximate matching algorithm

5.1 Basic algorithm

In the last section, the problem under consideration has been that of computing the value of the best alignment between G and a word of $\mathcal{L}(S)$. Our initial problem, however, is that of finding all occurrences of $\mathcal{L}(S)$ in G, with at most k errors. In the case of the classical dynamic programming method, Sellers [23] noticed that it suffices to modify the initial conditions. We reuse this approach here, initializing $P(i,\theta,e)$ to the empty tree $NULL$ for all $1 \leq i \leq n$ (not only for $i = 0$). This has the effect of making every θ-vertex a source vertex as opposed to just $(0,\theta)$. By applying the dynamic programming paradigm, one can compute the sets of stacks for every node (i,s) in increasing order of i and any topological order of V. The complete algorithm is given in Figure 11.

Theorem 1 *The value c computed by* Scanall *is the value of a least error valid path ending at (i,ϕ).*

Proof. By Lemma 3, for $0 \leq i \leq k$, the resulting set of stacks for the error level e at terminal state (i,ϕ) corresponds to valid e-error paths, and conversely, each e-error path ending in

Algorithm Scanall(G,S,k)

1. **For** $e = 0 \ldots k$ **Do** $P(0, \theta, e) = NULL$ **End of for**
2. **For** $i = 0 \ldots n$ **Do**
3. **For** each state s in topological order **Do**
4. *Update-(ε-p-u-sl-sr)-Stack*(i, s);
5. **End of for**
6. Let $c = \min\{k+1, \{e \mid VT(i, \phi, e) \neq \emptyset\}\}$;
7. **If** $c \leq k$ **Then** reports i as the position
8. of a c-occurrence of S in G.
9. **End of for**

Figure 11: Complete algorithm for searching for a secondary expression S in a genome G with at most k errors.

(i, ϕ) leads to a stack in $P(i, \phi, e)$. Consecutively, by taking as c the smallest level of error for which there exists a set of stacks (Line 7 of *Scanall*), if $c \leq k$, then c is the value of a least error valid path ending at (i, ϕ) □

5.2 Complexity

Let p be the size of the secondary expression S (the number of all characters of the network expression *NetExp*(S)), and r be the number of symbols | in S. Let n be the size of the genome being traversed.

Let us first consider the complexity of upgrading one binary tree at a single node. Each operation INSERT, REMOVE and COMBINE takes $O(1)$ time, and operation MERGE takes time proportional to the size of the final merged tree, which can not be more than $O(r)$. Thus upgrading a tree takes worst case $O(r)$ time.

For each node, each procedure *Update-(ε-p-u-sl-sr)-Stack* updates exactly $k+1$ binary trees. Thus, the worst time complexity of these procedures is $O(kr)$.

There are $O(pn)$ nodes in the alignment graph, and therefore the worst case time complexity of the whole algorithm is $O(krpn)$.

5.3 Reporting the occurrences

The algorithm *Scanall* reports all right ends of approximate matches (within a threshold k) of the secondary expression S in the genome G. Let J be the set of such right ends. The first problem is to find the set of left ends corresponding to J. In the case of a network expression R, Myers and Miller [18] noticed that the left ends are obtained by building an automaton for the reverse R^r of R [1], and scanning G in reverse. A similar result holds for secondary expressions, with the following definition of the reverse of a secondary expression:

Definition 7 (reverse) *The* **reverse** S^r *of the secondary expression* S *is the secondary expression inductively defined by:*

- *If* $S = (E, p)$, *then* $S^r = (E^r, p)$, *where* E^r *is the reverse of the network expression* E;

[1] The reverse of a network expression R is the expression R^r that matches the reverse of every word matched by R. It is obtained by inductively applying the rules $(RT)^r = T^r R^r$ and $(R|T)^r = R^r|T^r$.

- *If* $S = (E_1, p)(E_2, sl)\ S'\ (\overline{E_2}, sr)(E_3, p)$, *such that* E_1, E_2, E_3 *are network expressions and* S' *is a secondary expression, then* $S^r = (E_3^r, p)(\overline{E_2}^r, sl)\ (S')^r\ (E_2^r, sr)(E_1^r, p)$.

Now, to find the set of left ends corresponding to each right end $j \in J$, it suffices to run the algorithm *Scanall* on the reverse S^r of S and on the substring $G[j \cdots j - r - k]$ of G where r is the length of the longest possible exact match to S^r, by initializing the sets of stacks to $NULL$ only for the new origin j. More precisely, $P(j, \theta, e) = NULL$.

6 Extension to pseudo-knots and generalized secondary expressions

Let S be a secondary expression. By definition of a secondary expression, there is no network expression marked sl after a network expression marked sr in S. We define the left part of S, denoted by S_l, as the prefix of S ending with the last sl network expression. The right part of S, denoted by S_r, is the remaining suffix of S.

6.1 Pseudo-knots

A specific grammar for expressing RNA motifs including pseudo-knots has been defined in [20]. We present a simpler representation using two nested secondary expressions, for simple pseudo-knots.

Definition 8 (pseudo-knot) *A pseudo-knot is an expression of form* $S_l^1\ S_l^2\ S_r^1\ S_r^2$, *where* S^1 *and* S^2 *are two secondary expressions.*

To search for a pseudo-knot with at most k errors, we directly extend *Scanall* (Figure 11) by managing two blocks of $k+1$ sets of stacks, one for $S_l^1\ S_r^1$, and the other for $S_l^2\ S_r^2$. As these two "pseudo-helices" do not constrain one another, these two stack blocks can be managed independently. More precisely, for each node (i, s), if s corresponds to a state of one pseudo-helix, then the sets of stacks corresponding to the second pseudo-helix are transferred without any change. This is done by procedure *Update-neutral-Stack*.

```
Procedure Update-neutral-Stack(i,s)
1.  For  e = 0...k Do
2.       P(i,s,e) =MERGE{P(j,t,e), where (j,t) → (i,s)
3.            is an insertion, deletion or substitution edge}
4.  End of for
```

Figure 12: Transferring the stacks of one pseudo-helix to a state corresponding to the second pseudo-helix.

Theorem 2 *The value* c *computed by* Pseudo-knot *is the value of a least error valid path ending at* (i, ϕ).

Proof. The main argument is that *Update-neutral-Stack* ensures that the stacks corresponding, respectively to $S_l^1\ S_r^1$ and $S_l^2\ S_r^2$ are independent.

Algorithm Pseudo-knot(G,S^1,S^2,k)

1. **For** $e = 0 \ldots k$ **Do** $P^1(0, \theta, e) = NULL$; $P^2(0, \theta, e) = NULL$ **End of for**
2. **For** $i = 0 \ldots n$ **Do**
3. **For** each state s in topological order **Do**
4. **If** s is in S^1 **Then**
5. *Update*^{1}*-(ε-p-u-sl-sr)-Stack*(i, s);
6. *Update*^{2}*-neutral-Stack*(i, s);
7. **Else**
8. *Update*^{1}*-neutral-Stack*(i, s);
9. *Update*^{2}*-(ε-p-u-sl-sr)-Stack*(i, s);
10. **End of if**
11. **End of for**
12. Let $c^1 = \min\{k+1, \{e \mid VT^1(i, \phi, e) \neq \emptyset\}\}$;
13. Let $c^2 = \min\{k+1, \{e \mid VT^2(i, \phi, e) \neq \emptyset\}\}$;
14. If $c = c^1 + c^2 \leq k$ then reports i as the position
15. of a c-occurrence of $S_l^1\ S_l^2\ S_r^1\ S_r^2$ in G.
16. **End of for**

Figure 13: Searching for a pseudo-knot $S_l^1\ S_l^2\ S_r^1\ S_r^2$ in a genome G with at most k errors. P^1 (respec. P^2) represents the set of $k+1$ stacks corresponding to S^1 (respec. S^2). VT^1 (respec. VT^2) corresponds to the valid tails linked to the nodes of S^1 (respec. S^2). Update^1 (respec. Update^2) corresponds to the procedures of Figure 10 applied to P^1 (respec. P^2).

In consequence, by computing c^1 (respec. c^2) (Lines 12, 13 of *Pseudo-knot*) exactly as in *Scanall*, c^1 (respec. c^2) corresponds to the value of a least error path traversing $S_l^1\ S_r^1$ (respec. $S_l^2\ S_r^2$). Therefore, $c = c^1 + c^2$ is the value of a least error valid path ending at (i, ϕ).

Conversely, suppose there exists a least c-error valid path ending at (i, ϕ), such that $c \leq k$. This path goes through $S_l^1\ S_r^1$ and $S_l^2\ S_r^2$ in an independent way, that is the two parts of the path do not constrain one another. Suppose one of the two parts is not a least error path. This would contradict the fact that c^1 and c^2 are the minimum of such error paths □

6.2 Generalized secondary expressions

All the concepts and algorithms described above for searching a single helix can be generalized to secondary expressions representing complex structures with an arbitrary number of helices. Such a generalized secondary expression is defined as follows.

Definition 9 (generalized secondary expressions) *S is a generalized secondary expression if and only if one of the three recursive conditions is verified:*

- *S is a secondary expression;*
- *$S = (E, sl)S'(\overline{E}, sr)$ where S' is a generalized secondary expression and E is a network expression;*
- *$S = S_1 S_2$ where S_1 and S_2 are two generalized secondary expressions.*

An example of such a generalized secondary expression is shown in Figure 14. The ε-NFPA recognizing a generalized secondary expression S is based on the ε-NFA recognizing

the network expression $NetExp(S)$, and constructed in a similar way than for a simplified secondary expression, except that the automaton is not restricted to one left strand and one right strand. Indeed, each helix of the generalized secondary expression has a left strand and a right strand (with possibly unpaired regions). The states are then pushed and popped from the $k+1$ sets of stacks depending on their type (sl-state, sr-state, p-state) in the same way than for simple secondary expressions (Figure 10).

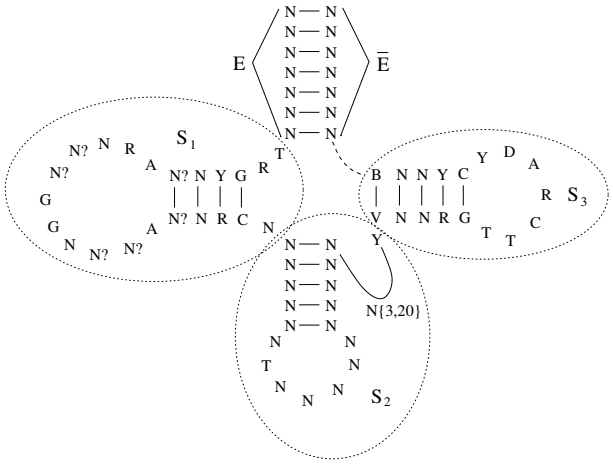


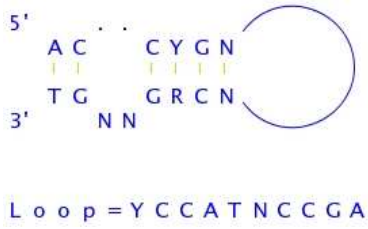
Figure 14: A generalized secondary expression representing a complete tRNA cloverleaf. The formal representation is: $GS = (E, sl)S_1S_2S_3(E, sr)$ where E is a network expression and S_1, S_2, S_3 are secondary expressions.

7 Practical evaluation

The algorithm has been implemented as a prototype called BIOSMATCH (for BIOlogical Secondary MATCHing), with a graphical interface allowing to represent the structure in a natural way. The program reports the final positions of all the occurrences of a helix in a genome.

We do not compare our program to Eddy and Durbin's program [3], as they are conceptually different, and should be used in different contexts. The Eddy and Durbin's stochastic approach requires a family of RNAs, and deduces a stochastic context free based model from a multiple alignment of these RNAs, whereas BIOSMATCH is a deterministic program that is helpful when the user knows the consensus structure that is looking for. The only algorithms that are conceptually similar to ours are RNAMOT [9], RNABOB [4] and RNAMOTIF [15]. They are all based on the same approach, though RNAMOTIF is the most matured program, allowing for mismatches, mispairings, and regular expressions. The helix is modeled as a grammar of primary and secondary constraints, and a backtracking search strategy is considered.

We compared the results of BIOSMATCH and RNAMOTIF, as well as the flexibility of the helix representation, for the 5S ribosomal RNA subunit. This gene, about 120 nucleotides long, is formed of 5 regions, the most conserved one being region III. It constitutes a good anchor for a research strategy. Figure 15 gives a bacterial consensus of region III, as reported in [24].




```
parms
        wc += gu;

descr
        h5(len=2, seq="^AC$")
        h5(len=4, seq="^CYGN$")
        ss(seq="^YCCCATNCCGAAC$")
        h3
        ss(len=2)
        h3
```

Figure 15: The mitochondrial consensus of the 5S rRNA helix III, as reported in [24]. The left figure is the BIOSMATCH graphical representation, and the right figure is the corresponding RNAMOTIF descriptor without errors

Table 2 gives the false positive and false negative results for a set of completely sequenced and annotated bacterial genomes from different taxonomic families. These results are obtained by comparing the algorithms outputs with the annotated sequences of 5S rRNA. We first notice that helix III is a very good anchor in a general strategy for RNA 5S identification. Indeed, except in the case of running BIOSMATCH with 5 errors, the obtained specificity and sensitivity are close to 100% with BIOSMATCH.

To search for approximate occurrences with RNAMOTIF, we have to specify the number of mispairings and mismatches allowed in each part of the structure. A general score constraining the total number of errors can also be defined. The first part of Table 2 gives the results obtained by RNAMOTIF when we allow for three errors in each part of the structure, and three errors in total. These results are comparable to those obtained by running BIOSMATCH with one and two errors. All missed genes, except those in *C. jejuni* that highly diverge from the consensus, are obtained by running BIOSMATCH with three errors.

The difference between the results obtained by RNAMOTIF and BIOSMATCH with three errors is due to the fact that RNAMOTIF does not allow for insertions and deletions. It is possible to modify the RNAMOTIF descriptor in order to find these structures. However, we should know, in advance, in which part of the structure insertions/deletions are located. In other words, RNAMOTIF should be run many times with different descriptors.

	RNAMOTIF 3 errors		BIOSMATCH 1 error		BIOSMATCH 2 errors		BIOSMATCH 3 errors		BIOSMATCH 5 errors	
Organism	False	Missed	False	Missed	False	Missed	False	Missed	False	Missed
M. thermo	0	2	0	2	0	2	1	0	78	0
Halobacterium sp.	0	1	0	1	0	1	0	0	284	0
S. solfataricus	0	0	0	1	0	1	0	0	13	0
B. subtilis	0	0	0	0	0	0	0	0	82	0
M. leprae	0	1	0	1	0	0	1	0	278	0
C. jejuni	0	3	0	3	0	3	0	3	3	0
C. diphtheriae	0	5	0	5	0	5	2	0	128	0
E. coli	0	0	0	0	0	0	0	0	167	0
P. aeruginosa	0	0	0	0	0	0	1	0	653	0
S. pneumoniae	0	0	0	0	0	0	0	0	27	0

Table 2: Results of searching for the consensus of the 5S rRNA helix III (Figure 15) in a collection of bacterial genomes taken from GenBank. "False" correspond to the sequences that have been found by the algorithm, but do not correspond to annotated 5S rRNAs. "Missed" correspond to annotated 5S RNAs sequences that have not been found by the algorithm.

Figure 16: A possible secondary expression representing the $P4$ region of the RNase P RNA.

To test BIOSMATCH on a helix with a large loop, we have searched for the $P4$ region of the RNase P RNA (Figure 16 (b)) in the two strands of the *Reclinomonas americana* mitochondrial genome (AF007261 entry name in GenBank - 141 KB - 1 annotated RNase P RNA). The experiments have been performed on a PC running Linux 3.0.3-8 Red Hat, with an 1.2 GHz Intel Pentium4 processor. Table 3 shows the results. The $P4$ secondary expression has been deduced from [11].

Error(s)	0			1			2		
	Fpos	Fneg	Time	Fpos	Fneg	Time	Fpos	Fneg	Time
RNase P	0	0	1 : 34	0	0	2 : 21	0	0	9 : 45

Table 3: Results of searching RNase P RNA's P4 domain in the two strands of Reclinomonas americana (141 KB - 1 RNase P RNA). Times are given in minutes and seconds.

These experiments show the flexibility of our approach to search for different kinds of helices in a reasonable time, even for helices with large loops, weak constraints (several N and N?) and when allowing indel/mismatches.

8 Conclusion

We have presented a method to search for conserved secondary structures in a sequence. It allows for a flexible representation of helices: each position is defined by a class of characters or by any other network expression, stems and loops can be of variable length, loops can be very large, and bulges and internal loops can be specified. The algorithm accounts for a possible deviation from the consensus. Moreover, the method is naturally extendable to pseudo-knots and general structures containing an arbitrary number of helices.

To our knowledge, this work represents the first attempt to take advantage of the possibilities of pushdown automata in the context of approximate matching or RNA representation. As a pushdown automaton recognizes a context free langage, an alternative approach would have been to use a context free grammar representation, as considered in [16].

This method is a well founded framework for secondary structures approximate matching. As for the case of simpler patterns, this is a prerequisite for developing efficient filters that allow faster searches in practice. As a drawback, developing such filters is more difficult for secondary structures than for network expressions, since the type of filtering must depend on the nature of the constraints in the secondary expression.

Acknowledgments

Research supported by grants from the Ministry of Research, Science and Technology, on the program *Assistance financière à la coopération scientifique et technologique*, and from France-Canada research Funding. NE-M is affiliated with the Evolutionary Biology Program of the Canadian Institute for Advanced Research. MR is a *Centre National de Recherche Scientifique* researcher.

References

[1] B. Billoud, M. Kontic, and A. Viari. Palingol: a declarative language to describe nucleic acids' secondarys tructures and to scan sequence databases. *Nucleic Acids Research*, 24(8):1395-1403, 1996.

[2] A. Cornish-Bowden. Nomenclature for incompletely specified bases in nucleic acid sequences: recommendations 1984. *Nucleic Acids Research*, 13:3021-3030, 1985.

[3] S. Eddy and R. Durbin. RNA sequence analysis using covariance models. *Nucleic Acids Research*, 22:2079-2088, 1994.

[4] S. R. Eddy. RNABOB: a program to search for RNA secondary structure motifs in sequence databases. `http://bioweb.pasteur.fr/docs/man/man/rnabob.1.html#toc1`, 1992.

[5] N. El-Mabrouk and F. Lisacek. Very fast identification of RNA motifs in genomic DNA. Application to tRNA search in the yeast genome. *Journal of Molecular Biology*, 264:46–55, 1996.

[6] N. El-Mabrouk and M. Raffinot. Approximate matching of secondary structures. In *Proceedings of the sixth annual international conference on computational molecular biology (RECOMB)*, pages 156-164. ACM press, 2002.

[7] G.A. Fichant and C. Burks. Identifying potential tRNA genes in genomic dna sequences. *Journal of Molecular Biology*, 220:659- 671, 1991.

[8] C. Gaspin, J. Cavaille, G. Erauso, and J. P. Bachellerie. Archaeal homologs of eukaryotic methylation guide small nucleolar RNAs:lessons from the pyrococcus genomes. *Journal of Molecular Biology*, 297, 2000.

[9] D. Gautheret, F. Major, and R. Cedergren. Pattern searching/alignment with RNA primary and secondary structures. *Comput. Appl. Biosci.*, 6(4):325–331, 1990.

[10] S. Graf, D. Strothmann, S. Kurtz, and G. Steger. Hypalib: a database of rnas and rna structural elements defined by hybrid patterns. *Nucleic Acids Research*, 29(1):196-198, 2001.

[11] B.F. Lang, G. Burger, C.J. O'Kelly, R.J. Cedergren, B. Golding, C. Lemieux, D. Sankoff, M. Turmel, and M.W. Gray. An ancestral mitochondrial dna resembling a eubacterial genome in miniature. *Nature*, 387:493–497, 1997.

[12] F. Lisacek, Y. Diaz, and F. Michel. Automatic identification of group I introns cores in genomic DNA sequences. *J. Mol. Biol.*, 235:1206-1217, 1994.

[13] T.M. Lowe and S.R. Eddy. tRNAscan-SE: a program for improved detection of transfer rna genes in genomic sequence. *Nucleic Acids Research*, 25:955-964, 1997.

[14] T.M. Lowe and S.R. Eddy. A computational screen for methylation guide snoRNAs in yeast. *Science*, 283(5405):1168-1171, 1999.

[15] T. Macke, D. Ecker, R. Gutell, D. Gautheret, D.A. Case, and R. Sampath. RNAmotif - a new RNA secondary structure definition and discovery algorithm. *Nucleic Acids Research*, 29:4724-4735, 2001.

[16] E.W. Myers. Approximately matching context-free languages. *Information Processing Letters*, 54(2):85–92, 1995.

[17] E.W. Myers. Approximate matching of network expression with spacers. *Journal of Molecular Biology*, 3(1):33–51, 1996.

[18] E.W. Myers and W. Miller. Approximate matching of regular expressions. *Bull. of Mathematical Biology*, 51(1):5–37, 1989.

[19] A.D. Omer, T.M. Lowe, A.G. Russell, H. Ebhardt, S.R. Eddy, and P.P. Dennis. Homologs of small nucleolar RNAs in archaea. *Science*, 288(5465):517-522, 2000.

[20] E. Rivas and S. R. Eddy. The language of RNA: a formal grammar that includes pseudoknots. *Bioinformatics*, 16(4):334–340, 2000.

[21] M.F. Sagot and A. Viari. Flexible identification of structural objects in nucleic acid sequences: palindromes, mirror repeats, pseudoknots and triple helices. In A. Apostolico and J. Hein, editors, *Lecture Notes in Computer Science*, volume 1264 of *Eighth Combinatorial Pattern Matching Conference*, pages 224–246. Springer, 1997.

[22] Y. Sakakibara, M. Brown, R. Hughey, S. Mian, K. Sjölander, and R. Underwood. Stochastic context-free grammars for trna modeling. *Nucleic Acids Research*, 22(23):5112-5120, 1994.

[23] P.H. Sellers. The theory and computation of evolutionary distances: pattern recognition. *J. Algorithms*, 1:359- 373, 1980.

[24] M. Szymanski, M.Z. Barciszewka, J. Barciszewski, and V.A. Erdman. 5S ribosomal RNA data bank. *Nucleic Acids Research*, 27(1):158-160, 1999.

[25] K. Thompson. Regular expression search algorithm. *Communications of the ACM*, 11:419–422, 1968.