Algorithms Seminar 2001–2002,
F. Chyzak (ed.), INRIA, (2003), pp. 107–112.

Available online at the URL
http://algo.inria.fr/seminars/.

Approximate Matching of Secondary Structures

Matthieu Raffinot

Génopôle, Université d'Évry (France)

February 25, 2002

Summary by Pierre Nicodème

Abstract

This talk presents an algorithm to search for all approximate matches of a helix in a genome, where a helix is a combination of sequence and folding constraints. It is a joint work with Nadia El-Mabrouk of University of Montréal and was presented at the RECOMB 2002 congress [1]. The method applies for more general secondary RNA structures including several helices.

1. Introduction

We give in this section an intuitive description of the problem considered and of the method used. We refer to the next section for more precise definitions. We consider the alphabet $\Sigma = \{A, C, G, T\}$ of DNA. RNA molecules are subject to Watson–Cricks base-pairings constraints, where the pairs are $A \leftrightarrow T$ and $C \leftrightarrow G$. A *network expression* over $\Sigma^\star$ is a regular expression built with the union and concatenation operators. The *complement* $\overline{w}$ of a word w is obtained by reversing the order of the letters of a word and by taking the pairing letter for each letter. For instance,

$$\text{complement}(AAGT) = \overline{AAGT} = ACTT.$$

The complement $\overline{E}$ of a network expression E is the set of complements of the words of the language defined by E. A secondary expression S is of the form

$$S = N_1 E_1 N_2 E_2 N_3 \dots N_3' \overline{E_2} N_2' \overline{E_1} N_1',$$

where the N_i, N_i', and E_i are network expressions. The $E_1, E_2, \dots$ (resp. $\overline{E_1}, \overline{E_2}, \dots$) are marked *sl* (resp. *sr*) for left (resp. right) strands. Figure 1 represents an example of secondary structure,

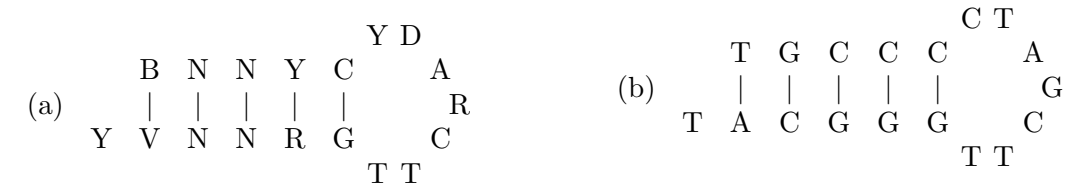

FIGURE 1. (a) A secondary expression S representing a signature for the $T\Psi C$ region of tRNAs; (b) An occurrence of the secondary expression S.

where $B = C|G|T$, $N = A|C|G|T$, $Y = C|T$, $D = A|G|T$, $R = A|G$ and $V = A|C|G$. With the same definition for the letters B, N, Y, D, R, and V, and the network expression E defined by $E = BNNYC$, this secondary structure may be written $EYDARCTT\overline{E}Y$. The problem is to

find all occurrences of such a structure in a DNA text. The more general approximate matching problem searches for matches with errors.

The algorithm goes along the following steps for matching with a secondary expression S.

1. Build a deterministic finite automaton $\mathcal{A}$ recognizing the language $\mathcal{S}$ defined by S when pairing constraints are erased.
2. Build over $\mathcal{A}$ a pushdown automaton $\mathcal{P}$. This automaton is designed to memorize which choices are made each time a union symbol $|$ is met during the reading of the left strands of S (stacking phase), and to constraint the path followed during the reading of the right strands (unstacking phase).
3. When matching with errors is considered with a sequence of size n, an alignment graph is built with $n+1$ copies of the pushdown automaton $\mathcal{P}$ and a dynamical programming method is used to find the best alignment. Different valid (in the sense of the unstacking constraints) paths may lead to the same state, and it is therefore necessary to maintain during the dynamical programming step *sets of stacks*.

Note that Myers and Miller [2] give an algorithm to find approximate matching of regular expressions with complexity $O(np)$, where n is the size of the sequence and p is the size of the regular expression; this method applies to primary structures, but not to secondary structures.

2. Definitions

Definition 1 (network expression)**.** For $\alpha \in \Sigma \cup \{\epsilon\}$, the symbol α is a network expression. If E_1 and E_2 are network expressions, $E_1|E_2$ and E_1E_2 are network expressions.

Definition 2. The set *NetSet* is the set of network expressions.

Definition 3 (complement)**.** The complement $\overline{E}$ of a regular expression is defined by: *(i)* $\overline{\epsilon} = \epsilon$, *(ii)* $\overline{A} = T$, $\overline{T} = A$, $\overline{C} = G$, $\overline{G} = C$, *(iii)* $\overline{E_1E_2} = \overline{E_2}\,\overline{E_1}$ and $\overline{E_1|E_2} = \overline{E_1}|\overline{E_2}$.

Definition 4 (secondary expression)**.** A secondary expression is a sequence of elements of $\mathit{NetSet} \times \{p, sl, sr\}$, where p, sl, and sr respectively label unpaired, left strand, and right strand network expressions. The set of secondary expressions is recursively defined by: *(i)* if E is a network expression, then $S = (E, p)$ is a secondary structure; *(ii)* if E_1, E_2, E_3 are network expressions, 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.

Definition 5. The language $\mathcal{L}(S)$ specified by a secondary expression S is recursively defined 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

$$\mathcal{L}(S) = \left\{ u \in \Sigma^\star \;\middle|\; u = vwx\overline{w}z \text{ for } v \in \mathcal{L}(E_1), w \in \mathcal{L}(E_2), z \in \mathcal{L}(E_3), \text{ and } x \in \mathcal{L}(S') \right\}.$$

Definition 6. For a secondary expression S, the *NetSet* expression $\mathit{NetSet}(S)$ is obtained by erasing the labels in the secondary expression.

As an example, if $S = (E_1, sl)(E_2, p)(\overline{E_1}, sr)$, then $\mathit{NetSet}(S) = E_1E_2\overline{E_1}$.

Definition 7 (approximate match)**.** Given a scoring function δ between two sequences (hamming distance, edit distance, measure of similarity), the set of sequences approximately matching a secondary expression S within k under scoring function δ is $\mathcal{L}_\delta(S, k) = \left\{ A \;\middle|\; \exists B \in \mathcal{L}(S), \delta(A, B) \leq k \right\}$. Note that this defines approximate matching of *primary* structures (sequences).

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 the automaton is exponential in the number of symbols | in S. Using a pushdown automaton gives a more efficient algorithm.

El-Mabrouk anf Raffinot use a state labelled[1] finite pushdown automaton referred to later as ϵ-NFPA. Formally, an ϵ-NFPA $\mathcal{P} = \langle \Sigma, \Gamma, V, E, \lambda, \gamma, \theta, \phi, I \rangle$ consists of:

- an input alphabet Σ;
- a stack alphabet Γ;
- a set V of vertices called states;
- a set E of directed edges between vertices;
- a mapping λ of V on $\Sigma \cup \{\epsilon\}$;
- a mapping γ of $V \times (\Sigma \cup \{\epsilon\}) \times \Gamma$ on a finite subset of $V \times \Gamma^\star$;
- an initial state θ;
- a final state ϕ;
- a particular stack symbol $I \in \Gamma$ called the start symbol.

If s and t are states, l is a letter of $\Sigma \cup \{\epsilon\}$, and the value of the top of the stack is Z, the interpretation of $\gamma(t, l, Z) = (s, \alpha)$, with $\alpha \in \Gamma^\star$ is that the automaton moves from state s to state t while reading letter l, popping Z from the top of the before pushing α into the stack. From there follows a partial mapping μ of $(V, \Sigma^\star, \Gamma^\star)$ onto itself defined by

$$(t, lw, Z\beta) \stackrel{\mu}{\longmapsto} (s, w, \alpha\beta) \quad \text{if} \quad \gamma(t, l, Z) = (s, \alpha).$$

Let $\mu^\star$ be the transitive closure of μ. The language accepted by the pushdown automaton $\mathcal{P}$ is

$$\mathcal{L}(\mathcal{P}) = \left\{ w \;\middle|\; (\theta, w, I) \stackrel{\mu^\star}{\longmapsto} (\phi, \epsilon, \alpha),\ \alpha \in \Gamma^\star \right\}.$$

(Note that by construction, for secondary structures, we have $\alpha = \epsilon$ in the last equation.) The letter μ will be omitted in what follows.

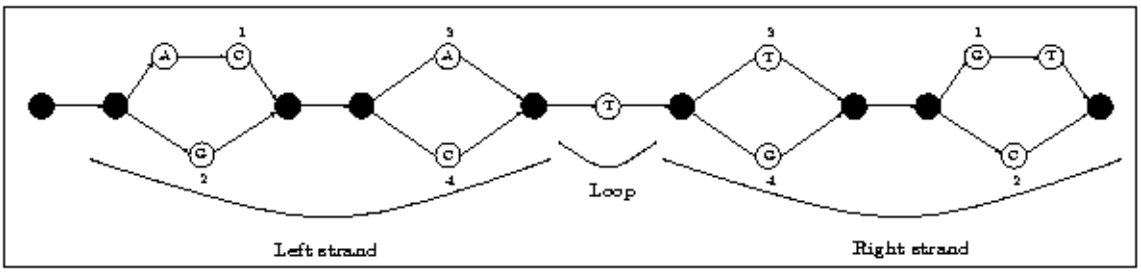

FIGURE 2. The state labelled ε-NFA recognizing *NetExp*(S), for $S = (E_1, sl)(E_2, p)(\overline{E_1}, sr)$, with $E_1 = ((AC)|G)(A|C)$ and $E_2 = T$. Black states are labelled by ε. The numbers $1, 2, 3, 4$ mark the marked states. The loop is an unpaired region.

The construction of the automaton $\mathcal{P}$ recognizing S goes along the following steps:

1. build a state-labelled ϵ-NFA $\mathcal{A}$ recognizing Netset(S), with labelling function λ;
2. mark the possible choices for each union symbol | of the left strands of S;
3. define the rules for stacking the marks during reading the left strands of S;
4. define the unstacking and transitions rules while reading the right strands of S.

[1] A corresponding classical transition labelled automaton would be such that all the transitions entering a state are labelled with the same letter of $\Sigma \cup \{\epsilon\}$, whatever this state is.

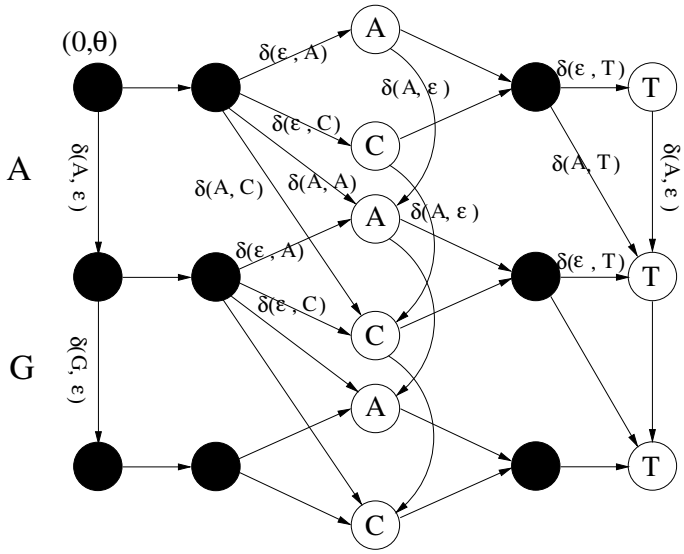

FIGURE 3. The alignment graph for a sequence $Q = AG$ versus the network expression $(A|C)T$.

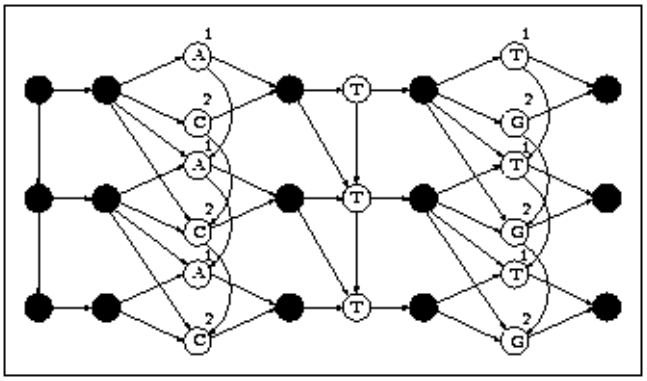

FIGURE 4. The alignment graph for $Q = \mathrm{AT}$ versus $NetExp(S) = (A|C)T(T|G)$.

Marking the states. For each $(E_{i_1}|E_{i_2})$ expression of S, where neither E_{i_1} nor E_{i_2} contains a union symbol $|$, and E_{i_1} and E_{i_2} are left strands expressions, let s_{i_j} be the state of $\mathcal{A}$ corresponding to the last atomic expression of E_{i_j} $(j = 1, 2)$. Each such state is marked unambiguously with a letter γ of Γ (two different states are marked by different letters). The other states of the left strands and the states of unpaired regions remain unmarked. Mark the states of the right strands by mirroring the corresponding left strands. An example of marking is given in Figure 2 where Γ is a subset of $\mathbb{N}^+$. Let ν be the mapping associating to a marked state s its mark $\nu(s)$.

Defining the mapping γ. The mapping γ of the pushdown automaton $\mathcal{P}$ is defined as follows:

Let Z be the top symbol of the stack, l be any character of $\Sigma \cup \epsilon$, s be any state, and $t \to s$ be any edge leading to s in the automaton $\mathcal{A}$. The transition $\gamma(t, l, Z)$ is defined in the automaton $\mathcal{P}$ if and only if $l = \lambda(s)$. In that case:

- if s is an unmarked state, then $\gamma\big(t, \lambda(s), Z\big) = (s, Z)$;
- if s is a marked sl-state, then $\gamma\big(t, \lambda(s), Z\big) = \big(s, \nu(s)Z\big)$;
- if s is a sr-state such that $\nu(s) = Z$, then $\gamma\big(t, \lambda(s), Z\big) = (s, \epsilon)$.

This definition of γ constrains the traversal of the right strands to be the mirror of the traversal of the corresponding left strand.

Lemma 1. *The pushdown automaton $\mathcal{P}$ recognizes the language generated by the secondary expression S.*

See [1] for a proof.

4. Matching with Errors and Alignment Graph

For the problem of aligning a network expression E to a sequence Q of size n within a threshold k, Myers and Miller[2] showed in [2] 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 Q. The graph is constructed from $n + 1$ copies of the ϵ-NFA recognizing E, arranged one on top of another. Figure 3 shows an alignment graph of the expression $E = (A|C)T$ and of the sequence $Q = AG$.

Formally, the vertices of the graph are the pairs (i, s) for $1 \leq i \leq n + 1$ and $s \in V$. Insertion, deletion and substitution edges are defined as follows:

- if $i > 0$, then there is a *deletion edge* from $(i - 1, s)$;
- if $s \neq \theta$, then for each state t such that $t \to s$, there is an *insertion edge* from (i, t);
- if $i > 0$ and $s \neq \theta$, then for each state t such that $t \to s$, there is a *substitution edge* from $(i - 1, t)$.

The construction of Myers and Miller is applied to the pushdown automaton $\mathcal{P}$. Figure 4 shows the alignment obtained when matching $Q = AT$ against $NetExp(S) = (A|C)T(T|G)$, with $S = (A|C)T(\overline{A|C})$. The problem is that several paths may lead to the same state; it is therefore necessary to maintain sets of stacks. For a state (i, s), let $\Pi(i, s)$ be the set of least cost paths from $(0, \theta)$ to (i, s). For a path $\pi \in \Pi(i, s)$ let $\sigma(\pi)$ be the sequence obtained by concatenating the labels λ of the states on this path. The set of stacks of a state (i, s) is defined by

$$\mathrm{Stack}(i, s) = \big\{\, \alpha \in \Gamma^\star \;\big|\; \exists \pi \in \Pi(i, s) \text{ such that } (\theta, \sigma(\pi), I) \xrightarrow{\star} (s, \epsilon, \alpha) \,\big\}.$$

A path aligning the first i letters $Q[1, i]$ of Q and a sequence $\sigma(\pi)$ for a state s is a *valid path* if it respects the constraints given by the secondary expression. Therefore $\sigma(\pi)$ must belong to the language recognized by $\mathcal{P}(s)$, where s is made the final state of $\mathcal{P}$.

An edge from (j, t) to (i, s) is valid (noted $(j, t) \xrightarrow{v} (i, s)$) if the two following conditions are met:

- $(j, t) \to (i, s)$ is an insertion, deletion or substitution edge;
- if $(j, t) \to (i, s)$ is a substitution or deletion edge and s is a marked sr-state, then there is a stack P in $\mathrm{Stack}(j, t)$ with top symbol $\lambda(s)$.

Thus the problem of approximately matching a prefix of size i of Q to a prefix $\sigma(\pi)$ of a word of $\mathcal{L}(S)$ is equivalent to finding a least cost valid path between source vertex $(0, \theta)$ and (i, s). Computing such a path may be done by dynamic programming (procedure *CentralRec* of Figure 5).

[2]There is an error in this section that follows the content of the talk: a suboptimal left strand alignment may lead to an optimal right strand alignment. El-Mabrouk and Raffinot are working at correcting this error, that is compatible with Myers and Miller's approach.

procedure CentralRec:

1. $C(0, \theta) = 0$
2. $C(i, s) = \min_{(i,t) \xrightarrow{v} (i,s)} \{C(i, t) + \delta(\varepsilon, \lambda(s))\}$
3. **if** $(i-1, t) \xrightarrow{v} (i, s)$ **then**
4. $\quad C(i, s) = \min\{C(i, s), C(i-1, t) + \delta(q_i, \lambda(s))\}$
5. **if** $(i-1, s) \xrightarrow{v} (i, s)$ **then**
6. $\quad C(i, s) = \min\{C(i, s), C(i-1, s) + \delta(q_i, \varepsilon)\}$

FIGURE 5. Central recurrence computing the value of a least cost valid path from the source vertex to each vertex (i, s) of the alignment graph. The letter q_i is the letter at position i in Q.

Maintaining the set of stacks. El-Mabrouk and Raffinot implement the set of stacks as binary trees. They define a set of operations over these trees:

- *Insert*: a new node is inserted at the top of a tree;
- *Remove*: remove the top element;
- *Combine*: a new root points to trees T_1 and T_2 that were previously constructed;
- *Merge*: "superposition" of two trees; there must be coherence between the nodes of the two trees.

During the reading of the *sl*-strands, trees are grown through Insert, Combine and Merge operations, while during the reading of the right strand, the Remove operation is used, and one of the left or right tree is substituted to the tree representing the stacks.

Approximate matching algorithm. When looking for approximate matches of a secondary expression against a sequence, one alignment graph is constructed for each position of the sequence. Note that practically only two copies of the automaton $\mathcal{P}$ are maintained.

5. **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.

There are $O(np)$ vertices in the alignment graphs, and the in-degree of the vertices is at most 3. Computing the value at each vertex by *CentralRec* takes $O(1)$ time. Thus, computing all the costs $C(i, s)$ can be done in $O(np)$ time. When considering the stacks, the procedure *Merge* is $O(r)$ in the worst case (other procedures have lower complexity).

This gives a final complexity of $O(rpn)$.

As an example, scanning the 4MB of *bacillus subtillis* with a 200 base long secondary structure takes 215 seconds.

Bibliography

[1] El-Mabrouk (N.) and Raffinot (M.). – Approximate matching of secondary structures. In *Sixth Annual International Conference on Computational Molecular Biology*. pp. 156–164. – ACM Press, 2002.

[2] Myers (Eugene W.) and Miller (Webb). – Approximate matching of regular expressions. *Bulletin of Mathematical Biology*, vol. 51, n° 1, 1989, pp. 5–37.