Exact Pattern Matching for RNA Secondary Structures

Ying Xu **Lusheng Wang** **Xiaotie Deng**

Department of Computer Science
City University of Hong Kong
Kowloon, Hong Kong, China
Email: {lwang, deng}@cs.cityu.edu.hk

Abstract

Many RNA structures are assembled from a collection of RNA motifs, which appear repeatedly and in various combinations. Identification of RNA structural motifs will enhance our understanding of RNA structures and functions. Searching for secondary structural patterns in sequence databases is the basic technique and fundamental problem for extracting and identifying such motifs. A number of algorithms and programs have been developed for this purpose.

In this paper, we adopt a representation of secondary structure called *secondary expressions*, and present two algorithms for finding all exact matches of a given secondary expression.

Keywords: RNA secondary structures, and exact pattern matching.

1 Introduction

RNA secondary structures play an important role in regulating gene expressions. Many of these RNA structures are assembled from a collection of RNA motifs. These basic patterns appear repeatedly and in various combinations to form different RNA types and define their unique structural and functional properties. Identification of RNA structural motifs will therefore enhance our understanding of RNA structures and their association with functional and regulatory elements.

An important technique for extracting and identifying secondary motifs is to search patterns in sequence databases. A number of algorithms and software have been developed for this purpose. Early attempts in structural motif searching were designed for specific families, e.g., FAStRNA (El-Mabrouk & Lisacek 1996) for tRNAs, and CITRON (Lisacek, Diaz & Michel 1994) for group I introns. Tools for general secondary structures appear in (Billoud, Kontic & Viari 1996, Macke, Ecker, Gutell, Gautheret, Case & Sampath 2001, Pesole, Liuni & D'Souza 2000). In those general purpose tools, description of secondary structures is very flexible, but the major drawback is that such definitions do not allow efficient searching algorithm.

Copyright ©2004, Australian Computer Society, Inc. This paper appeared at the 2nd Asia-Pacific Bioinformatics Conference (APBC2004), Dunedin, New Zealand. Conferences in Research and Practice in Information Technology, Vol. 29. Yi-Ping Phoebe Chen, Ed. Reproduction for academic, not-for profit purposes permitted provided this text is included.

You can use the /toappear again to add a second footnote for grant and other information.

In (El-Mabrouk & Raffinot 2002), a representation which extends regular expressions with pairing operators, called *secondary expressions*, is developed for describing secondary structure. Here we work on the following problem: given a secondary expression P and a string T, we want to find all (exact) occurrences of P in T.

If P is a string, there are many elegant linear or sub-linear algorithms (Gusfield 1997). If P is a regular expression of size m, one can convert the expression into a nondeterministic finite automaton (NFA) with $O(m)$ nodes and search in $O(mn)$ time, where n is the length of the text string (Baeza -Yates 1996). Another choice is to convert the expression into a DFA and search in $O(n)$ time. However, the DFA might have 2^m states (Baeza -Yates 1996).

In this paper, we give algorithms for finding all exact matches of a given secondary expression in $O(nm^2)$ time, where m is the size of the secondary expression and n is the size of the text. The rest of this paper is organized as follows. Definitions and preliminaries are given in Section 2. In Section 3, we propose a dynamic programming algorithm to solve the problem. Section 4 describes a more efficient algorithm by keeping history links.

2 Preliminaries

We first give the definition of secondary expressions and the language they accept. We then discuss how to convert a secondary expression into a finite automaton.

2.1 Secondary expressions

A *network expression* over alphabet Σ is any expression built up with the operations, *concatenation* and *alternation* ($|$), using the symbols in $\Sigma \cup \{\varepsilon\}$, where ε denotes an empty string. Network expressions are, in fact, a subset of regular expressions by excluding the operator of Kleene closure. The motivation of this definition is that the Kleene closure operator is not so useful in most applications in molecular biology (Myers 1996).

The *complement of network expression* E over alphabet Σ, denoted by E', is the network expression defined recursively by: (1) if $E = \varepsilon$, then $E' = \varepsilon$; (2) if $E = A(, U, G, C)$, then $E' = U(, A, C, G)$; (3) if $E = E_1E_2$, then $E' = E'_2E'_1$; (4) if $E = E_1|E_2$, then $E' = E'_1|E'_2$.

For example, the complement expression of $(A|AG)CC$ is $GG(U|CU)$.

The RNA secondary structure may include paired and unpaired regions, so we assign a property ('p', 'sl', or 'sr') to a network expression to indicate its pairing state. An expression marked with 'p' represents an unpaired region (which might be a loop, a bulge, or

an internal loop), 'sl' is the left strand of a paired region, and 'sr' is the right strand.

A *region* is a pair (E, s), where E is a network expression and s is a pairing state, p, sl, or sr. A *secondary expression* is a sequence of regions constructed recursively by:

- (R1) (E_1, p) is a secondary expression;
- (R2) if S is a secondary expression, then $(E_1, p)(E_2, sl)S(E_2', sr)(E_3, p)$ is a secondary expression, where E_2' is the complement of E_2;
- (R3) If S_1 and S_2 are two secondary expressions, then S_1S_2 is a secondary expression.

By definition, every sl or sr region is paired with a unique complement region, and an sl region always appears before its complement sr region.

Since an RNA secondary structure contains non-crossing base-pairs and unpaired bases, any RNA secondary structure can be described by a secondary expression. However, secondary expression cannot describe tertiary structures such as pseudoknots, where there are crossing base-pairs.

An important subset of secondary expression is *hairpin expressions*, which are defined by recursively applying (R1) and (R2) only. (Note that the definition of secondary expression in (El-Mabrouk et al 2002) is equivalent to our hairpin expression.) A hairpin expression can only model one stem-loop structure, but cannot describe multiple loops.

Example 1:

$$S_1 = (E_1, p)(E_2, sl)(E_3, p)(E_2', sr)(E_4, p),$$

where

$$E_1 = AA|A, E_2 = A|AG, E_3 = G|\varepsilon, E_4 = T,$$

is a hairpin expression, and also a secondary expression.

$$S_2 = S_1(E_5, p)(E_6, sl)(E_7, p)(E_6', sr)(E_8, p),$$

where

$$E_6 = U, E_5 = E_7 = E_8 = \varepsilon,$$

is a secondary expression, but not a hairpin expression.

The language $L(S)$ *accepted* by a secondary expression S is defined recursively as follows: (1)if $S = (E_1, p)$, then $L(S) = L(E_1)$, where $L(E_1)$ is the language accepted by the expression E_1; (2) if $S = (E_1, p)(E_2, sl)S'(E_2', sr)(E_3, p)$, then $L(S) = \{uxvx'w | u \in L(E_1), x \in L(E_2), v \in L(S'), w \in L(E_3), x' \text{ is complement of } x\}$; (3)if $S = S_1S_2$, then $L(S) = \{uv | u \in L(S_1), v \in L(S_2)\}$.

Example 2: The language of S_1 in Example 1 is

$$\{AAUT, AAGUT, AAGCUT, AAGGCUT, AAAUT, AAAGUT, AAAGCUT, AAAGGCUT\}.$$

The problem we are going to study in this paper is:
given a text (string) T and a secondary expression P, find all occurrences of substrings of T that are in $L(P)$.

2.2 Finite Automata for Secondary Expressions

In order to design algorithms, it is convenient to represent a secondary expression as a nondeterministic finite automaton (NFA).

The construction of a NFA for a secondary expression S is based on the NFA of the network expression of S. There exist several techniques to build an NFA from a network expression (**?**). Here we adopt the classical Thompson construction as follows; (a) the NFA for an expression a for $a \in \Sigma$ is simply a state labelled with a; (See Figure 2 (a).) (b) the NFA for the expression RS, denoted as F_{RS}, is obtained by linking two NFA's F_R and F_S; (See Figure 1 (b).) (c) the NFA for the expression $R|S$, denoted as $F_{R|S}$, is based on F_R and F_S and is shown in Figure 1 (c). The automaton of every expression E has one source state and one sink state, denoted as θ_E and ϕ_E, respectively. We also use θ and ϕ as the source and sink of the whole NFA. The automata thus constructed have the following property:

1. every state has an in-degree and out-degree of 2 or less;
2. the number of states in F is linear in terms of the secondary expression size.

For a state s, let $s.out = \{v | (s, v) \text{ is an edge}\}$, and $s.in = \{v | (v, s) \text{ is an edge}\}$. Every state s is labelled with a symbol in $\Sigma \cup \{\varepsilon\}$.

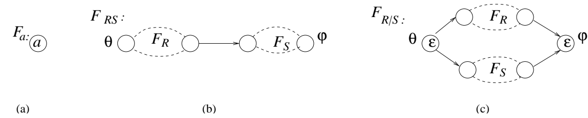

Figure 1: Constructing an NFA for network expression. (a) the NFA for an expression a for $a \in \Sigma$ is simply a state labelled with a. (b) the NFA for the expression RS denoted as F_{RS}, is obtained by linking two NFA F_R and F_S. (c) the NFA for the expression $R|S$ denoted as $F_{R|S}$, is based on F_R and F_S and two new states are added.

Now we extend the NFA of a network expression to the NFA of a secondary expression by including pairing information. Every state s in a NFA belongs to a region, and we say s is an sl (, p, sr) state respectively. According to the above construction procedure, the two subgraphs of NFA corresponding to a pair of complement regions have the same underlying undirected graph. We say s is the **complement state** of t, denoted $s = t.comp$, if s and t are in two complement regions, and correspond to the same vertex in the underlying undirected graph.

In traditional NFAs, a word w is accepted by an NFA F if there exists a path between θ and ϕ such that the sequence spelled by concatenating all labels in the path equals to w. For the NFA of a secondary expression, we have additional requirement that two complement states are either both in the path, or both not.

An order among states is defined for the description of algorithms.

Definition 1 *An* **F-order** $(\prec_F)$ *on states is a total order defined recursively by following rules: (1) if* $E = E_1E_2$*, then for any state* s *in* E_1*, and any state state* t *in* E_2*,* $s \prec_F t$*; (2) if* $E = E_1|E_2$*, if* E *is an sl or p expression, then for any* s *in* E_1*, and any* t *in* E_2*,* $s \prec_F t$*; if* E *is an sr expression, then* $t \prec_F s$*; (3) For any expression* E*, for any* s *in* E*,* $\theta_E \preceq_F s \preceq_F \phi_E$*.*

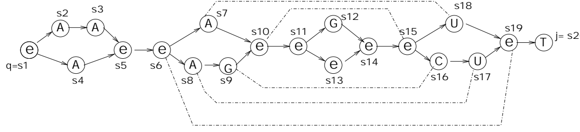

Figure 2: NFA for secondary expression $S_1 = (E_1, p)(E_2, sl)(E_3, p)(E_2', sr)(E_4, p)$, where $E_1 = AA|A, E_2 = A|AG, E_3 = G|\varepsilon, E_4 = T$.

For example, Figure 2 is the NFA for secondary expression S_1, and the states are labelled by their F-order. A pair of complement states are linked with a dotted line.

Denote $s.pre$ the largest state t satisfying $t \prec_F s$, and $s.succ$ the smallest state t satisfying $t \succ_F s$. It is easy to verify that F-order has following property:

1. If s is the complement state of t, and both s and $s.pre$ are sl states, then $s.pre$ is the complement state of $t.succ$;
2. If $s \neq \varepsilon$, then both $s.in$ and $s.out$ contain a unique state.

3 Dynamic Programming Algorithm

We first describe the algorithm for the hairpin expression. After that, we extend the algorithm to work for any secondary expression.

3.1 Matching a Hairpin Expression

Recall that a hairpin expression models a stem-loop structure, so there is a p region at the middle of the expression representing the loop such that all sl expressions are to its left and all sr expressions are to its right. Such a p region is called the *middle* p region. The *middle state ms* is defined as the source state of the middle p region, and we assume that it is labelled with ε. (If the source state is not ε, we can insert an ε-state without changing the language of the NFA.)

We start with a middle state ms. The main idea of this algorithm is to decide whether a substring of T can be derived from a subautomaton of F with a dynamic programming approach.

Definition 2 *The* **SubAutomaton** *of an NFA F between state s and t ($s \preceq_F t$), denoted $F(s,t)$, is the subgraph of F containing all paths from s to t.*

Given a text T and a secondary expression S ($|T| = n$, $|S| = m$), we first construct the NFA F of S. Let s and t be two states in F, where $s \preceq_F t$. Consider a substring $T_{i+1} \dots T_j$. Let $B(i,j,s,t)$ be *true* if $T_{i+1} \dots T_j$ can be accepted by $F(s,t)$, and $false$ otherwise. $B(i,j,\theta,\phi) = true$ means that the substring $T_{i+1} \dots T_j$ is accepted by the automaton F. Therefore, our goal is to decide the values of $B(i,j,\theta,\phi)$'s, for all i,j.

Since the subautomaton $F(ms,ms)$ only accepts ε, we have the following initial condition.

Lemma 1

$$B(i,j,ms,ms) = \begin{cases} true, \text{ if } i = j \\ false, \text{ if } i < j. \end{cases}$$

We compute $B(i,j,s,t)$'s with the following recurrence equations:

Lemma 2 $B(i,j,s,t) =$

$$\begin{cases} \bigvee_{t_1 \in t.in} ((B(i,j-1,s,t_1) \wedge t = T_j) \vee (B(i,j,s,t_1) \wedge t = \varepsilon)), \\ \quad \textit{if } s \textit{ and } t \textit{ are } p \textit{ states, and } t.pre \textit{ is a } p \textit{ state} \\ \bigvee_{s_1 \in s.out} ((B(i+1,j,s_1,t) \wedge s = T_{i+1}) \vee (B(i,j,s_1,t) \wedge s = \varepsilon)), \\ \quad \textit{if } s \textit{ and } t \textit{ are } p \textit{ states, and } t.pre \textit{ is not a } p \textit{ state} \\ \bigvee_{s_1 \in s.out,\ t_1 = s_1.comp} ((B(i+1,j-1,s_1,t_1) \wedge s = T_{i+1} \wedge t = T_j) \vee (B(i,j,s_1,t_1) \wedge s = \varepsilon), \\ \quad \textit{if } s \textit{ is an } sl \textit{ state, } t \textit{ is an } sr \textit{ state, and } t = s.comp \end{cases}$$

Proof. (1) Consider the first formula where s, t and $t.pre$ are p states.

If t is labelled with a character $c \neq \varepsilon$, then in order to make $T_{i+1} \dots T_j$ accepted by $F(s,t)$, T_j must be equal to c, and $T_{i+1} \dots T_{j-1}$ must be accepted by the rest of subautomaton $F(s,t_1)$ where t_1 is a state immediately preceding t.

If t is labelled with ε, then t consumes no character. $T_{i+1} \dots T_j$ can be accepted by $F(s,t)$ if and only if $T_{i+1} \dots T_j$ can be accepted by $F(s,t_1)$.

Combining the above two cases, we get the first formula.

(2) If s is labelled with a character $c \neq \varepsilon$, then in order to make $B(i,j,s,t)$ *true*, T_{i+1} must be equal to c, and $T_{i+2} \dots T_j$ must be accepted by the rest of subautomaton $F(s_1,t)$ where s_1 is a state immediately following s.

If s is labelled with ε, then $T_{i+1} \dots T_j$ can be accepted by $F(s,t)$ if and only if $T_{i+1} \dots T_j$ can be accepted by $F(s_1,t)$.

Combining the above two cases, we get the second formula.

(3) When s is an sl state, t is its complement sr state.

If s is labelled with a character $c \neq \varepsilon$, then t is labelled with c', the complement of c. In order to make $B(i,j,s,t)$ *true*, we must have: $T_{i+1} = c$, $T_j = c'$, and $T_{i+2} \dots T_{j-1}$ is accepted by the rest of subautomaton $F(s_1,t_1)$.

If s is labelled with ε, then t is also a ε state. $T_{i+1} \dots T_j$ can be accepted by $F(s,t)$ if and only if $T_{i+1} \dots T_{j-1}$ can be accepted by $F(s_1,t_1)$.

Combining the above two cases, we get the last formula.

For each fixed pair of (i,j), the order to compute $B(i,j,s,t)$ is as follows: At the beginning, both s and t are at ms, the source state of the middle p region. Fix s, move t forward until t is about to enter an sr expression. The definition of a hairpin expression guarantees that s and t are about to enter a pair of complement regions. Now move s backward and t forward simultaneously until they move out of the two pairing regions, keeping them to be complement states. Again, move s backward with t fixed until s is going to enter an sl expression; then move t forward with s fixed until t is going to enter an sr expression; and then move them together when they are in a pair of complement regions. Repeat the procedure until s reaches θ and t reaches ϕ.

For example, consider the secondary expressoin S_1 and its NFA in Figure 2. The order of pairs of states computed is as follows: (s_{11}, s_{11}), (s_{11}, s_{12}), (s_{11}, s_{13}), (s_{11}, s_{14}), (s_{10}, s_{15}), (s_9, s_{16}), (s_8, s_{17}), (s_7, s_{18}), (s_6, s_{19}), (s_5, s_{19}), (s_4, s_{19}), (s_3, s_{19}), (s_2, s_{19}), (s_1, s_{19}), (s_1, s_{20}).

The algorithm is given in Figure 3. The procedure ComB() computes $B(i,j,s,t)$'s as described above. It takes 6 parameters as input: i and j are two indices of the string; s starts from the state $start_s$ and ends at end_s; similarly, $start_t$ and end_t are starting state

```
Input   Text T of length n, NFA F for a hairpin
        expression S of length m, and the middle
        state ms
Output  all (i,j) such that T_{i+1} ... T_j is a matching
        of S.
    for i = 1 to n, k = 0 to m do
        set B(i, i+k, ms, ms) as Lemma2;
    for i = n to 1 do
        for k = 1 to m do
            call comB(i, i+k, ms, ms, θ, φ);
            if B(i, i+k, θ, φ) = true then
                output (i, i+k);

    procedure comB(i, j, start_s, start_t, end_s, end_t)
        s = start_s;
        t = start_t;
        while (s ≻_F end_s or t ≺_F end_t) do
            if s.pre is a p state then s = s.pre; else
            if t.succ is a p state then t = t.succ; else
            s = s.pre, t = t.succ;
            compute B(i, j, s, t) as Lemma3;
```

Figure 3: Algorithm 1: matching a Hairpin Expression

and ending state of t, respectively. The main procedure first initializes $B(i, j, ms, ms)$ as Lemma 1, and then calls ComB() with s and t both starting from ms and ending at θ and ϕ, respectively. Note that k is iterated from 1 to m, because we need to consider only the substrings of lengths at most m. (m is the size of the input secondary expression).

Now we analyze the complexity of Algorithm 1. According to the property of NFA, for any state s, $|s.out| \leq 2$, and $|s.in| \leq 2$. Thus, computing a $B(i, j, s, t)$ as in Lemma 2 takes constant time. In each iteration of the while loop, s moves backward, or t moves forward, or both. Thus, for a fixed (i, k), the execution time of while loop is no more than the number of states in F, which is $O(m)$. Since there are $n \times m$ (i, k)'s, the total time required is $O(m^2 n)$.

3.2 Matching a Secondary Expression

Now we extend the algorithm to solve the general case of a secondary expression. We solve the problem based on the recurrence definition of secondary expressions.

Case (R1): $S = (E_1, p)$. In this case, S is a hairpin expression and thus can be solved with Algorithm 1.

Case (R2): $S = (E_1, p)(E_2, sl)S'(E_2', sr)(E_3, p)$, where S' is a secondary expression. Let θ_S and ϕ_S be the source and sink states of region S. Suppose that S' has been matched against T. After computing $B(i, j, \theta_{S'}, \phi_{S'})$s, $B(i, j, \theta_S, \phi_S)$ can be computed by calling procedure $comB(i, j, \theta_{S'}, \phi_{S'}, \theta_S, \phi_S)$.

Case 3: $S = S_1 S_2$. In this case, we guess a breaking position k in the string. $T_{i+1} \dots T_j$ is a match of S, if and only if $T_{i+1} \dots T_k$ can be derived from S_1, and $T_{k+1} \dots T_j$ can be derived from S_2. S_1 and S_2 are matched against T first. After that, we compute $B(i, j, \theta_S, \phi_S)$ as follows:

$$B(i, j, \theta_S, \phi_S) = \bigvee_{i \leq k \leq j} (B(i, k, \theta_{S_1}, \phi_{S_1}) \wedge B(k, j, \theta_{S_2}, \phi_{S_2})),$$

where $B(i, j, \theta_{S_1}, \phi_{S_1})$ and $B(i, j, \theta_{S_2}, \phi_{S_2})$ are known.

The algorithm is given in Figure 4. ComputeS() is a recurrence procedure that takes a secondary expression $SecExp$ as input and matches $SecExp$ against T.

```
Input   Text T of length n, NFA F for a secondary
        expression S of length m
Output  all (i,j) such that T_{i+1} ... T_j is a matching
        of S.

    ComputeS(S);
    for i = 1 to n, k = 0 to m do
        if B(i, i+k, θ, φ) = true then
            output (i, i+k);

    procedure computeS(SecExp)
        if SecExp = (E, p) then
            Compute B(i, j, θ_SecExp, φ_SecExp)
                as Algorithm 1;
        if SecExp = (E1, p)(E2, sl)S'(E2', sr)(E3, p)
            then ComputeS(S');
            for i = n to 1, k = 0 to |SecExp| do
                comB(i, i+k, θ_S', φ_S', θ_SecExp, φ_SecExp);
        if SecExp = S1 S2 then
            ComputeS(S1);
            ComputeS(S2);
            for i = 1 to n, j = i to i + |SecExp| do
                for k = 0 to |SecExp| do
                    if B(i, k, θ_S1, φ_S1) = true
                        and B(k, j, θ_S2, φ_S2) = true then
                            B(i, j, θ_SecExp, φ_SecExp) = true;
```

Figure 4: Algorithm 2: matching a Secondary Expression

It uses the procedure ComB() in Figure 3 when computing.

Let t_S be the number of times that (R3) is used in S. We prove that

Theorem 3 *The time complexity of Algorithm 2 is $O((t_S + 1)m^2 n)$, where $|T| = n$, $|S| = m$ and t_S is the number of times that (R3) is used in the secondary expression.*

Proof. We prove it by induction.

Case (R1). According to the analysis of Algorithm 1, the time complexity is $O(m^2 n)$. In this case, $t_S = 0$, so the conclusion holds.

Case (R2). $S = (E_1, p)(E_2, sl)S'(E_2', sr)(E_3, p)$. According to the induction, $ComputeS(S')$ takes $O((t_{S'} + 1) * |S'|^2 n)$ time. Procedure $\mathrm{comB}(i, i+k, \theta_{S'}, \phi_{S'}, \theta_S, \phi_S)$ takes time $O(|F(\theta_S, \theta_{S'})| + |F(\phi_{S'}, \phi_S)|) = O(|S| - |S'|) = O(m - |S'|)$. (The argument is similar to Algorithm 1.) In this case, $t_S = t_{S'}$. Therefore, the time to compute S is

$$\begin{aligned} &O((t_{S'} + 1)|S'|^2 n) + nmO(m - |S'|) \\ &= O((t_{S'} + 1)m|S'|n) + nmO(m - |S'|) \\ &= O((t_S + 1)m^2 n). \end{aligned}$$

Case (R3). $S = S_1 S_2$. According to the induction, $ComputeS(S_1)$ and $ComputeS(S_2)$ take time $O((t_{S_1} + 1) * |S_1|^2 n)$ and $O((t_{S_2} + 1) * |S_2|^2 n)$, respectively. In this case, $t_S = t_{S_1} + t_{S_2} + 1$. Therefore, the time to compute S is

$$\begin{aligned} &O((t_{S_1} + 1) * |S_1|^2 n) + O((t_{S_2} + 1) * |S_2|^2 n) + m^2 n \\ &\leq O(t_S * n(|S_1|^2 + |S_2|^2) + m^2 n) \\ &= O((t_S + 1)m^2 n). \end{aligned}$$

4 Matching Algorithm with History Links

In this section, we introduce another algorithm that keeps history links. The worst case time complexity of this algorithm is the same as that of Algorithm 2. However, it works much faster in most cases.

Before describing our algorithm, we give a quick review of the classical regular expression scanning algorithm with NFA (Gusfield 1997, ?). Let $S(i)$ be the set of states s satisfying that some suffix of $T_1 \dots T_i$ can be accepted by subautomaton $F(\theta, s)$. If $\phi \in S(i)$, there is one or more occurrence of P ending at i. $S(i)$ can be calculated from $S(i-1)$ by starting from any state in $S(i-1)$ or the ε-closure of θ, then moving forward by one state labelled $T(i)$ followed by zero or more state labelled ε. The size of $S(i)$ is $O(m)$, so is the time to calculate $S(i)$. We need to compute all $S(i)$ for $1 \leq i \leq n$, so the total time complexity is $O(nm)$.

Contrary to regular expressions, secondary expressions have an additional requirement that the derivation of an *sr* expression is decided by that of its *sl* counterpart. If we know what the corresponding *sl* character is when an sr character is to be matched, the problem is settled. Therefore, we extend the classical regular expression scanning algorithm by keeping *history links* to record the positions in T that match with the complement state of the current state.

History links

For each T_i in the text, $S(i)$ is defined in the same way as for regular expression, i.e., $S(i)$ is the set of states s satisfying that some suffix of $T_1 \dots T_i$ can be accepted by subautomaton $F(\theta, s)$. For each *sr* state s in $S(i)$, we define $S(i).s.Hlink$ to be the set of *valid positions* j, where $T_k \dots T_i$ (for some $k \leq j$) is accepted by subautomaton $F(\theta, s)$ and in this process $F(\theta, s)$ reaches the complement state of s after accepting T_j. For example, consider the NFA for S_1 in Figure 2. Suppose the text is $T = CACAAGCUTU$. $S(6).s_{15}.Hlink = \{5, 6\}$. Note that j may not be unique for some reasons, e.g., k (the starting point of the pattern) is not unique for the text etc. Once we have $S(i).s.Hlink$ for an *sr* state, we can move forward to match T_{i+1} with the NFA for the secondary expression.

In order to correctly compute Hlinks for sr states, we need to record history information for p states and sl states, too. For a p state s, the sl region E *right before* s is the rightmost sl-region such that all states in E are before s, while all states in its complement sr region are after s in the secondary expression. Let H_s be the sink state of the sl region that is right before a p state s. We define $S(i).s.Hlink$, where s is a p state, to be the set of *valid positions* j, where $T_k \dots T_i$ (for some $k \leq j$) is accepted by subautomaton $F(\theta, s)$ and in this process, $F(\theta, s)$ reaches H_s after accepting T_j. This information will be passed to its complement state (the source state of the complement sr region) via those p states in between.

Since we also want to know the starting positions in the text that a secondary expression is matched, if no sl region is right before the p state s, we define $S(i).s.Hlink$ to be the set of k's such that $T_k \dots T_i$ is accepted by subautomaton $F(\theta, s)$.

For a source state s of a sl region, $S(i).s.Hlink$ is defined the same as p states. (See Case 6 below.) For a sl state that is not a source state of a sl region, we do not have to keep the links.

Once we have $S(i).s.Hlink$ for each sr state, we can move forward to match the text. Similar to the classical algorithm for regular expression, $S(i)$ can be computed based on $S(i-1)$ and $S(i-1).s.Hlink$ for each $s \in S(i-1)$ by starting from a state s in $S(i-1)$ or the ε-closure of θ (We always have to consider states in ε-closure of θ since any position in the text could be the starting position of a pattern. $S(0)$ is set to be empty at the beginning), and moving forward to next state t *matching* T_i followed by zero or more ε state(s). If t is a p state or a sl state, state t *matches* T_i if state t is labeled with T_i. If t is a sr state, state t *matches* T_i if state t is labeled with T_i and T_i is complement to T_{j-1} for $j \in S(i-1).s.Hlink$.

Now we focus on how to maintain $S(i)$ and $S(i).s.Hlink$ for each $s \in S(i)$ when scanning the text. Suppose we have scanned the text $T_1 T_2 \dots T_{i-1}$, and $S(i-1)$ and $S(i-1).s.Hlink$ for each $s \in S(i-1)$ have been computed. Let t be a state that is labeled with T_i and $t \in s.out$ for some $s \in S(i-1)$ or the ε-closure of θ. (For a state s in the ε-closure of θ, $s.Hlink = \{i\}$.) Consider the following cases.

Case 1. s and t are both p states: add t to $S(i)$ and set $t.Hlink = s.Hlink$; (Pass s's information to t.)

Case 2. s and t are both *sr* states: We use the following codes.

add t to $S(i)$;
if $s = \varepsilon$ (s is the source of the sr region) **then**
  $S(i).t.Hlink = S(i-1).s.Hlink$;
else
  for each position $j \in S(i-1).s.Hlink$ **do** add $j-1$ to $S(i).t.Hlink$;
for each $j \in S(i).t.Hlink$ **do**
  if T_j is not complement to T_i **then** delete j from $t.Hlink$;
if $t.Hlink$ is empty (after checking) **then** delete t from $S(i)$;

Here we have to check if T_i is the complement character of its sl counterpart. That is why we have to introduce history links.

Case 3. s and t are both *sl* states: add t to $S(i)$. No need to maintain $S(i).t.Hlink$;

Case 4. s is the sink of an *sl* region: add t to $S(i)$ and set $S(i).t.Hlink = \{i-1\}$.

Here we remember the position and thus the character of the end of a sl region. This information will be passed to the source state of the corresponding sr region via p states in between.

Case 5. s is a p state and t is an sr state: we use the following codes.

add t to $S(i)$;
set $S(i).t.Hlink = S(i-1).s.Hlink$;
for each $j \in S(i).t.Hlink$ **do**
  if T_j is not complement to T_i **then** delete j from $t.Hlink$;
if $t.Hlink$ is empty (after checking) **then** delete t from $S(i)$;

In this case, t is the source state of the sr region (that we are going to match with the text) and the information about the sink state of the corresponding sl region is passed. We then use the passed information to check if T_j and T_i are complement characters.

Case 6. s is the sink of an sr region: add t to $S(i)$ and set

$$t.Hlink = \bigcup_{p \in S(i-1).s.Hlink} S(p).(s.comp).Hlink;$$

Here each $p \in S(i-1).s.Hlink$ is the starting position of the matched stem (both sl region and sr region are matched to $T_k \dots\dots T_i$). $s.comp$ is the source state of a sl region, and $S(p).(s.comp).Hlink$ contains the valid positions of the sink state of the sl region right before t, i.e., the sl region has been matched against the text and its corresponding sr region is going to be matched. The following example illustrates this case.

Example 3: $S = (E_1, sl)(E_2, p)(E_3, sl)(E_4, p)(E_3', sr)(E_5, p)(E_1', sr)$, where $E_1 = CG|GG$, $E_2 = \varepsilon$, $E_3 = UCG|UG$, $E_4 = U|\varepsilon$, $E_5 = UU$; $T = GGUCGUGCAUUCC$. Assume s is $\phi_{E_3'}$, t is

Input	Text T of length$n$, NFA F for a secondary expression S
Output	all (i, j) such that $T_i \dots T_j$ is a matching of S.
1	**for** $i = 1$ to n **do**
2	compute $S(i)$;
3	compute $S(i).s.Hlink$ for each $s \in S(i)$;
4	**if** $\phi \in S(i)$ **then for** each j in $\phi.Hlink$ **do** output (j, i);

Figure 5: Algorithm3: Matching of a Secondary Expression with History Links

θ_{E_5}. When matching $T_9 = A$, we reach s, and $S(9).s.Hlink = \{3, 6\}$ (both $UGCA$ and $UCGUGCA$ are matches of $E_2E_3E'_2$). For $T_3 = U$, we have recorded $S(3).\theta_{E_3}.Hlink = \{2\}$, which points to the valid position of ϕ_{E_1}; and for $T_6 = U$, we have recorded $S(6).\theta_{E_3}.Hlink = \{5\}$. Now we come to $T_{10} = U$ and reach t from s, and $S(10).t.Hlink = \bigcup_{p \in S(9).s.Hlink} S(p).(\theta_{E_3}).Hlink = \{2\} \cup \{5\} = \{2, 5\}$. Thus, we get the valid positions of ϕ_{E_1} in $t.Hlink$, and can use them when matching (E'_1, sr).

Case 7. s is a p state and t is an sl state: set $t.Hlink = s.Hlink$. (Pass the s's information to t.)

After t is included in $S(i)$, any state t' in the ε-closure of t is added to $S(i)$. We have to compute $S(i).t'.Hlink$ by modifying $S(i).t.Hlink$ according to the seven cases. The details are left to interested readers.

The algorithm is given in Figure 5. Again, we assume that θ is labelled with ε. If not, we can insert an ε-state as the source state without changing the language of the NFA.

Example 4: Let S_1 in Figure 2 be the secondary expression and $T = CACAAGCUTU$ the text. For each state s in $S(i)$, we use a set to indicate $Hlink$. The computing result is as follows:

$S(1) : -$
$S(2) : (s_2, \{2\}), (s_4, \{2\}), (s_5, \{2\}), (s_6, \{2\})$
$S(3) : -$
$S(4) : (s_2, \{4\}), (s_4, \{4\}), (s_5, \{4\}), (s_6, \{4\})$
$S(5) : (s_2, \{5\}), (s_3, \{4\}), (s_4, \{5\}), (s_5, \{4, 5\}), (s_6, \{4, 5\}), (s_7, -), (s_8, -), (s_{10}, -), (s_{11}, \{5\}), (s_{13}, \{5\}), (s_{14}, \{5\}), (s_{15}, \{5\})$
$S(6) : (s_9, -), (s_{10}, -), (s_{11}, \{6\}), (s_{12}, \{5\}), (s_{13}, \{6\}), (s_{14}, \{5, 6\}), (s_{15}, \{5, 6\})$
$S(7) : (s_{16}, \{6\})$
$S(8) : (s_{17}, \{5\}), (s_{19}, \{4\})$
$S(9) : (s_{20}, \{4\})$ output(4, 9)
$S(10) : -$

Now we analyze the complexity of Algorithm 3.

Theorem 4 *The worst case time complexity of Algorithm 3 is $O(n(sh + lh^2))$, where l is the number of sl regions in S, s is the maximum size of $S(i)$, and h is the maximum size of $S(i).s.Hlink$.*

Proof. When maintaining $S(i).t.Hlink$, Cases 1, 2, 5, and 7 take time $O(|Hlink|) = O(h)$. Cases 3 and 4 take constant time. Therefore, except for case 6, maintaining $S(i).t.Hlink$ takes time at most $O(h)$ for each s. Since there are at most s state in the NFA, we know that for each fixed i, Cases, 1, 2, 3, 4, 5, and 7 takes at most $O(sh)$ time.

Now consider Case 6 . For an sr sink $\phi_{E'}$, the size of its $Hlink$ is bounded by h. Also, for each possible position p of θ_E, the size of its $Hlink$ is bounded by h. Therefore, the union operation to update $S(i).t.Hlink$ takes time $O(h^2)$. For each i, there are l sr sinks, so Case 6 takes time at most $O(lh^2)$.

Combining the two cases, for each i, the execution time is bounded by $O(sh + lh^2)$. Thus the whole algorithm takes time $O(n(sh + lh^2))$.

By the definition of $Hlink$, the size of $Hlink$ of any state is at most the length of a word derived from S, which is $O(m)$. The size of a state set $S(i)$ is also bounded by $O(m)$. Therefore the worst case time complexity of Algorithm 3 is $O(lm^2n)$, which is the same as that of Algorithm 2. However, usually $S(i)$ does not contain all states in the NFA and $S(i)s.Hlink$ does not contain all m possible positions in the text. Thus, Algorithm 3 is much more efficient in practice.

The space complexity is $O(lm^2)$ that is required for keeping all states with their Hlinks in $S(i-1)$ and, for Case 6, Hlinks of sl-sources from $S(i-m)$ to $S(i-1)$.

5 Conclusion

We have presented algorithms for searching exact matches of secondary structure patterns in strings. To our knowledge, these are the first polynomial algorithms on this problem.

In biological applications, it is extremely useful to include sets of characters in expression, e.g. R for $A|G$ (purine), N for $A|C|G|U$. It is straightforward to extend these algorithms to allow such symbols in expressions. We only need to modify the NFA to allow multiple characters in one state. We can also consider more powerful expressions, for example, expressions that can model higher order structures like pseudoknots.

Approximate matching is also useful in biological study and might be a direction to pursue.

Acknowledgements The work is fully supported by A grant from the Research Grants Council of the Hong Kong Special Administrative Region, China [Project No. CityU 1087/00E].

References

Baeza -Yates, R. (1996), 'A unified view to pattern matching algorithms', *SOFSEM96*, pp. 1-15.

Baeza-Yates, R. & Gonnet, F. (1992), 'A new approach to text searching', *Communication of ACM*, **35**, pp. 74-82

Billoud, B., Kontic, M. & Viari, A., (1996), 'Palingol: a declarative programming language to describe nucleic acids' secondary structures and to scan sequence databases', *Nucleic Acids Res.*, **24**, pp. 1395-1403.

hastain, M. & Tinoco, I. (1991), 'Structural elements in RNA', *Prog. Nucleic Acids Res.*, **41**, pp. 131-177

Earley, J., (1970), 'An efficient context-free parsing algorithm', *Communication of ACM*, **13**, pp. 94-102.

El-Mabrouk, N. & Lisacek, F., (1996), 'Very fast identification of RNA motifs in genomic DNA. Application to tRNA search in the yeast genome', *J. Mol. Biol.*, **264**, pp. 46-55.

El-Mabrouk, N. & Raffinot, M., (2002), 'Approximate matching of secondary structures', *RECOMB02*, 156-164.

Gusfield, D., (1997), 'Algorithms on strings, trees and sequences', Cambridge University Press.

Hopcroft, J. & Ullman, J., (1979), *Introduction to automata theory, languages and computation*, Addison-Wesley.

Lisacek, F., Diaz,Y. & Michel, F., (1994), 'Automatic identification of group I introns cores in genomic DNA sequences', *J. Mol. Biol.* **235**, pp. 1206-1217.

Macke, T.J., Ecker, D.J., Gutell, R.R., Gautheret, D., Case, D.A. & Sampath, R., (2001), 'RNAMotif, an RNA secondary structure definition and search algorithm', *Nucleid Acids Research*, **29**, pp. 4724-4735

Myers, E., (1996), 'Approximate Matching of network expressions with spacers', *J. of Computational Biology*, **3**, pp. 33-51.

Myers, E., (1992), 'A four-russians algorithm for regular expression pattern matching', *Journal of ACM*, **39**, pp. 430-448.

Navarro, G., (2001), 'A guided tour to approximate string matching', *ACM Computing surveys*, **33**, pp. 31-88.

Pesole, G., Liuni, S. & D'Souza, M., (2000), 'PatSearch: a pattern matcher software that finds functional elements in nucleotide and protein sequences and assesses their statistical significance', *Bioinformatics*, **16**, pp. 439-450.

Thompson, K., (1968), 'Regular expression search algorithm', *Communications of the ACM*, **11**, 419-422.

Wu, S. & Manber, U., (1992), 'Fast text searching allowing errors', *Communication of ACM*, **35**, pp. 83-91.