The Forward Stem Matrix

An Efficient Data Structure for Finding Hairpins in RNA Secondary Structures

Richard Beal
West Virginia University
Morgantown, WV 26506
r.beal@acm.org

Donald Adjeroh
West Virginia University
Morgantown, WV 26506
don@csee.wvu.edu

Ahmed Abbasi
University of Virginia
Charlottesville, VA 22903
abbasi@comm.virginia.edu

ABSTRACT

With the rapid growth in available genomic data, robust and efficient methods for identifying RNA secondary structure elements, such as hairpins, have become a significant challenge in computational biology, with potential applications in prediction of RNA secondary and tertiary structures, functional classification of RNA structures, micro RNA target prediction, and discovery of RNA structure motifs. In this work, we propose the Forward Stem Matrix (FSM), a data structure to efficiently represent all k-length stem options, for $k \in K$, within an n-length RNA sequence T. We show that the FSM structure is of size $O(n|K|)$ and still permits efficient access to stems. In this paper, we provide a linear $O(n|K|)$ construction for the FSM using suffix arrays and data structures related to the Longest Previous Factor (LPF), namely, the Furthest Previous Non-Overlapping Factor ($FPnF$) and Furthest Previous Factor (FPF) arrays. We also provide new constructions for the $FPnF$ and FPF via a novel application of parameterized string (p-string) theory and suffix trees. As an application of the FSM, we show how to efficiently find all hairpin structures in an RNA sequence. Experimental results show the practical performance of the proposed data structures.

Categories and Subject Descriptors

F.2.2 [**Analysis of Algorithms and Problem Complexity**]: Nonnumerical Algorithms and Problems—*Pattern matching*; J.3 [**Computer Applications**]: Life and Medical Sciences—*Biology and genetics*; E.0 [**Data**]: General

General Terms

Algorithms, Theory

Keywords

Forward Stem Matrix, FSM, RNA secondary structures, hairpins, Furthest Previous Non-Overlapping Factor, FPnF, Furthest Previous Factor, FPF, parameterized pattern matching, p-match, pattern matching

1. INTRODUCTION AND BACKGROUND

RNA secondary structures are sequences composed of nucleotides with different regions of the RNA strand bonding to form structural elements such as stems, loops, bulges, etc. The core challenge of finding RNA secondary structures in a sequence is to integrate pattern matching with structure analysis. A philosophy used to find RNA secondary structures is to first find the stems and then postprocess the resulting structures. The results of such an exercise will have significant implications in various problems in computational biology, such as in prediction of RNA secondary and tertiary structures [7], RNA structure design [17], functional classification of RNA structures [22], micro RNA target prediction [32], and discovery of RNA structural motifs [24], among others. The recent interest in long non-coding RNAs [26] with significantly complicated structures [20] motivates the need to efficiently find RNA secondary structures.

We introduce the Forward Stem Matrix to efficiently store and permit quick access to all k-stem possibilities, for $k \in K$, in an n-length RNA sequence T. Each $FSM[i][j]$ either (a) contains a set of indices for the $K[i]$-length closing-stem halves $\beta = \texttt{reverse}(\texttt{complement}(T[j...j + K[i] - 1]))$ that correspond to the opening-stem half $\alpha = T[j...j + K[i] - 1]$ if $T[j...j + K[i] - 1]$ is the first occurrence of α in T, (b) contains $\emptyset$ if (a) yields no such indices, or (c) contains the negated index of the furthest previous occurrence of α in T, otherwise. In this paper, we construct the FSM data structure and provide an example application in finding RNA hairpin structures.

Other data structures and methods have been proposed to address variations of RNA secondary structure matching. The Structural Suffix Tree (sST) [28] data structure was developed to handle an extension of the exact pattern matching between an RNA pattern P and text T, in that the individual symbols are used to determine *how* the RNA secondary structure may occur. In cases where we want to specify search for an RNA secondary structure, we require pattern matching between a text and an approximate structure query. Along the lines of other suffix structures [18, 1], the Affix Tree [19] and Affix Array [30] (implemented in the structator system [21]) were constructed specifically for matching with RNA and shown to apply to matching structure queries (also handled in [21]). In an analysis of the time and space resources needed to match RNA secondary structures, the use of even the most elegant data structures still shows the true combinatorial nature of the problem [23, 16]. For example, the methods reported in [19, 30] require exponential time with respect to the number of fragments in an RNA structure query to find inexact matches of an input structure query pattern. This motivates the need for more efficient data structures that more naturally integrate pattern matching with RNA structure analysis.

Our construction of the FSM makes use of the Furthest Previous Non-Overlapping Factor ($FPnF_k$) and the Furthest Previous Factor (FPF_k). The $FPnF_k$ preserves, for each k-length substring at index i in the n-length string T, the minimum index in T where $T[i...i+k-1]$ occurred in a non-overlapping fashion, i.e. $T[\min(x)...\min(x)+k-1] = T[i...i+k-1]$ such that $x+k-1 < i$. The Furthest Previous Factor (FPF_k) omits the non-overlapping condition. The $FPnF_k$ and FPF_k arrays are related to the Longest Previous Factor (LPF) [9] array that maintains information on the longest factors in a string, which are useful in data compression, string factorization, etc. [10]. From the LPF, a family of data structures were derived, including the Longest Previous Non-Overlapping Factor ($LPnF$) [12], Longest Previous Reverse Factor ($LPrF$) [11], Longest Previous Non-Overlapping Reverse Factor ($LPnrF$) [11], and *Prior* [14] for traditional strings from the symbol alphabet Σ. This family was extended in [5] to include the Parameterized Longest Previous Factor ($pLPF$) and variations [6] for the parameterized string (p-string) [2] from a constant alphabet Σ and a parameter alphabet Π. In this work, we provide new constructions for the $FPnF_k$ and FPF_k arrays.

Main Contributions: The Forward Stem Matrix (FSM) is proposed to efficiently store stem options in an RNA sequence for quick access. To assist with the FSM construction, we first introduce the Furthest Previous Non-Overlapping Factor ($FPnF$) and the Furthest Previous Factor (FPF) arrays. We then provide a construction of the $FPnF$ and FPF arrays by making an interesting connection with p-string theory. An improved linear time construction of the $FPnF$ and FPF arrays is shown via suffix trees. Next, the aforementioned results are integrated with suffix arrays (SA) to yield a linear time construction of the FSM. Afterwards, we provide an example application of the FSM: to naturally find hairpin structures in an RNA sequence. Our main results are formalized below, where SA_T and LCP_T respectively denote the suffix array and longest common prefix array for sequence T.

Theorem 1. Given an n-length T and a constant k, the $FPnF_k$ array is constructed in $O(n)$ time and $O(n)$ space.

Theorem 2. Given the n-length T and the set of stem lengths K, the Forward Stem Matrix (FSM) is constructed in $O(n|K|)$ time using $O(n)$ extra space.

Theorem 3. Given SA_T, LCP_T, and FSM on T with $K = \{l, l+1, ..., h-1, h\}$, the act of finding all hairpins H (with a known loop sequence L and an unknown stem sequence with length in the range $[l, h]$) in the RNA sequence T can be accomplished in $O(\max\{|L|, \eta_L \log(h-l)\log n\})$ time with η_L as the number of times that L occurs in T.

In all of our construction algorithms, we avoid data structures that tend to require heavy preprocessing in practice, such as the processing required to instantaneously determine the longest prefix common between *any* two suffixes of a string via range minimum query (RMQ) or lowest common ancestor (LCA) computations.

2. PRELIMINARIES

A string on an alphabet Σ is a production $T = T[1]T[2]...T[n]$ from Σ^n with $n = |T|$ the length of T. We will use the following string notations: $T[i]$ refers to the ith symbol of string T, $T[i...j]$ refers to the substring $T[i]T[i+1]...T[j]$, and $T[i...n]$ refers to the ith suffix of T: $T[i]T[i+1]...T[n]$. We append a terminal symbol \$ (\$ $\notin \Sigma$ with \$ $< \sigma$ for $\sigma \in \Sigma$) to the end of a string to clearly differentiate the lexicographical ordering of suffixes. The terminal may be omitted for brevity. The m-length prefix of a suffix is the substring with the first m symbols of the suffix. The equality between strings, say α and β, only considering the first z symbols is denoted by $\alpha =_z \beta$. The notation $S \circ T$ denotes the concatenation between the strings S and T. This notation may be suppressed where apparent. Suffix structures such as the suffix tree (ST) and the suffix array (SA) with the Longest Common Prefix (LCP) [18] are useful for efficient pattern matching [1].

The parameterized pattern match (p-match) [2] is conducted between parameterized strings (p-strings) from the constant alphabet Σ and the parameter alphabet Π. Two equal length p-strings are p-matches iff (1) the constant symbols match exactly and (2) there exists a bijection between the parameter symbols. The p-match is more directly addressed by comparing `prev` encodings.

Definition 1. **([2]) Previous (`prev`) encoding:** For the n-length p-string T, $\texttt{prev}(T)[i]$ is defined for each $1 \le i \le n$ such that (1) $\texttt{prev}(T)[i] = T[i]$ if $T[i] \in (\Sigma \cup \{\$\})$, (2) $\texttt{prev}(T)[i] = 0$ if $T[i] \in \Pi \wedge T[i] \neq T[j]$ for any $1 \le j < i$, and (3) $\texttt{prev}(T)[i] = i - \max\{j | T[i] = T[j], 1 \le j < i\}$ otherwise.

For a p-string T of length n, the $O(n)$ space `prev` encoding requires the construction time of order $O(n\log(|\Sigma| + |\Pi|))$ via a balanced tree data structure. The construction is linear on an indexed alphabet with an $O(|\Pi|)$ mapping structure.

Lemma 1. An n-length p-string T from Σ and Π can be constructed in $O(n\log(|\Sigma| + |\Pi|))$ with $O(|\Sigma| + |\Pi|)$ extra space.

3. FORWARD STEM MATRIX

An RNA secondary structure, composed of the nucleotides adenine (A), cytosine (C), guanine (G), and uracil (U), is formed when nucleotides from different regions bond within a strand of RNA; the Watson-Crick complementary base pairs between the nucleotides are (A,U) and (C,G). Figure 1 shows the basic structural elements [3] that can make up an RNA secondary structure: single strands, stems (or stacks), loops, and bulges. To search for an RNA structure in a text, an approximate search query (shown in Figure 2) is used. These queries are composed of the basic structural elements, each of which is a fragment of the query. For each fragment, we either specify (1) the exact RNA sequence desired, (2) a range $[l, h]$ of the fragment length, or (3) a hybrid of both (1) and (2) with wildcard symbols. A philosophy used for matching structure queries is to begin with identifying the stems and filter the results as appropriate.

A quick look at Figure 1 shows that the stem is a fundamental secondary structure. All of the double stranded RNA structures can basically be viewed in terms of a stem, or a set of stems. A stem or stack (see Figure 1) forms on a single

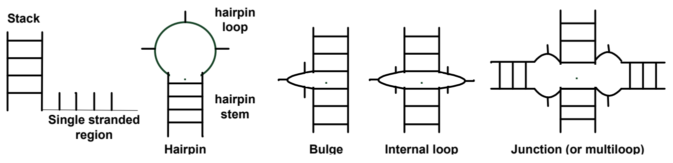

Figure 1: Basic structural elements of an RNA secondary structure.

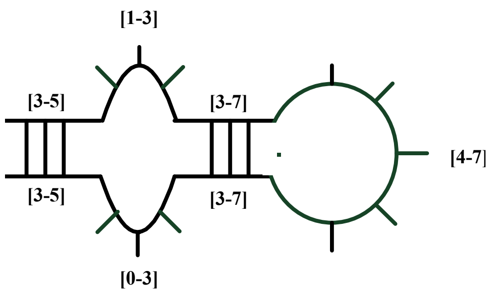

Figure 2: Example definition for an approximate structure query composed of fragments. Numbers in brackets indicate the range for the given fragment length.

strand of RNA, say T, when some k-length substring of RNA at position i in T, i.e. $\alpha = T[i...i+k-1]$, forms a complementary base pairing with a forward, non-overlapping, reversed substring in T, i.e. some $j > i+k-1$ where $\beta = T[j...j+k-1] = \texttt{reverse}(\texttt{complement}(\alpha))$, `reverse` reverses a string, and `complement` forms the base pairings (see Definition 2). For discussion, we call α the opening-stem half and β the closing-stem half.

Definition 2. **`complement` function:** We define the following function to return a complementary RNA sequence: $\texttt{complement}(U[1] \circ U[2] \circ ... \circ U[|U|]) = \overline{U[1]} \circ \overline{U[2]} \circ ... \circ \overline{U[|U|]}$, where $1 \le i \le |U|$ and each $\overline{U[i]} =$'C' if $U[i] =$'G', $\overline{U[i]} =$'G' if $U[i] =$'C', $\overline{U[i]} =$'A' if $U[i] =$'U', and $\overline{U[i]} =$'U' if $U[i] =$'A'.

The RNA sequence $T = ACCCCUGGGGU$, for example, has a stem of length $k = 5$ at indices $i = 1$ and $j = 7$ since $ACCCC = \texttt{reverse}(\texttt{complement}(GGGGU))$. For a structure query, we need to find all stems in T of lengths within the ranges for the considered stem fragments: $k_1, k_2, ..., k_c \in K$. Since a stem is non-overlapping, then $\lfloor \frac{n}{2} \rfloor = \max\{k \mid k \in K\}$. We define the Forward Stem Matrix (FSM) to find all k-length closing-stem halves in T for any opening-stem half in T and every $k \in K$.

Definition 3. **Forward Stem Matrix** (FSM)**:** Consider the n-length RNA sequence T from $\Sigma_{RNA} = \{A, C, G, T\}$ and a set of integers K, with $1 \le k \le \lfloor \frac{n}{2} \rfloor$ for $k \in K$. From any $K[i]$-length opening-stem half in T starting at position j, i.e. $T[j...j+K[i]-1]$, $FSM[i][j]$ provides access to the indices of all forward, non-overlapping, complementary closing-stem halves in T, i.e. $\texttt{reverse}(\texttt{complement}(T[j...j+K[i]-1]) = T[w...w+K[i]-1]$ for $j + K[i] - 1 < w$. Formally, each $FSM[i][j] = y$ is defined below for $1 \le i \le |K|$ and $1 \le j \le n$.

$$y = \begin{cases} \{w \mid \texttt{reverse}(\texttt{complement}(T[j...j+K[i]-1])) = \\ \quad T[w...w+K[i]-1] \ \wedge \ j+K[i]-1 < w\}, \\ \quad \text{if } T[j...j+K[i]-1] \neq T[a...a+K[i]-1] \ \forall \ a < j \\ -\min\{x \mid T[j...j+K[i]-1] = T[x...x+K[i]-1] \\ \quad \wedge \ x < j\}, \text{ otherwise} \end{cases}$$

Consider using this data structure when we want to find all $K[\widehat{i}]$-length closing-stem halves for the opening-stem half $\alpha = T[\widehat{j}...\widehat{j} + K[\widehat{i}] - 1]$. In the case that $FSM[\widehat{i}][\widehat{j}] = \emptyset$, then there are no such closing-stems. In the case that $|FSM[\widehat{i}][\widehat{j}]| = 1$ and $FSM[\widehat{i}][\widehat{j}][1] < 0$, then we know that $Q = FSM[-FSM[\widehat{i}][\widehat{j}][1]]$ preserves all closing-stem halves for an earlier opening-stem; our only task is to report the appropriate closing-stem halves for α at $\widehat{j}$ in T, i.e. $q \in Q$ such that $\widehat{j} + K[\widehat{i}] - 1 < q$. Otherwise, either $|FSM[\widehat{i}][\widehat{j}]| > 1$ or $|FSM[\widehat{i}][\widehat{j}]| = 1$ and $FSM[\widehat{i}][\widehat{j}][1] > 0$, so we have encountered the furthest previous occurrence of the opening-stem half α; here, the closing-stem halves are simply $FSM[\widehat{i}][\widehat{j}]$. By accessing closing-stem halves of opening-stem halves α via the results from the furthest previous occurrence of α, space is saved. The space complexity of the data structure is formalized below.

Lemma 2. Given the n-length string T and the set of considered stem lengths K, the space required by FSM is in $O(n|K|)$.

Proof. Consider one array $FSM[i]$ for some $1 \le i \le |K|$. By Definition 3, we have that each $FSM[i][j]$, $1 \le j \le n$, is either (1) a single negated integral index pointing to the furthest previous occurrence of the opening stem half $T[j...j+K[i]-1]$, or (2) a set of indices denoting the location of all closing-stem halves. Since a unique opening-stem half α has a unique closing-stem half β by Definition 2, then, in the worst case, there are exactly n elements contributed by (2) across all $FSM[i][j]$ for the considered i. In the worst case that all of the n elements appear in one set $FSM[i][j]$, then exactly $n-1$ elements are contributed by (1) for the remaining $FSM[i][h]$ with $1 \le h \le n$ and $h \neq j$. Therefore, at most $2n-1$ elements are needed for each array $FSM[i]$. Since there are $|K|$ such arrays, then $O(n|K|)$ space is required. $\square$

A naïve $O(n^3|K|)$ algorithm to construct the FSM would, for each of the $|K|$ rows in FSM, perform an $O(nk)$ search (with $k \in K$) to find the closing-stem halves for a single opening-stem half, since in the worst case, all of the opening-stem halves are unique in T and $k \in O(n)$. By expediting the pattern matching via the *border* array [29, 4], the algorithm can be improved to $O(n^2|K|)$. To more efficiently compute the FSM, we must compute some new data structures. In the following, we introduce and construct the Furthest Previous Non-Overlapping Factor (FPF) and the Furthest Previous Factor (FPF) arrays. Then, we use these new arrays in our improved construction of the FSM.

3.1 Furthest Previous Non-Overlapping Factor

Motivated by string factor related data structures, we define the Furthest Previous Non-Overlapping Factor ($FPnF_k$) to maintain, for each k-block in T, the index of the first non-overlapping occurrence of that k-block in T.

Definition 4. **Furthest Previous Non-Overlapping Factor ($FPnF$):** For an n-length string T and a chosen integer $1 \leq k \leq \lfloor \frac{n}{2} \rfloor$, the $FPnF_k$ array preserves an index x, for each $1 \leq i \leq n$, that locates the furthest previous non-overlapping occurrence of $T[i...i+k-1]$ in T. More formally, $FPnF_k[i] = \min\{x \mid T[x...x+k-1] = T[i...i+k-1] \ \wedge \ x+k-1 < i\}$. In the case that either $|T[i...i+k-1]| < k$ or no such x exists, define $FPnF_k[i] = 0$.

Omitting the non-overlapping guard yields the Furthest Previous Factor (FPF).

Definition 5. **Furthest Previous Factor (FPF):** For a chosen k and an n-length string T, the FPF_k array preserves an index x, for each $1 \leq i \leq n$, that locates the furthest previous occurrence of $T[i...i+k-1]$ in T. More formally, $FPF_k[i] = \min\{x \mid T[x...x+k-1] = T[i...i+k-1] \wedge x < i\}$. In the case that either $|T[i...i+k-1]| < k$ or no such x exists, define $FPF_k[i] = 0$.

To construct the $FPnF$ and FPF, we can first construct the $FPnF$ and obtain FPF by omitting the non-overlapping restrictions. In a naïve algorithm, we would compute $FPnF$ by considering the $n-k+1$ total k-blocks, which begin at i, and compute the earliest exact match of the k-block in the text via a left-to-right scan. This algorithm would require $O(n^2k)$ time. We can improve this to $O(n^2)$ time using a *border* array for pattern matching [29, 4]. In the following, we improve on this construction time of $FPnF$ and note how to modify the construction to also obtain the FPF.

3.1.1 Construction

To construct the $FPnF_k$, we provide a novel application of p-string theory. Consider a string T from the alphabet A. The $FPnF_k$ preserves, for each ith k-block $T[i...i+k-1]$, the earliest index in T in which the k-block occurred. When $\Sigma = \emptyset$ and $\Pi = A$, the `prev`(T) maintains, for each ith symbol $T[i]$, the distance to the previous $T[i]$ in T. If the `prev` would record the *first* previous occurrence rather than the *immediate* occurrence, it would be a special flavor of $FPnF_k$ where $k = 1$. Our method, shown in Algorithm 1, exploits the `prev` encoding to construct $FPnF_k$. Consider a new array $\widehat{T}$ where we assign integers to the $O(n)$ k-blocks of T such that two integers are equal only when the corresponding k-blocks are equal. When $\Sigma = \emptyset$ and $\Pi = A$, we can use `prev`$(\widehat{T})$ to find not only the previous occurrence of the k-blocks, but via a left-to-right scan, we can successively use the previous occurrences to find the furthest occurrence of each k-block. After we find the first occurrence of the k-block, we only must check the non-overlapping condition. The following lemma formalizes the complexity of the algorithm, which is dominated by the computation `prev`$(\widehat{T})$.

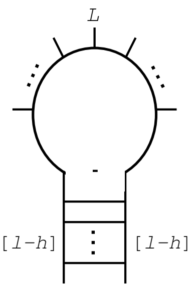

Figure 3: Hairpin query.

Algorithm 1. Constructing $FPnF_k$ array.

```
1  int[n] construct_FPnF_k(int k, char T[n]){
2      int FPnF_k[n], T̂[n], i=1, j, v, z=0
3      int (SA[n], LCP[n])=construct_SA_LCP(T)
4      do{
5          v=z++, T̂[SA[i]]=v
6          while(i<n ∧ LCP[i+1]≥k){
7              T̂[SA[i+1]]=v, i++
8          }i++
9      }while(i≤n)
10     Σ = ∅, Π = A, FPnF_k=prev(T̂)
11     for(i=1 to n){
12         j=FPnF_k[i]
13         if(j>0){
14             FPnF_k[i]=i−j
15             if(FPnF_k[FPnF_k[i]]>0)
16                 FPnF_k[i]=FPnF_k[FPnF_k[i]]
17         }
18     }for(i=1 to n){
19         j=FPnF_k[i]
20         if(j>0 ∧ j+k−1 ≥ i) FPnF_k[i] = 0
21     }return FPnF_k
22 }
```

Lemma 3. Given an n-length T from alphabet A and parameter k, the $FPnF_k$ array is constructed in $O(n \log(k|A|))$ time and $O(\max\{n, k|A|\})$ extra space.

PROOF. The SA and LCP on T require $O(n)$ time and $O(n)$ space to construct. Also, the desired array $FPnF_k$ requires $O(n)$ space. Constructing $\widehat{T}$ in lines 4-9 clearly requires $O(n)$ time since both nested loops are controlled by a variable that is incremented by both loops. The alphabet assignment in line 10 is simply an $O(1)$ operation via pointers. Since $\widehat{T}$ is from an alphabet of size $O(k|A|)$, then `prev` is constructed in $O(n \log(k|A|))$ time with $O(k|A|)$ extra space by Lemma 1. The time required to find the furthest previous indices for each k-block (lines 12-18) and the time required to check the non-overlapping condition (lines 19-21) clearly require a single $O(n)$ scan. Thus, $FPnF_k$ requires $O(n \log(k|A|))$ time and $O(\max\{n, k|A|\})$ extra space. □

In passing, we note that by omitting the last loop in Algorithm 1, we remove the non-overlapping condition and construct the FPF_k array. Even though we will improve the time complexity of the $FPnF$ and FPF constructions, we highlight that the previous constructions are a new application of p-string theory, which traditionally deals with detecting source code redundancy [2, 33] and similar biological sequences [28].

3.1.2 Improved Construction

We now present a more efficient solution to construct the n-length $FPnF_k$ array using the suffix tree ST for the n-length text T. The idea is to introduce computations in addition to a typical preorder traversal of the ST, in which, after we reach a k-block, we collect the suffix indices to report the first index (the furthest previous occurrence) for all such k-blocks in T. The algorithm is as follows:

- Step (1): Initially, set $FPnF_k = \{0, 0, ..., 0\}$.
- Step (2): Now, construct the suffix tree ST for T.
- Step (3a): Perform a preorder traversal on ST (with the root as depth-0). In addition to the traversal, when visiting a new node n_j at depth-k, do:
  - (3a*) Set $S = \emptyset$. For each leaf, i.e. the suffix index say i, reached as descendents from node n_j in the ST, we append this to S via $S = S \cup i$. Upon encountering n_j again in the traversal, the array S will have the list of all the suffixes beginning with the same k-length prefix. By scanning $S = \{s_1, s_2, ..., s_q\}$, we can determine the earliest occurrence of the currently traversed prefix in T at $m_s = \min(S)$. That is, we can determine elements $FPnF_k[s_v]$, for each $1 \leq v \leq q$ with $s_v \neq m_s$, by setting $FPnF[s_v] = m_s$ if $m_s + k - 1 < s_v$. After these $|S| - 1$ elements are populated, the preorder traversal of all children of n_j is complete. Continue to (3b).
- Step (3b): Continue with the preorder traversal. When a different n_j at depth-k is encountered, we execute (3a*).

The complexity of the algorithm is analyzed in the following.

Theorem 1. Given an n-length T and parameter k, the $FPnF_k$ array is constructed in $O(n)$ time and $O(n)$ space.

PROOF. The time needed for initialization in step (1) is in $O(n)$. The ST construction for T in step (2) requires $O(n)$ time and $O(n)$ space (see [1]). In step (3), all of the work is clearly done alongside an $O(n)$ preorder traversal. All that is left to consider is the time required to scan the S structures for the minimum occurrences m_s throughout the algorithm. Since the S is composed of the indices from the $O(n)$ leaves of the ST, then there are exactly n of the $S = S \cup i$ operations and so, each of the leaves is considered exactly once. So, an $O(n)$ scan to find the minimum m_s of each S and populate the other $FPnF_k[s_v]$ is amortized across the entire algorithm. Thus, the $FPnF_k$ construction requires $O(n)$ time and $O(n)$ space. $\square$

By omitting the check of $s_1 + k - 1 < s_v$ in step (3a*), we construct the FPF_k array. This omission does not alter the construction time or space complexity.

Corollary 1. The FPF_k array is constructed in $O(n)$ time and $O(n)$ space.

In passing, we note that, similar to this suffix tree solution, the $FPnF$ and FPF arrays may also be constructed via the suffix array (SA) and the longest common prefix (LCP) array.

3.2 FSM Construction

By looking at the definitions for $FPnF$ and FPF and comparing them with FSM, we see a striking resemblance. In $FSM[i][j]$, we either point to the furthest previous occurrence of a $K[i]$-length opening-stem half, i.e. exactly $FPF_{K[i]}[j]$, or preserve the set of indices of the forward, nonoverlapping, complementary $K[i]$-length closing-stem halves, a variation of $FPnF_{K[i]}[j]$. In the following, we use our $FPnF$ related data structures to construct FSM.

Each $FSM[i][j]$ either (a) contains a set of indices for the $K[i]$-length closing-stem halves $\beta = \texttt{reverse}(\texttt{complement}(T[j...j + K[i] - 1]))$ that correspond to the opening-stem half $\alpha = T[j...j + K[i] - 1]$ if $T[j...j + K[i] - 1]$ is the first occurrence of α in T, (b) contains $\emptyset$ if (a) yields no such indices, or (c) contains the negated index of the furthest previous occurrence of α in T, otherwise. We construct $FSM[i][j]$ individually by row $FSM[i]$ for $K[i]$. Initially $i = 1$.

- Step (1): Define $\overline{A} = \texttt{negate}(A)$ to negate each element of A in $\overline{A}$. Construct SA_T, LCP_T, and $FPF^T_{K[i]} = \texttt{construct\_FPF}_k(K[i], T, SA_T, LCP_T)$. We handle condition (c) by setting $FSM[i] = \texttt{negate}(FPF^T_{K[i]})$, which will populate all $FSM[i][j]$ for an i and every j with the negated index of the furthest previous occurrence of the $K[i]$-length opening-stem half α in T.
- Step (2): Here, we handle the case (a). Let $crT = \texttt{reverse}(\texttt{complement}(T))$ and $\widehat{T} = T \circ \$_1 \circ crT \circ \$_2$. Now, construct $SA_{\widehat{T}}$ and $LCP_{\widehat{T}}$. Let S be a stack with $\texttt{push}$ and $\texttt{pop}$ operations, where initially $S = \emptyset$. Observation (*): By using $\widehat{T}$, it is guaranteed that for each opening-stem half, there will be at least one closing-stem half coming from the latter half of $\widehat{T}$. The trick is to use the $LCP_{\widehat{T}}$ and (*) to group closing-stem halves in T with the guaranteed occurrence(s) O in the latter half of $\widehat{T}$ and use this O to determine where the corresponding opening-stem is in T.
  - Step (2.1) Let $d = -1$. Perform one scan through $LCP_{\widehat{T}}$ to collect all suffixes in T (suffix indices in the range $[1, n]$) that match at least $K[i]$ symbols and $\texttt{push}$ them on S. When we find a suffix in the latter half of $\widehat{T}$ (suffix indices in the range $[n + 2, |\widehat{T}|]$), say at t, that matches at least $K[i]$ symbols, set $d = SA_{\widehat{T}}[t]$. Continue this until we find a suffix that matches less than $K[i]$ symbols to end the grouping. Now, S has the indices of closing-stems in T that should be paired with the furthest previous occurrence of $T[l...l + K[i] - 1]$ with $l = |\widehat{T}| - d - K[i] + 2$. We can find this using $f = l$ if $FPF^T_{K[i]}[l] = 0$ or $f = FPF^T_{K[i]}[l]$ otherwise. Until S is empty, $\texttt{pop}$ all elements with values less than $|T|$ into V. Only a subset $\overline{V}$ of V are actually closing-stem halves for α, i.e. $\overline{V} = \{v \mid v \in V \wedge f + K[i] - 1 < v\}$. Note that we can collect $\overline{V}$ from V with a simple scan to yield an unsorted $\overline{V}$; we will sort this efficiently in step (4). Set $FSM[i][f] = \overline{V}$ to enforce (a) and continue the scan in (2.1) until all elements in $LCP_{\widehat{T}}$ are considered once.
- Step (3): Now, we handle the final case (b). From the previous steps, the elements where $FSM[i][j] = 0$ (for

the considered i and any j) represent those furthest previous opening-stem halves in T where no closing-stem half exists. Thus, we set $FSM[i][j] = \emptyset$ to uphold (b).

- Step (4): Finally, sort the index sets in the current row $FSM[i]$. Let $M = \emptyset$. For $1 \leq z \leq n$, when $|FSM[i][z]| \geq 1 \wedge FSM[i][z][1] \geq 1$, let $M = M \cup (FSM[i][z][y], z)$ for each y with $1 \leq y \leq |FSM[i][z]|$, then let $FSM[i][z] = \emptyset$. Now M has all pairs (c, o), within the row $FSM[i]$, with closing-stem half indices c and the corresponding opening-stem half index o. Perform one radix sort on the c attribute of the pairs in M. Work left-to-right on the sorted M and set $FSM[i][M[w].o] = FSM[i][M[w].o] \cup M[w].c$ with $1 \leq w \leq |M|$. Now, each set in the row $FSM[i]$ has been sorted. Afterwards, each condition (a), (b), and (c) is handled for the row $FSM[i]$. Now increment i and if $i \leq |K|$, consider the next row by (1). Otherwise, FSM is complete.

The time and space complexities of the algorithm are formalized below.

Theorem 2. Given the n-length T and the set of stem lengths K, the Forward Stem Matrix (FSM) is constructed in $O(n|K|)$ time using $O(n)$ extra space.

PROOF. Consider the work performed for each of the $|K|$ rows to construct FSM. Step (1) requires $O(n)$ time, since SA_T and LCP_T are constructed in linear time [18], $FPF^T_{K[i]}$ is constructed in linear time by Corollary 1, and `negate` is an $O(n)$ operation. In step (2), crT is built in linear time due to the $O(n)$ operations `reverse` and `complement`. Also, $SA_{\widehat{T}}$ and $LCP_{\widehat{T}}$ are linear time constructions [18]. In this step, a single $O(n)$ scan is done on the $LCP_{\widehat{T}}$, pushing indices onto S that represent suffixes that match at least $K[i]$ symbols. When $K[i]$ symbols do not match, S is popped onto V and the resulting indices are selected to appear in an element of FSM. Here, exactly n elements are pushed onto S, popped, and selected to appear in $FSM[i][j]$ throughout the entire scan. Thus, this work is amortized across the entire $O(n)$ scan. Finally, a simple $O(n)$ scan is done in step (3). By Lemma 2, step (4) will collect $O(n)$ pairs into M, which are composed of indices from the alphabet $O(n)$. So, a radix sort of M requires $O(n)$ time. Scanning M to repopulate the $FSM[i]$ entries is also an $O(n)$ operation. Since there are $|K|$ rows in FSM, then the algorithm constructs FSM in $O(n|K|)$ time. In terms of extra space (beyond the FSM), the algorithm shares the following structures when computing each row of FSM: the SA, LCP, and FPF arrays based on $|T|, |crT|, |\widehat{T}| \in O(n)$ and the S, V, and $\overline{V}$ based on $|\widehat{T}| \in O(n)$. Thus, $O(n)$ extra space is used. $\square$

3.3 Applications

3.3.1 Finding Hairpins with a Fixed Loop Sequence

A hairpin is made up of a stem and a loop (see Figure 1). The hairpin structure [31] is of great significance in guiding RNA folding, assisting in protein recognition, and even protecting messenger RNA. Consider the hairpin structure query in Figure 3 where the loop pattern is known and the stem sequence is unknown with length in some range. Formally, we address the problem of finding ***all*** hairpins H in the RNA sequence T, where H has a known loop sequence L and an unknown stem sequence with length in the range $[l, h]$ with $h \geq l$. Here, we find all hairpins in an RNA sequence using FSM.

Given SA_T, LCP_T, and FSM with $K = \{l, l+1, ..., h-1, h\}$, the general method is to use SA_T and LCP_T to find all occurrences of the loop L in T and using these occurrences, oracle FSM elements and report matches when there exists an opening-stem half just before the occurrence of L and a corresponding closing-stem half just after the occurrence of L.

- Step (1): Use the SA_T and LCP_T to find the range $[occ_L, occ_H]$ in SA_T where all suffixes have the prefix L. In the case that $occ_H \leq 0$ or $occ_L \leq 0$, there does not exist a match of L and so, no hairpins H with the given loop exist in T. Otherwise, let $i = occ_L$ and continue to step (2).
- Step (2): This step requires that we find some $K[j]$-length stem surrounding the considered occurrence L at $T[SA_T[i]...SA_T[i] + |L| - 1]$. Observation (*): For any stem with length $u > 1$, all of the corresponding length suffixes of the opening-stem half and prefixes of the closing-stem half are also stems. Since for each L occurrence, there can be at most one stem of each length $[l, h]$, then the goal is to find the longest such stem and simply report that also smaller stems exist for this occurrence of L. To do this, we need nested binary searches: the outer binary search varies the stem length, say g, and the inner binary search oracles an element of FSM to say whether or not the stem length g exists around L. Set the boundaries of the outer binary search to the range of $[l, h]$ in K, i.e. $s = 1$ and $e = h - l + 1$, and initially say that no maximum stem exists, i.e. $k = 0$.
  - Step (2.1) If $s \leq 0$ or $e \leq 0$ or $s > e$, then the outer binary search is complete; now, report the results by continuing to step (2.2). Otherwise, set $m = \lfloor \frac{s+e}{2} \rfloor$. The next task is to see if a stem of length $K[m]$ surrounds the L at $T[SA_T[i]...SA_T[i] + |L| - 1]$, i.e. the opening-stem half of length $K[m]$ must exist at $a = SA_T[i] - K[m]$ and the closing-stem must start at $b = SA_T[i] + |L|$. Let $Q = FSM[m][a]$. From Definition 3, we can find a $K[m]$-length closing-stem starting at b for a $K[m]$-length opening-stem starting at a. If $|Q| = 1$ and $Q[1] < 0$, then let $S = FSM[-(Q[1])][a]$; otherwise, let $S = Q$. Binary search for b in S. If b exists, then set $k = K[m]$ if $K[m] > k$ and consider larger possible stem lengths, i.e. set $s = m+1$ and go back to step (2.1). Otherwise, b does not exist, so we consider smaller stem lengths, i.e. $e = m-1$ and go back to step (2.1).
  - Step (2.2): If still $k = 0$, then no stem of any length was found to surround this occurrence of L; proceed to step (3) to try to find a hairpin for the next occurrence of L. If the maximum length $k = l$, then report $(l, SA_T[i])$, signifying that there exists a hairpin H in T where the opening-stem of length l is at $SA_T[i] - l$ in T that is followed by L and the length l closing-stem. Otherwise, the maximum length $k > l$, so,

report $(l, k, SA_T[i])$, signifying that there exists hairpins in T with stem lengths in the range $[l, k]$ with L starting at $SA_T[i]$ in T, i.e. the hairpins with opening-stem length q, for $l \leq q \leq k$, that start at $SA_T[i] - q$ in T are followed by L and a q-length closing-stem.

- Step (3): We increment i and if $i \leq occ_H$, we loop to step (2) to consider matching stems for $T[SA_T[i]...SA_T[i] + |L| - 1]$, the currently considered occurrence of loop L. Otherwise, there are no more occurrences of L to consider and all H in T have already been reported.

The time complexity for the previously detailed algorithm follows.

Theorem 3. Given SA_T, LCP_T, and FSM on T with $K = \{l, l+1, ..., h-1, h\}$, the act of finding all hairpins H (with a known loop sequence L and an unknown stem sequence with length in the range $[l, h]$) in the RNA sequence T can be accomplished in $O(\max\{|L|, \eta_L \log(h-l) \log n\})$ time with η_L as the number of times that L occurs in T.

PROOF. Step (1) executes in $O(|L| + \log n + \eta_L)$ time (see [18]). For step (2), each of the $O(\log(h-l))$ individual iterations from the outer binary search executes the inner binary search in $O(\log(\max\{|FSM[c][d]| \mid 1 \leq c \leq |K| \wedge 1 \leq d \leq n\})) \in O(\log n)$ time (by the proof of Lemma 2). Thus, step (2) executes in $O(\log(h-l) \log n)$ time. Lastly, step (3) loops to step (2) a total of η_L times. Collectively, the time is in $O(|L| + \log n + \eta_L + \eta_L(\log(h-l) \log n)) \in O(\max\{|L|, \eta_L \log(h-l) \log n\})$. $\square$

We acknowledge that by scanning the FSM on the stem lengths K and modifying each index set, i.e. where $|FSM[i][j]| \geq 1 \wedge FSM[i][j][1] \geq 1$ for $1 \leq i \leq |K|$ and $1 \leq j \leq n$, with a two-level hash scheme for perfect hashing [8], we can omit the $\log n$ work of the innermost binary search in the previous hairpin detection algorithm, since a query into each index set will only require $O(1)$ time. Given a satisfactory collection of hash functions, the extra work to build and store hash tables will not change the time or space complexity of the FSM data structure. Thus, the hairpin detection algorithm may be improved to require $O(\max\{|L|, \eta_L \log(h-l)\})$ time. In practice, this improvement uses some extra resources in the FSM construction to offer faster querying to the FSM for applications.

Without the FSM, the act of finding all H in T would also begin by finding all L in T (see step (1)). For each of the η_L occurrences, we begin with matching stem lengths of $1, 2, ..., l-1$ that surround L and for all successive matches up to an h-length stem, we report an occurrence of H in T. This algorithm requires $O(\max\{|L|, \log n, \eta_L h\})$ time. There are two key problems here: (A) for large stem lengths $h \in O(n)$, the algorithm executes in time based on $|T|$ and further, cannot take advantage of cases when both $l, h \in O(n)$ and (B) the problems of (A) are magnified when we extend the result to more complex RNA secondary structures. The FSM data structure expedites the time to find stems so that matching complex RNA secondary structures with many fragments like Figure 2 becomes a problem of finding non-stems (bulges, loops, etc.) that align correctly rather than finding a large number of possible structures and filtering by stems with expensive pattern matching routines.

In passing, we note that the FSM data structure may be augmented with additional data for validating the RNA structures. For example, each opening and closing-stem half in the FSM can include data on the probability that the stem occurs in nature. When building an RNA structure, the probabilities of the stems can be assessed to determine how frequently a structure may exist; those structures with a collection of stems occurring together with some probability p less than a threshold θ may be removed from further consideration. We may further include a separate table of conditional stem probabilities to further validate how frequently a collection of stems in a structure occurs; this addition will help filter infrequently occurring and potentially invalid RNA structures. A further study is required to analyze how the FSM may be augmented to apply more biological approaches to the validation.

3.3.2 Other Applications

Though different variations of hairpin matching problems have been addressed in [19, 30, 21], our algorithm for detecting hairpins with a fixed loop sequence via FSM is very natural. We note that there are a multitude of applications that the FSM can help address. For instance, the FSM can also be used to identify hairpins in an RNA sequence consisting of loops with wildcards, stems with wildcards, etc. In fact, the FSM can be used to find more sophisticated RNA secondary structure patterns (such as Figure 2) in a large RNA database or even assist in predicting the structure of an RNA given a sequence of nucleotides. These patterns can be composed of various RNA structural elements such as stems, loops, bulges, simple hairpins, etc. The patterns can also specify all, some, or none of the nucleotides present in each structural element. To find these RNA secondary structure patterns in a large database, we may take many approaches. One approach is to use the FSM to find a collection of stems matching between the pattern and areas of the database, and postprocess to determine if these stem regions coexist with the other structural elements of the pattern. Let us refer to the stems of an RNA secondary structure pattern and their locations as a *stem signature* of that pattern. Another approach is to use the FSM to first process the pattern and identify the simple hairpins, that is those hairpins that can be extracted from the pattern which look like Figure 3. Let us refer to the simple hairpins of an RNA secondary structure pattern and their locations as the *hairpin signature* of that pattern. Next, we can use the FSM to locate these simple hairpins in the database and finally, postprocess to determine if the locations of the hairpins allow the other RNA structural elements to coexist. We highlight that with the latter approach, the FSM is treated as *an index to hairpins.* That is, the FSM is oracled for natural and efficient access to simple hairpins with a stem and a loop. In a practical sense, using the FSM to locate regions of a signature ultimately evaluates to resolving collisions between RNA secondary structure patterns sharing the same signature. Augmentations to the signature can reduce the probability of collisions and improve use in practice. In a practical setting, using the FSM with an effective signature can be especially useful for dealing with arbitrary pseudoknot structures, which typically require exponential time and super-quadratic space to handle [27, 15, 25].

4. EXPERIMENTS

Table 1: Characteristics of sequences, where $||FSM||_K$ denotes the total number of elements in the FSM data structure on $K = \{1, 2, ..., 10\}$.

T	$\|T\|$	$\mathtt{max}(LCP_T)$	$\mathtt{mean}(LCP_T)$	$\|\|FSM\|\|_K$
EBV	172281	32442	3071.4	2973170
HPOX	212633	71	8.6	3775665
PTM	984602	372	10.4	18746709

To show the practical performance of our algorithms, we implemented the FSM construction and the hairpin detection algorithm each in two ways. First, we implemented the FSM using the proposed algorithm with the SA, LCP, and $FPnF$-related arrays. The $FPnF$-related arrays were constructed using the SA and LCP. The hairpin detection was also implemented as proposed in this paper with the FSM, SA, and LCP data structures. Second, we implemented the FSM construction in a semi-naïve way using a Boyer Moore (BM) pattern matcher [14, 29] to find both the first occurrence of opening-stem halves and all corresponding closing-stem halves. We also implemented the hairpin detection algorithm, which executes like the proposed algorithm except that the loops are found with BM and we replace the outer binary search to find the length of the stem by a sequential search. The programs were written in Java because of the natural way to represent the FSM with a matrix of jagged arrays. For convenience, we also use some basic functions from the Arrays and Collections classes; we could implement these basic utility functions for better performance. The algorithms were executed in a Cygwin environment running on a Dell Inspiron 570 desktop with 3.10 GHz clock speed and 8 GB RAM. Below, we discuss the performance of the algorithms on the following sequences from Rfam [13]: V01555.2 Epstein-Barr Virus (EBV), CR548612.1 Paramecium Tetraurelia Macronuclear (PTM), and DQ792504.1 Horsepox Virus (HPOX). Table 1 shows some characteristics of the aforementioned sequences.

First, we discuss the FSM construction time, which is composed of (1) the time for SA and LCP preprocessing and (2) the time for the FSM algorithm. Figure 4 shows the construction time for the FSM on prefixes of the EBV sequence with $K = \{1, 2, ..., 10\}$ using both the proposed approach (SA) and the semi-naïve approach (BM). It is obvious that over time, the proposed approach, which appears linear, performs better. Further, the EBV sequence has a large amount of repetition according to Table 1, which is quite taxing on the semi-naïve implementation. The improved construction is not impacted by this repetition. The FSM is constructed for the PTM sequence on $K = \{1, 2, ..., 5\}$ and $K = \{1, 2, ..., 10\}$ in Figure 5 and Figure 6, respectively. We see that changing the value of K, i.e. the considered stem lengths, alters the execution time. Recall that the running time for the FSM is directly related to the number of elements in the FSM (see Theorem 2 and Lemma 2). This is clear when observing the behavior between the FSM construction time in Figure 6 and the respective number of FSM elements in Figure 8. Further, we can bound the FSM construction times by the line $2n|K|$, which is the most possible elements in the FSM. In practice, we would achieve this bound in the worst case if, at each prefix of the sequence, the number of elements in the FSM is maximum. In the case of HPOX (see Figure 7), we see that the FSM algorithm actually executes faster than the SA and LCP linear time preprocessing. In our implementation, the provided Java sort methods are used along with some other Java utility functions, for convenience. Performance improvements are possible by implementing these functions along with an integer `radixsort`.

Next, we discuss the time for hairpin detection. In this experiment, we use the FSM structures on $K = \{1, 2, ..., 10\}$ to find all hairpins in the form of Figure 3 with $L = \{A\}^s$, $l = K[1] = 1$, and $h = K[|K|] = 10$. In other words, we search for hairpins where the fixed loop L is a run of s adenine symbols and has a stem length in the range of $[1, 10]$. In the case of the EBV sequence in Figure 9, we detect hairpins with both the proposed approach (SA) and the semi-naïve approach (BM). Notice that for both hairpin detection approaches, the running time is similar. We observe that the running time for hairpin detection is fast for the all of the sequences considered (see Figure 10). In general, we see that hairpin detection is simple given the FSM. Thus, we may suggest that the FSM data structure is *an index to hairpins* in the form of Figure 3. In passing, we note that the way in which we find hairpins is exclusively symbol-based and additional biological validation may be required. Nonetheless, we can quickly provide a multitude of hairpins to a biological filter given the FSM. Also, extensions to the hairpin detection algorithm can identify more sophisticated RNA secondary structures such as Figure 2.

5. CONCLUSIONS

In this paper, we propose the Forward Stem Matrix (FSM), which preserves information regarding all $k \in K$ stem-length options within an n-length RNA sequence. We provide a linear $O(n|K|)$ construction for the FSM via suffix arrays and arrays related to the Longest Previous Factor (LPF), in particular, the Furthest Previous Non-Overlapping Factor ($FPnF$) and Furthest Previous Factor (FPF) arrays. We provide two new constructions for the $FPnF$ and FPF including a novel use of p-string theory and an improved linear solution with suffix trees. As an application of the FSM, we devise a method to find hairpins within an RNA sequence. Experimental results are included to show the empirical performance of the proposed data structure. Using the FSM, other RNA secondary structure problems can also be addressed. For future work, we are interested in extending our data structures to support non-Watson-Crick wobble pairings and applying the data structures to assist in RNA alignment. Also, we wish to investigate how to efficiently filter discovered RNA structures for biological relevance.

6. REFERENCES

[1] Adjeroh, D., Bell, T., Mukherjee, A.: The Burrows-Wheeler Transform: Data Compression, Suffix Arrays, and Pattern Matching. Springer, New York (2008)

[2] Baker, B.: Parameterized pattern matching: Algorithms and applications. Journal of Computer and System Sciences. 52(1), 28-42 (1996)

[3] Batey, R., Rambo, R., Doudna, J.: Tertiary motifs in RNA structure and folding. Angewandte Chemie International Edition. 38, 2326-2343 (1999)

[4] Beal, R., Adjeroh, D.: Border array for structural strings. In: IWOCA'12, pp. 189-205 (2012)

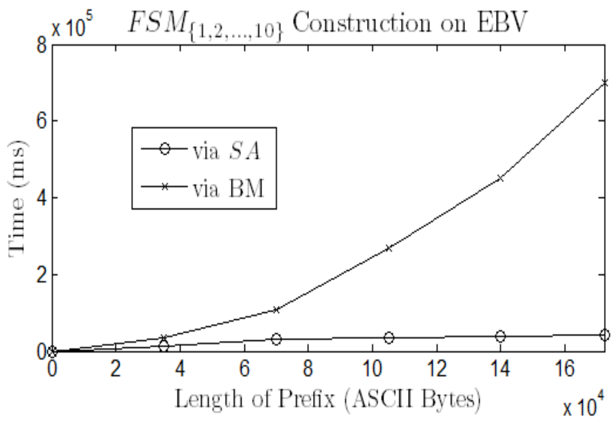

Figure 4: FSM construction on EBV sequence.

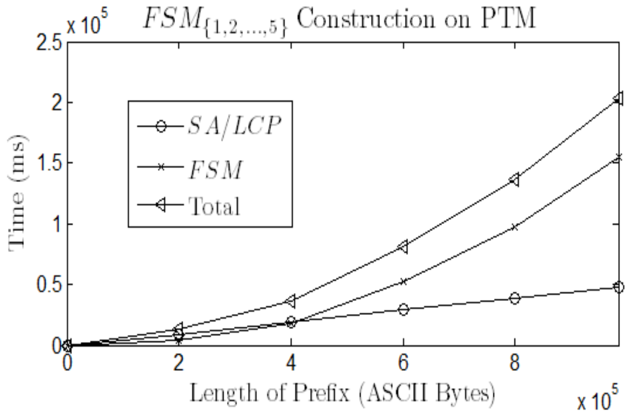

Figure 5: FSM construction on PTM sequence.

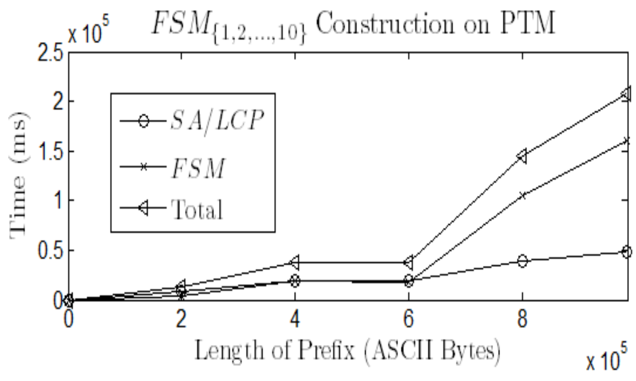

Figure 6: FSM construction on PTM sequence.

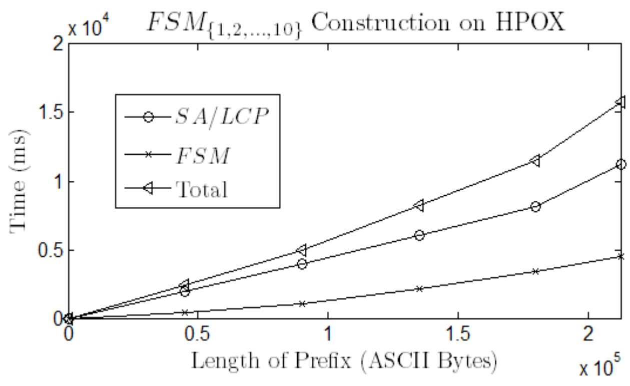

Figure 7: FSM construction on HPOX sequence.

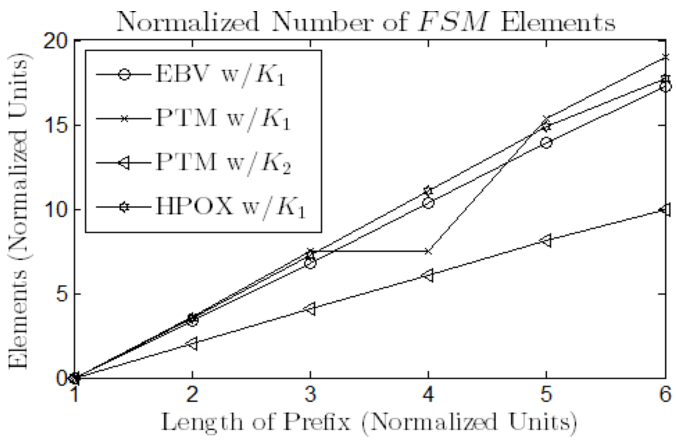

Figure 8: Number of elements in the FSM, where $K_1 = \{1, 2, ..., 10\}$ and $K_2 = \{1, 2, ..., 5\}$. Both the number of elements and the prefix length are normalized by the length of each sequence.

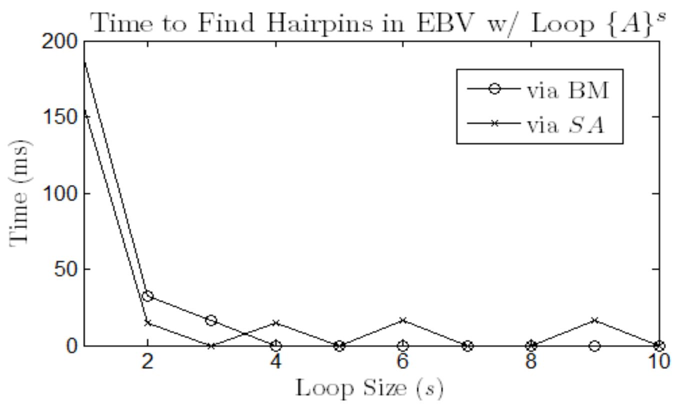

Figure 9: Time to find hairpins in EBV with $L = \{A\}^s$, $l = 1$, and $h = 10$ (see Figure 3).

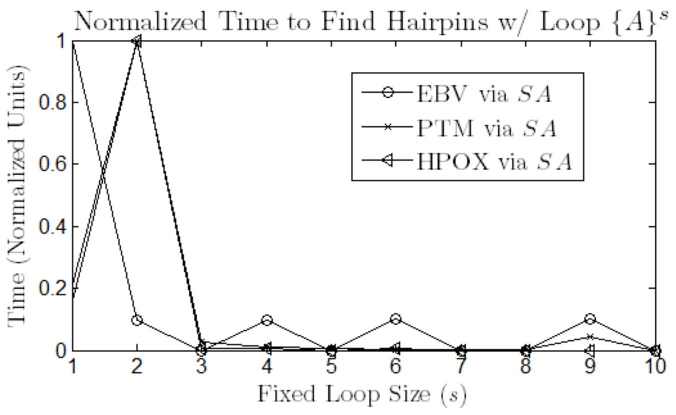

Figure 10: Time to find hairpins with $L = \{A\}^s$, $l = 1$, and $h = 10$ (see Figure 3), normalized by the maximum time to find hairpins within each sequence.

[5] Beal, R., Adjeroh, D.: Parameterized longest previous factor. Theoretical Computer Science. 437, 21-34 (2012)

[6] Beal, R., Adjeroh, D.: Variations of the parameterized longest previous factor. Journal of Discrete Algorithms. 16, 129-150 (2012)

[7] Capriotti, E., Marti-Renom, M.: Computational RNA structure prediction. Current Bioinformatics. 3, 32-45 (2008)

[8] Cormen, T., Leiserson, C., Rivest, R., Stein, C.: Introduction to Algorithms. McGraw-Hill, Boston (2002)

[9] Crochemore, M., Ilie, L.: Computing longest previous factor in linear time and applications. Information Processing Letters. 106(2), 75-80 (2008)

[10] Crochemore, M., Ilie, L., Smyth, W.: A simple algorithm for computing the Lempel Ziv factorization. In: DCC'08, pp. 482-488 (2008)

[11] Crochemore, M., Iliopoulos, C., Kubica, M., Rytter, W., Waleń, T.: Efficient algorithms for three variants of the LPF table. Journal of Discrete Algorithms. 11, 51-61 (2012)

[12] Crochemore, M., Tischler, G.: Computing longest previous non-overlapping factors. Information Processing Letters. 111(6), 291-295 (2011)

[13] Griffiths-Jones, S., Bateman, A., Marshall, M., Khanna, A., Eddy, S.: Rfam: An RNA family database. Nucleic Acids Research. 31(1), 439-441 (2003)

[14] Gusfield, D.: Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology. Cambridge University Press, Cambridge, UK (1997)

[15] Han, B., Dost, B., Bafna, V., Zhang, S.: Structural alignment of pseudoknotted RNA. Journal of Computational Biology. 15(5), 489-504 (2008)

[16] Hofacker, I., Schuster, P., Stadler, P.: Combinatorics of RNA secondary structures. Discrete Applied Mathematics. 88(1-3), 207-237 (1998)

[17] Laing, C., Schlick,T.: Computational approaches to RNA structure prediction, analysis, and design. Current Opinion in Structural Biology. 21(3), 306-318 (2011)

[18] Manber, U., Myers, G.: Suffix arrays: A new method for on-line string searches. SIAM Journal on Computing. 22, 935-948 (1993)

[19] Mauri, G., Pavesi, G.: Algorithms for pattern matching and discovery in RNA secondary structure. Theoretical Computer Science. 335(1), 29-51 (2005)

[20] Novikova, I., Hennelly, S., Tung, C., Sanbonmatsu, K.: Rise of the RNA machines: Exploring the structure of long non-coding RNAs. Journal of Molecular Biology. (In Press)

[21] Meyer, F., Kurtz, S., Backofen, R., Will, S., Beckstette, M.: Structator: Fast index-based search for RNA sequence-structure patterns. BMC Bioinformatics. 12, 214 (2011)

[22] Pedersen, J., Bejerano, G., Siepel, A., Rosenbloom, K., Lindblad-Toh, K., Lander, E., Kent, J., Miller, W., Haussler, D.: Identification and classification of conserved RNA secondary structures in the human genome. PLoS Computational Biology. 2(4), 0251-0262 (2006)

[23] Penner, R., Waterman, M.: Spaces of RNA secondary structures. Advances in Mathematics. 101(1), 31-49 (1993)

[24] Rabani, M., Kertesz, M., Segal, E.: Computational prediction of RNA structural motifs involved in posttranscriptional regulatory processes. PNAS. 105(39), 14885-14890 (2008)

[25] Reidys, C., Huang, F., Andersen, J., Penner, R., Stadler, P., Nebel, M.: Topology and prediction of RNA pseudoknots. Bioinformatics. 27(8), 1076-1085 (2011)

[26] Rinn, J., Chang, H.: Genome regulation by long noncoding RNAs. Annual Review of Biochemistry. 81, 145-166 (2012)

[27] Rivas, E., Eddy, S.: A dynamic programming algorithm for RNA structure prediction including pseudoknots. Journal of Molecular Biology. 285(5), 2053-68 (1999)

[28] Shibuya, T.: Generalization of a suffix tree for RNA structural pattern matching. Algorithmica. 39(1), 1-19 (2004)

[29] Smyth, W.: Computing Patterns in Strings. Pearson, New York (2003)

[30] Strothmann, D.: The affix array data structure and its applications to RNA secondary structure analysis. Theoretical Computer Science. 389(1-2), 278-294 (2007)

[31] Svoboda, P., Di Cara, A.: Hairpin RNA: A secondary structure of primary importance. Cellular and Molecular Life Sciences. 63, 901-918 (2006)

[32] Wang, X.: Computational prediction of microRNA targets. Methods in Molecular Biology. 667, 283-295 (2010)

[33] Zeidman, B. Software v. software. IEEE Spectrum. 47, 32-53 (Oct. 2010)

Acknowledgments

We thank the reviewers for their comments and suggestions that have yielded improvements to the paper. This work is partly supported by NSF grants IIS-1236970 and IIS-1236983.