Bioinformatics, 32(8), 2016, 1151–1157
doi: 10.1093/bioinformatics/btv738
Advance Access Publication Date: 16 December 2015
Original Paper

Sequence analysis

OMPPM: online multiple palindrome pattern matching

Hwee Kim and Yo-Sub Han*

Department of Computer Science, Yonsei University, Seoul 120-749, Republic of Korea

*To whom correspondence should be addressed.

Associate Editor: John Hancock

Received on 23 August 2015; revised on 11 December 2015; accepted on 12 December 2015

Abstract

Motivation: A palindrome is a string that reads the same forward and backward. Finding palindromic substructures is important in DNA, RNA or protein sequence analysis. We say that two strings of the same length are pal-equivalent if, for each possible centre, they have the same length of the maximal palindrome. Given a text T of length n and a pattern P of length m, we study the palindrome pattern matching problem that finds all indices i such that P and $T[i-m+1:i]$ are pal-equivalent.

Results: We first solve the online palindrome pattern matching problem in $O(m^2)$ preprocessing time and $O(mn)$ query time using $O(m^2)$ space. We then extend the problem for multiple patterns and solve the online multiple palindrome pattern matching problem in $O(m_k M)$ preprocessing time and $O(m_k n + c)$ query time using $O(m_k M)$ space, where M is the sum of all pattern lengths, m_k is the longest pattern length and c is the number of pattern occurrences.

Availability and implementation: The source code for all algorithms is freely available at http://toc.yonsei.ac.kr/OMPPM

Contact: kimhwee@cs.yonsei.ac.kr

Supplementary information: Supplementary data are available at *Bioinformatics* online.

1 Introduction

Finding motifs and patterns in bio strings has been one of the most popular topics in both computer science and biology (Adebiyi *et al.*, 2001; Buhler, 2001; Parisi *et al.*, 2003; Prüfer *et al.*, 2008; Rigoutsos and Floratos, 1998). A palindrome is a string that reads the same forward and backward. Namely, a string w is a palindrome if $w = w^R$, where w^R denotes the reversal of w. If a substring of a string is a palindrome, we say that the string has a palindromic substring or palindromic structure. It is important to find palindromes and identify similar palindromic structures in DNA, RNA or protein sequence analysis (Gusfield, 1997). Since palindromic structures in bio data reflect the capability of molecules to fold and form double-stranded stems (Kolpakov and Kucherov, 2009), bio data with similar palindromic structures may have similar secondary structures. Moreover, palindromic sequences are closely associated with DNA breakage during gene conversion (Krawinkel *et al.*, 1986), and palindromic substructures are presented in CRISPR/Cas9 (Kunin *et al.*, 2007), which has been used for gene editing and gene regulation in species (Mali *et al.*, 2013). Therefore, it is useful to identify palindromic substructures and palindromic equivalence efficiently.

We focus on the palindrome pattern matching problem introduced by I *et al.* (2013). Given a text T of length n and a pattern P of length m, the palindrome pattern matching problem is to find all indices i such that P and $T[i-m+1:i]$ have the same set of all centre-distinct maximal palindromes. See Figure 1 for an example.

I *et al.* (2013) presented two algorithms that solve the palindrome pattern matching for an arbitrary size alphabet. We notice that both algorithms by I *et al.* (2013) require a preprocessing step of T. This may slow down the whole process when T is an extremely large text and I/O for T is considerably slow due to the large but slow storages. Moreover, these algorithms might not be applicable if T is a stream data. Many researchers designed online string algorithms to avoid these problems, where each character in T is given online, and we want to report intermediate results without reading

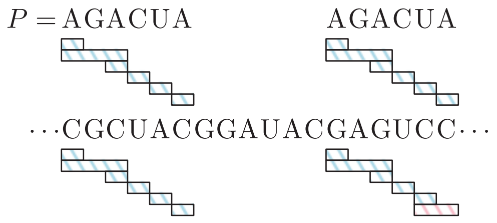

Fig. 1. An example of the palindrome pattern matching. Stripped boxes below a string represent the set of all centre-distinct maximal palindromes with the length at least 1. Note that the pattern on the left is matched, while the pattern on the right is not matched due to the red-stripped box

whole T (Ahmad *et al.*, 2003; Paten *et al.*, 2009). For the palindrome pattern matching problem, we want to report all matching indices i while reading T online. Based on the Knuth–Morris–Pratt algorithm (Knuth *et al.*, 1977), we first build an automaton $\mathcal{A}$ from P and process T using $\mathcal{A}$. For a text T of length n and a pattern P of length m, our algorithm requires $O(m^2)$ preprocessing time and runs in $O(mn)$ query time using $O(m^2)$ space. We, furthermore, tackle the online multiple palindrome pattern matching based on a modification of the Aho–Corasick automaton (Aho and Corasick, 1975). For multiple patterns $P_1, \ldots, P_k$ of length $m_1, \ldots, m_k$, our algorithm requires $O(m_k M)$ preprocessing time and runs in $O(m_k n + c)$ query time using $O(m_k M)$ space, where M is the sum of all pattern lengths, m_k is the longest pattern length and c is the number of pattern occurrences. Note that the second algorithm considers multiple patterns and has the same query time as the first algorithm except the number of pattern occurrences.

2 Methods

2.1 Strings, palindromes and finite automata

A finite-state automaton (FA) $\mathcal{A}$ is specified by $\mathcal{A} = (Q, \Sigma, \delta, s, F)$, where Q is a set of states, Σ is an alphabet, $\delta : Q \times \Sigma \rightarrow Q$ is a transition function, $s \in Q$ is the start state and $F \subseteq Q$ is a set of final states. A string w is accepted by $\mathcal{A}$ if there is a labeled path from s to a state in F such that the path spells out w. For complete background knowledge in automata theory, the reader may refer to textbooks (Hopcroft and Ullman, 1979; Wood, 1986).

For a string w, let w^R denote the reversed string of w. A string w is called a *palindrome* if $w = w^R$. The *radius* of a palindrome w is $\frac{|w|}{2}$. The *centre* of a palindromic substring $w[i:j]$ of a string w is $\frac{i+j}{2}$. A palindromic substring $w[i:j]$ is called the *maximal palindrome* at the centre $\frac{i+j}{2}$ if no other palindromes at the centre $\frac{i+j}{2}$ have a larger radius than $w[i:j]$; in other words, if $w[i-1] \neq w[j+1]$, $i = 1$ or $j = |w|$. Let $Pals(w)$ be the set of all centre-distinct maximal palindromes where each element is encoded by a pair of its centre and radius (I *et al.*, 2010). Namely, given a string w,

$$Pals(w) = \left\{ (c, r) \middle| \begin{array}{l} w[c-r+0.5 : c+r-0.5] \text{ is a maximal} \\ \text{palindrome at centre } c = 1, 1.5, 2, \ldots, n \end{array} \right\}.$$

For example, if $w = abbacabbba$, we have

$$\begin{aligned} Pals(w) = \{&(1, 0.5), (1.5, 0), (2, 0.5), (2.5, 2), (3, 0.5), \\ &(3.5, 0), (4, 0.5), (4.5, 0), (5, 3.5), (5.5, 0), \\ &(6, 0.5), (6.5, 0), (7, 0.5), (7.5, 1), (8, 2.5), \\ &(8.5, 1), (9, 0.5), (9.5, 0), (10, 0.5)\}. \end{aligned}$$

For two strings w and z of the same length, we say that w and z are *pal-equivalent* if $Pals(w) = Pals(z)$. Manacher (1975) proved that for a string w of length m, we can compute $Pals(w)$ in $O(m)$ time. From now on, we assume that the elements of $Pals(w)$ are sorted in increasing order of centres c—the algorithm of Manacher (1975) computes the elements of $Pals(w)$ in this order.

We first tackle the palindrome pattern matching problem in Definition 2.1. Note that while I *et al.* (2013) find start positions of matching occurrences, we search for end positions of matching occurrences.

Definition 2.1 (Palindrome Pattern Matching, Pal-Matching in Short): *Given a text T of length n and a pattern P of length m, compute all positions i such that $Pals(P) = Pals(T[i-m+1:i])$.*

We then define the multiple palindrome pattern matching problems as follows:

Definition 2.2 (Multiple Palindrome Pattern Matching, MPal-Matching in Short): *Given a text T of length n and patterns $P_1, \ldots, P_k$ of length $m_1, \ldots, m_k$, compute all pairs of a position i and a corresponding pattern P_j such that $Pals(P_j) = Pals(T[i-m_j+1:i])$.*

For a pattern matching problem, we can consider an environment where we want to report all matching occurrences at position i after reading each character $T[i]$. This often requires a preprocessing step of the pattern P—we call such a problem an *online pattern matching problem*. We call the time to preprocess P *preprocessing time*, and the time to read T and find all matching occurrences *query time*.

2.2 The algorithm for Pal-matching

We start from designing an algorithm for Pal-Matching in Definition 2.1. The main idea of our algorithm is to design a special automaton simulating the Knuth–Morris–Pratt algorithm (Knuth *et al.*, 1977). Before we design an algorithm, we have the following observation (See Figure. 2 for an illustration): For two strings w, z and an index i, if there exists $(c, r) \in Pals(w)$ such that $c \leq i$ and $c + r - 0.5 \geq i$, then $z[i] = z[2c - i]$. If there is no (c, r) satisfying the condition, then $z[i] \notin \{z[2r-i] | (c, r) \in Pals(w) \text{ and } c + r - 0.5 = i - 1\}$. Note that $z[i]$ is computed based on $z[j]$'s for $j < i$, instead of characters in w. This leads us to define z to be a new sequence of variables, where we can assign characters to variables based on equality and inequality conditions, and the result string is pal-equivalent to w. Based on the observation, we define a *variable pattern* of P as follows:

Definition 2.3: For a pattern P of length m over Σ of size t, a variable pattern P' is defined by an array $\mathbb{A}[m]$ of variables and an array $\mathbb{B}[m]$ of inequality conditions satisfying the following conditions:

1. *$P'[i] = \mathbb{A}[l_i]$ for $1 \leq i, l_i \leq m$.*
2. *If there exists $(c, r) \in Pals(P)$ where $c \leq i$ and $c + r - 0.5 \geq i$, then $l_i = l_{2c-i}$, and thus, $P'[i] = P'[2c - i]$.*
3. *Otherwise, for all $j \in \{2r - i | (c, r) \in Pals(P) \text{ and } c + r - 0.5 = i - 1\}$, $P'[i] \neq P'[j]$. For $P'[i] = \mathbb{A}[l_i]$ and $P'[j] = \mathbb{A}[l_j]$, we use $\mathbb{B}[l_i] = l_j$ and $\mathbb{B}[j] = i$ to denote $P'[i] \neq P'[j]$.*

Namely, if we assign characters to $\mathbb{A}$ based on inequality conditions, then $Pals(P') = Pals(P)$. Initially, we have no variables for constructing P'. The inequality condition of Definition 2.3 implies that for every index i where every maximal palindrome with a centre $c \leq i$ ends before i, we need to introduce a new variable satisfying inequality conditions with respect to the previously-used variables. We construct an array $\mathbb{A}[m]$ of variables. We also construct an array $\mathbb{B}[m]$ that represents the inequality conditions between all pairs of

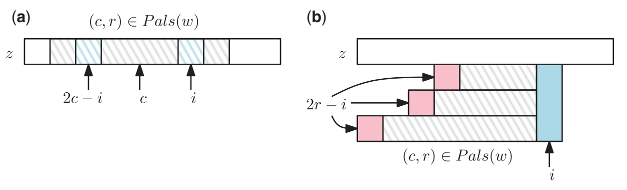

Fig. 2. Two cases in searching pal-equivalent strings. (**a**) There exists $(c, r) \in Pals(w)$ such that $c \leq i$ and $c + r - 0.5 \geq i$. (**b**) There is no (c, r) satisfying the condition. Stripped boxes represent maximal palindromes

$P = \boxed{A}\boxed{G}\boxed{C}\boxed{G}\boxed{T}\boxed{A}$

$P' = \boxed{\mathbb{A}_1}\boxed{\mathbb{A}_2}\boxed{\mathbb{A}_3}\boxed{\mathbb{A}_2}\boxed{\mathbb{A}_4}\boxed{\mathbb{A}_5}$

$\mathbb{A}_4 \neq \mathbb{A}_1, \mathbb{A}_2, \mathbb{A}_3, \mathbb{A}_5 \Rightarrow$

$\mathbb{B}$	
1	2, 3, 4
2	1, 3, 4, 5
3	1, 2, 4
4	1, 2, 3, 5
5	2, 4

Fig. 3. A variable pattern P' and an array $\mathbb{B}$ of inequality conditions for $P = AGCGTA$. Variables $\mathbb{A}[i]$ are written as $\mathbb{A}_i$ in the figure for better readability

variables. Thus, if $j \in \mathbb{B}[i]$, then the condition $\mathbb{A}[i] \neq \mathbb{A}[j]$ holds. Now we construct P' as described in Algorithm 1. Figure 3 shows P' and $\mathbb{B}$ for $P = AGCGTA$.

Based on Definition 2.3, we establish the following result: after running Algorithm 1, if there is a surjection of $\mathbb{A}$ to Σ where $\mathbb{A}[i] \neq \mathbb{A}[j]$ holds for all i, j such that $j \in \mathbb{B}[i]$, then $Pals(P') = Pals(P)$. Moreover, given a string w such that $Pals(w) = Pals(P)$, there exists a surjection of $\mathbb{A}$ to Σ such that $P' = w$.

We analyze the time and space complexity of Algorithm 1. Computing $Pals(P)$ takes $O(m)$ time. Since the **for** loop from line 6 to line 10 takes $O(m)$ time and line 12 also takes $O(m)$ time, the time complexity of the algorithm is $O(m^2)$. For the space complexity, $\mathbb{A}[m]$ and P' requires $O(m)$ space and $\mathbb{B}[m]$ requires $O(m^2)$ space. Therefore, the space complexity is $O(m^2)$.

Once we have P', we can construct a special automaton $\mathcal{A} = (Q, \mathbb{A} \cup \{\sharp\}, \delta, s, F, \Sigma, \mathbb{B}, \delta_f, \mathcal{H})$ that finds all occurrences of P' in T as follows:

- Q is the set of states,
- $\mathbb{A}$ is the array of variables (which is used as an alphabet in $\mathcal{A}$) and $\sharp$ is a wildcard variable,
- $\delta : Q \times \mathbb{A} \rightarrow Q$ is the transition function,
- s is the start state,
- F is the set of final states,
- Σ is the alphabet of the original pattern P,
- $\mathbb{B}$ is the array for inequality conditions of variables,
- $\delta_f : Q \rightarrow Q$ is the failure transition function, and
- $\mathcal{H} : Q \rightarrow 2^{\mathbb{A} \times (\mathbb{A} \cup \{\sharp\})}$ is the set of injective functions for variables.

Note that four parameters——$\Sigma, \mathbb{B}, \delta_f, \mathcal{H}$—are added to the definition of a traditional FA. The automaton $\mathcal{A}$ simulates the Knuth–Morris–Pratt algorithm, using P' instead of P as a pattern. In the Knuth–Morris–Pratt algorithm, when there occurs a mismatch, the algorithm uses the longest suffix of the prefix of T read so far, which is a prefix of P'. The automaton $\mathcal{A}$ simulates the process when a mismatch occurs by δ_f, and additionally, changes surjection of $\mathbb{A}$ to Σ according to $\mathcal{H}$. Algorithm 2 constructs an automaton $\mathcal{A}$ from P and Figure 4 shows an example automaton constructed from $P = AGCGTA$.

We establish the time and space complexity of Algorithm 2 as follows: We can compute $Pals(P)$ in $O(m)$ time and, based on $Pals(P)$, line 11 takes $O(m)$ time. Since other lines in the algorithm

Algorithm 1: ConstructVariablePattern

```
   Input: Pattern $P$ of length $m$ over $\Sigma$ of size $t$
   Output: Variable Pattern $P'$, array $\mathbb{A}[m]$ of variables,
           array $\mathbb{B}[m]$ of inequality conditions
1  construct $\mathbb{A}[m]$, $\mathbb{B}[m]$ and compute $Pals(P)$ // we insert
   $(0.5, 0)$ to $Pals(P)$ for convenience
2  $c \leftarrow 0.5$, $l \leftarrow 0$, $s \leftarrow 0$
3  for $i \leftarrow 1$ to $m$ do
4      if $l \geq i$ then $P'[i] \leftarrow P'[2c-i]$ else
5          $s \leftarrow s + 1$, $P'[i] \leftarrow \mathbb{A}[s]$
6          for each $(c', r') \in Pals(P)$ where $c \leq i$ do
7              if $c' + r' - 0.5 = i - 1$ then
8                  $\mathbb{A}[l] \leftarrow P'[i - 1]$
9                  $\mathbb{A}[l'] \leftarrow P'[c' + r' - 0.5]$
10                 add $l'$ to $\mathbb{B}[l]$, add $l$ to $\mathbb{B}[l']$
11     if $l \leq i$ then
12         find $c'$ such that $(c', r') \in Pals(P)$, $c' - r' + 0.5 \leq i$,
           $i + 1 \leq c' + r' - 0.5$ and $c' + r' - 0.5$ is the smallest.
13         if $c' \leq i + 0.5$ then $l \leftarrow c' + r' - 0.5$, $c \leftarrow c'$ else if
           $l = i - 1$ then $l \leftarrow l + 1$
14 return $P'$, $\mathbb{A}$, $\mathbb{B}$
```

Algorithm 2: ConstructSingleAutomaton

```
   Input: Pattern $P$ of length $m$ over $\Sigma$ of size $t$
   Output: Automaton $\mathcal{A} = (Q, \mathbb{A} \cup \{\#\}, \delta, s, F, \Sigma, \mathbb{B}, \delta_f, \mathcal{H})$
1  ConstructVariablePattern($P$)
2  add $q_0$ to $Q$
3  for $i \leftarrow 1$ to $m + 1$ do
4      if $i \neq m + 1$ then
5          add $q_i$ to $Q$
6          $\delta(q_{i-1}, \mathbb{A}[j]) \leftarrow q_i$ for $P'[i] = \mathbb{A}[j]$
7      if $i = 2$ then
8          $\delta_f(q_1) \leftarrow q_0$
9          add $(\mathbb{A}[1] \leftarrow \#)$ to $\mathcal{H}(q_1)$
10     else if $i > 2$ then
11         find the smallest $i'$ such that
           $Pals(P'[1 : i-i']) = Pals(P'[i' : i-1])$
12         $\delta_f(q_{i-1}) \leftarrow q_{i-i'}$
13         for $l \leftarrow 1$ to $i - i'$ do
14             add $(\mathbb{A}[h] \leftarrow \mathbb{A}[h'])$ to $\mathcal{H}(q_{i-1})$ for $P'[l] = \mathbb{A}[h]$
               and $P'[l+i'-1] = \mathbb{A}[h']$
15         for each $\mathbb{A}[h]$ in $P'[1 : i-1]$ without injective function
           in $\mathcal{H}(q_{i-1})$ do
16             add $(\mathbb{A}[h] \leftarrow \#)$ to $\mathcal{H}(q_{i-1})$ for $\mathbb{A}[h]$
17 return $(Q, \mathbb{A} \cup \{\#\}, \delta, q_0, \{q_m\}, \Sigma, \mathbb{B}, \delta_f, \mathcal{H})$
```

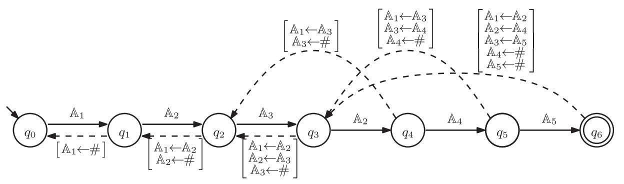
q_0
q_1
q_2
q_3
q_4
q_5
q_6
$\mathbb{A}_1$
$\mathbb{A}_2$
$\mathbb{A}_3$
$\mathbb{A}_2$
$\mathbb{A}_4$
$\mathbb{A}_5$
$[\mathbb{A}_1 \leftarrow \#]$
$[\mathbb{A}_1 \leftarrow \mathbb{A}_2, \mathbb{A}_2 \leftarrow \#]$
$[\mathbb{A}_1 \leftarrow \mathbb{A}_2, \mathbb{A}_2 \leftarrow \mathbb{A}_3, \mathbb{A}_3 \leftarrow \#]$
$[\mathbb{A}_1 \leftarrow \mathbb{A}_3, \mathbb{A}_3 \leftarrow \#]$
$[\mathbb{A}_1 \leftarrow \mathbb{A}_3, \mathbb{A}_3 \leftarrow \mathbb{A}_4, \mathbb{A}_4 \leftarrow \#]$
$[\mathbb{A}_1 \leftarrow \mathbb{A}_2, \mathbb{A}_2 \leftarrow \mathbb{A}_4, \mathbb{A}_3 \leftarrow \mathbb{A}_5, \mathbb{A}_4 \leftarrow \#, \mathbb{A}_5 \leftarrow \#]$

Fig. 4. An automaton $\mathcal{A}$ constructed from $P = AGCGTA$. Variables $\mathbb{A}[i]$ are written as $\mathbb{A}_i$ in the figure for better readability. A dashed transition from a state p is the failure transition $\delta_f(p)$ and the label on the failure transition with square brackets represents the set of injective functions $\mathcal{H}(p)$

Algorithm 3: FindPalindromeMatching

```
  Input: Text $T$ of length $n$ and pattern $P$ of length $m$ over $\Sigma$ of
         size $t$
  Output: Indices $i$ such that $Pals(P) = Pals(T[i{-}m{+}1 : i])$
1 ConstructSingleAutomaton($P$)
2 for $i \leftarrow 1$ to $m$ do $\mathbb{A}[i] \leftarrow \#$ $q_l \leftarrow q_0$ // current state
3 for $i \leftarrow 1$ to $n$ do
4     while one of the following conditions holds for
      $\delta(q_l, \mathbb{A}[j]) = q_{l+1}$
       1.$q_l = q_m$
       2.$\mathbb{A}[j] \neq T[i], \#$
       3.$\mathbb{A}[j] = \#$ and there exists $j' \in \mathbb{B}[j]$ such that $\mathbb{A}[j'] = T[i]$
5     do
6         for each $(\mathbb{A}[h] \leftarrow \mathbb{A}[h']) \in \mathcal{H}(q_l)$ do $\mathbb{A}[h] \leftarrow \mathbb{A}[h']$
          $q_l \leftarrow \delta_f(q_l)$
7     if $\mathbb{A}[j] = \#$ then $\mathbb{A}[j] \leftarrow T[i]$ $q_l \leftarrow q_{l+1}$
8     if $q_l = q_m$ then return $i$
```

except **for** loops require constant time, the total time complexity is $O(m^2)$. For the space complexity, there are $O(m)$ states in $\mathcal{A}$. For each state, there are one out-transition, one outgoing failure transition and $O(m)$ injective functions. Therefore, the space complexity is $O(m^2)$.

Now we present an algorithm that solves Pal-Matching using $\mathcal{A}$. Based on the Knuth–Morris–Pratt algorithm, Algorithm 3 processes T in $\mathcal{A}$ and reports all end-indices of matching occurrences.

We analyze the time and space complexity of Algorithm 3. Checking the condition in line 5 takes $O(m)$ time, and the **for** loop in line 6 takes $O(m)$ time. Note that lines 5–6 runs once for one execution of line 6, where l decreases. For each i, l increases by 1 in line 7. Since $l \geq 0$, the total runtime of the **while** loop from line 5 to line 6 is $O(mn)$. Combined with Algorithm 2 in line 1, the algorithm requires $O(m^2 + mn)$ time and $O(m^2)$ space. Thus, given a text T of length n and a pattern P of length m, we can solve the online palindrome pattern matching problem with $O(m^2)$ preprocessing time and $O(mn)$ query time using $O(m^2)$ space.

2.3 The algorithm for MPal-matching

Now we extend the previous algorithm to solve MPal-Matching. The basic idea of the algorithm is to process multiple patterns at once with a single automaton, based on the idea of the Aho–Corasick automaton (Aho and Corasick 1975). Assume that given patterns $P_1, \ldots, P_k$ of length $m_1, \ldots, m_k$ are sorted by ascending order with respect to the length of the pattern and M is the sum of all pattern lengths. For $P_1, \ldots, P_k$, we first compute variable patterns $P'_1, \ldots, P'_k$, while merging all $\mathbb{B}[m_i]$s to one $\mathbb{B}[k][m_k]$. It is straightforward to show that the process, which we call ConstructMultiVariablePattern, runs in $O(m_k M)$ time using $O(m_k M)$ space.

We define an automaton $\mathcal{B} = (Q, \mathbb{A} \cup \{\sharp\}, \delta, s, F, \Sigma, \mathbb{B}, \delta_f, \mathcal{H}, \delta_p)$. The definition of $\mathcal{B}$ is similar to the definition of $\mathcal{A}$, except for an additional parameter: The pattern suffix transition function $\delta_p : Q \to Q$ contains transitions to find multiple matching occurrences on a single state. The automaton $\mathcal{B}$ simulates the Aho–Corasick algorithm, using $P'_1, \ldots, P'_k$ instead of $P_1, \ldots, P_k$ as patterns. Algorithm 4 constructs $\mathcal{B}$ from $P_1, \ldots, P_k$. We use a supplementary function StateForVP to return the state denoting the end of a given variable pattern. Figure 5 shows an example automaton constructed from $P_1 = AGA, P_2 = ACTG, P_3 = ATAT, P_4 = TCTGC$.

We analyze the time and space complexity of Algorithm 4. We can compute $Pals(P_j)$ in $O(m_j)$ time and, based on $Pals(P_j)$, lines 14

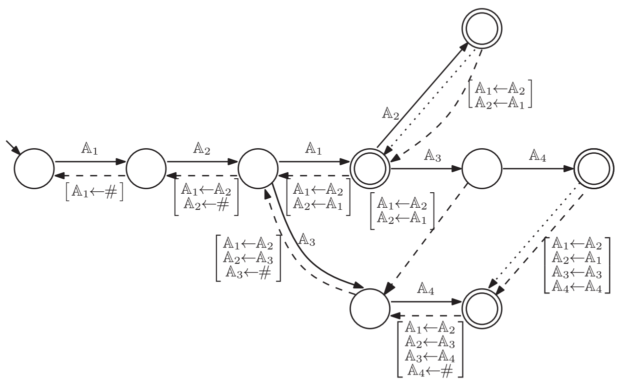
$\mathbb{A}_1$
$\mathbb{A}_2$
$\mathbb{A}_1$
$\mathbb{A}_2$
$\mathbb{A}_3$
$\mathbb{A}_4$
$\mathbb{A}_3$
$\mathbb{A}_4$
$[\mathbb{A}_1 \leftarrow \#]$
$[\mathbb{A}_1 \leftarrow \mathbb{A}_2, \mathbb{A}_2 \leftarrow \#]$
$[\mathbb{A}_1 \leftarrow \mathbb{A}_2, \mathbb{A}_2 \leftarrow \mathbb{A}_1]$
$[\mathbb{A}_1 \leftarrow \mathbb{A}_2, \mathbb{A}_2 \leftarrow \mathbb{A}_1]$
$[\mathbb{A}_1 \leftarrow \mathbb{A}_2, \mathbb{A}_2 \leftarrow \mathbb{A}_1]$
$[\mathbb{A}_1 \leftarrow \mathbb{A}_2, \mathbb{A}_2 \leftarrow \mathbb{A}_3, \mathbb{A}_3 \leftarrow \#]$
$[\mathbb{A}_1 \leftarrow \mathbb{A}_2, \mathbb{A}_2 \leftarrow \mathbb{A}_1, \mathbb{A}_3 \leftarrow \mathbb{A}_3, \mathbb{A}_4 \leftarrow \mathbb{A}_4]$
$[\mathbb{A}_1 \leftarrow \mathbb{A}_2, \mathbb{A}_2 \leftarrow \mathbb{A}_3, \mathbb{A}_3 \leftarrow \mathbb{A}_4, \mathbb{A}_4 \leftarrow \#]$

Fig. 5. An automaton $\mathcal{B}$ constructed from $P_1 = AGA, P_2 = ACTG, P_3 = ATAT, P_4 = TCTGC$. Variables $\mathbb{A}[i]$ are written as $\mathbb{A}_i$ in the figure for better readability. Dashed transitions represent failure transitions and dotted transitions represent pattern suffix transitions

Algorithm 4: ConstructMultiAutomaton

```
   Input: Patterns $P_1, \ldots, P_k$ of length $m_1, \ldots, m_k$ over $\Sigma$ of
          size $t$
   Output: Automaton $\mathcal{B} = (Q, \mathbb{A} \cup \{\#\}, \delta, s, F, \Sigma, \mathbb{B}, \delta_f, \mathcal{H}, \delta_p)$
 1 ConsturctMultiVariablePattern($P_1, \ldots, P_k$)
 2 add $q_\lambda$ to $Q$
 3 $p_1, \ldots, p_k \leftarrow q_\lambda$
 4 for $i \leftarrow 1$ to $m_k + 1$ do
 5     for each $P'_j$ where $i \leq m_j + 1$ do
 6         let $P'_j[i] = \mathbb{A}[l]$ and $p_j = q_s$
 7         find the smallest $i'$ and corresponding $j'$ such that
           $Pals(P'_{j'}[1 : i{-}i']) = Pals(P'_j[i' : i{-}1])$
 8         if $i \neq m_j + 1$ then
 9             $\delta(q_s, \mathbb{A}[l]) \leftarrow q_{s \cdot l}$
10             add $q_{s \cdot l}$ to $Q$
11         if $i = 2$ then
12             $\delta_f(q_s) \leftarrow q_\lambda$
13             add $(\mathbb{A}[1] \leftarrow \#)$ to $\mathcal{H}(q_s)$
14         else if $i > 2$ then
15             $\delta_f(q_s) \leftarrow$StateForVP($P'_{j'}[1 : i{-}i']$)
16             for $g \leftarrow 1$ to $i - i'$ do
17                 add $(\mathbb{A}[h] \leftarrow \mathbb{A}[h'])$ to $\mathcal{H}(q_s)$ for
                   $P'_{j'}[g] = \mathbb{A}[h]$ and $P'_j[g{+}i'{-}1] = \mathbb{A}[h']$
18             for each $\mathbb{A}[h]$ in $P'_j[1 : i{-}1]$ without injective
               function in $\mathcal{H}(q_s)$ do
19                 add $(\mathbb{A}[h] \leftarrow \#)$ to $\mathcal{H}(q_s)$
20         if $i = m_j + 1$ then
21             add $q_s$ to $F$
22             if $i - i' = m_{j'}$ then $\delta_p(p_j) \leftarrow$StateForVP($P'_{j'}$)
23         $p_j \leftarrow q_{s \cdot l}$
24 return $(Q, \mathbb{A} \cup \{\#\}, \delta, q_\lambda, F, \Sigma, \mathbb{B}, \delta_f, \mathcal{H}, \delta_p)$
```

Function StateForVP

```
  Input: Variable Pattern $P'$
  Output: State $q_s$
1 $q_s \leftarrow q_\lambda$
2 for $i \leftarrow 1$ to $|P'|$ do $q_s \leftarrow q_{s \cdot l}$ for $P'[i] = \mathbb{A}[l]$ return $q_s$
```

Algorithm 5: FindMultiPalindromeMatching

```
   Input: Patterns $P_1, \ldots, P_k$ of length $m_1, \ldots, m_k$ over $\Sigma$ of
          size $t$
   Output: Pairs of an index $i$ and a pattern $P_j$ such that
           $Pals(P_j) = Pals(T[i{-}m_j{+}1 : i])$
1  ConstructMultiAutomaton($P_1, \ldots, P_k$)
2  for $i \leftarrow 1$ to $m_k$ do $\mathbb{A}[i] \leftarrow \# \; q_l \leftarrow q_\lambda$      // current state
3  for $i \leftarrow 1$ to $n$ do
4      while one of the following conditions holds for all $\mathbb{A}[j]$
       such that $\delta(q_l, \mathbb{A}[j]) \neq \emptyset$
         1.$q_l$ has no out transition
         2.$\mathbb{A}[j] \neq T[i], \#$
         3.$\mathbb{A}[j] = \#$ and there exists $j' \in \mathbb{B}[g][j]$ such that $\mathbb{A}[j'] = T[i]$
           and $\delta(q_l, \mathbb{A}[j]) =$StateForVP($P'_g[1 : |l|{+}1]$)
5      do
6          for each $(\mathbb{A}[h] \leftarrow \mathbb{A}[h']) \in \mathcal{H}(q_l)$ do $\mathbb{A}[h] \leftarrow \mathbb{A}[h']$
           $q_l \leftarrow \delta_f(q_l)$
7      if $\mathbb{A}[j] = \#$ then $\mathbb{A}[j] \leftarrow T[i]$ $q_l \leftarrow \delta(q_l, \mathbb{A}[j])$
8      if $q_l \in F$ then
9          return $(i, P_{j'})$ where StateForVP($P'_{j'}$) $= q_l$
10         $p_l \leftarrow q_l$
11         while $\delta_p(p_l) \neq \emptyset$ do
12             $p_l \leftarrow \delta_p(p_l)$
13             return $(i, P_{j'})$ where StateForVP($P'_{j'}$) $= p_l$
```

and 21 takes $O(m_k)$ time. Since other lines in the algorithm except **for** loops require constant time, the main loop from lines 4 to 23 takes $O(m_kM)$ time. Therefore, the total time complexity is $O(m_kM)$. For the space complexity, there are $O(M)$ states in $\mathcal{B}$. For each state, there are one out-transition, one outgoing failure transition, at most one outgoing pattern suffix transition and $O(m_k)$ injective functions. Therefore, the space complexity is $O(m_kM)$.

We design Algorithm 5 similar to Algorithm 3 on $\mathcal{B}$ to solve MPal-Matching with an additional process: whenever the current state q_l reaches a final state q_f, return all patterns that are connected by δ_p from q_f. This additional process requires $O(c)$ total runtime, where c is the number of pattern occurrences. Since the size of $\mathcal{H}$ for each state in $\mathbb{B}$ is bounded to m_k, the algorithm requires $O(m_kM + m_kn + c)$ time and $O(m_kM)$ space. Therefore, given a text T of length n and a pattern P of length m, we can solve the online multiple palindrome pattern matching problem with $O(m_kM)$ preprocessing time and $O(m_kn + c)$ query time using $O(m_kM)$ space.

3 Experiments

We design three experiments to estimate the average performance of the algorithms. For Algorithm 3, we first establish two parameters—the length m of the pattern and the length n of the text—and estimated three values—the preprocessing time t_p, the query time t_q, the number s of variables—for random DNA patterns and texts. Second, we calculate the average number of variables for small m by considering all possible patterns of length m. Third, for Algorithm 5, we use real RNA data as a pattern set and measure the preprocessing time t_p and the query time t_q by two parameters—the sum M of all pattern lengths and the longest pattern length m_k.

The details of the experiment are as follows:

1. For the first experiment,
    - The length m of the pattern changes from 10 to 100 by 10, and then from 100 to 1000 by 100. The length n of the text changes from 10 000 to 100 000 by 10 000.
    - For each pair of m and n, we randomly generate a pattern and a text from an alphabet $\{A, G, C, T\}$ 100 times, and calculate the average value of the preprocessing time t_p, the query time t_q and the number of variables s.
2. For the second experiment, we iterate all possible strings for $1 \leq m \leq 10$ and calculate the average of s for each m.
3. For the third experiment,
    - We use 24 RNA secondary structures belonging to distinct RNA families from the Rfam database (Burge *et al.*, 2013) as a superset of a pattern set. The set of RNA secondary structures used is in the supplementary material.
    - We use a RNA-sequence of length 100 000 from the ArrayExpress database (Brazma *et al.*, 2003) as a text. We checked that each pattern in the superset does not appear in the text, which erases the factor c from the runtime.
    - We run 100 iterations. For each iteration, we first choose a pattern p_k, and then select each pattern in the superset with the length less than $|p_k|$ with the probability $\frac{1}{2}$ to form a set of patterns for the iteration. We compute the preprocessing time t_p and the query time t_q.

We obtain the following results from our experiments (note that we have rounded our results to the nearest hundredth.):

- **Preprocessing time of Algorithm 3**: Figure 6 shows the preprocessing time t_p of Algorithm 3 according to the length m of the pattern (the table for the graph is in the supplementary material).
- **Query time of Algorithm 3**: Figure 7 shows the query time t_q of Algorithm 3 according to the length m of the pattern and the length n of the text (tables for graphs are in the supplementary material).
- **Number of variables:** In Algorithm 3, the query time is bounded to $O(ns)$, where s is the number of variables. Figure 8 shows the number of variables s according to the length of the pattern m (The table for the graph is in the supplementary material). The data for $m =$ 1–10 is the average of s for all possible cases, and the data from m $= 10$ to $m = 1000$ is the average for 100 random cases.
- **Pre-processing time of** Algorithm 5: Figure 9 shows the preprocessing time t_p of Algorithm 5, according to the sum of all pattern lengths M and the longest pattern length m_k.
- **Query time of Algorithm 5**: Figure 10 shows the query time t_q of Algorithm 5 according to the longest pattern length m_k and the sum of all pattern lengths M. We observe that t_q is independent

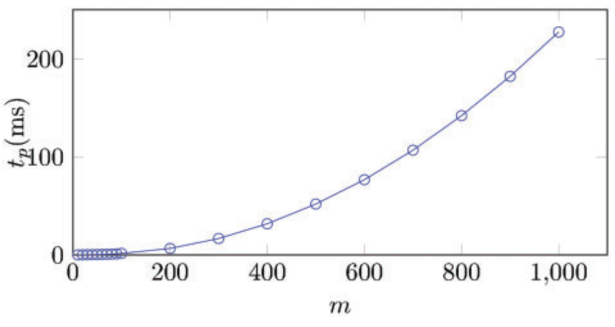

Fig. 6. Preprocessing time graph for Algorithm 3, where $10 \leq m \leq 1000$. m denotes the length of pattern and t_p denotes the preprocessing time. We can observe that t_p follows the quadratic function of m since $t_p = O(m^2)$.

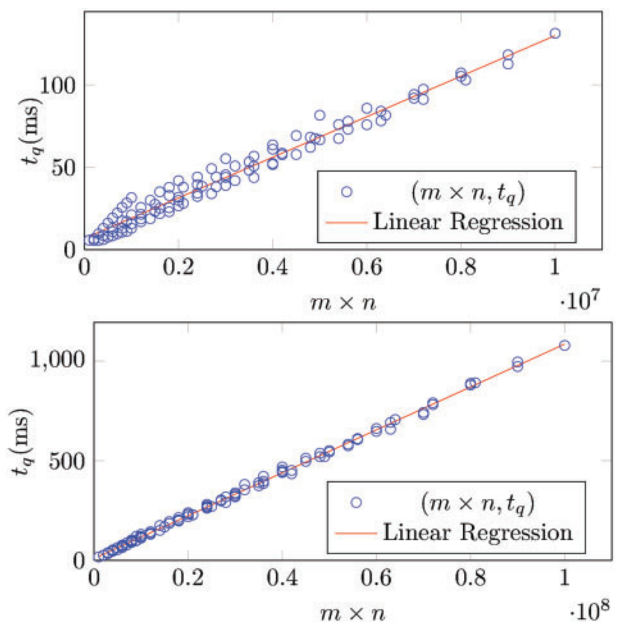

Fig. 7. Query time graph for Algorithm 3. m denotes the length of pattern and t_q denotes the query time. We observe that t_q is proportional to n and m since $t_q = O(nm)$. Note that t_q for $n = 10\,000$ and $m = 100$ is 13.02, whereas t_q for $n = 100\,000$ and $m = 10$ is 31.56. This implies that the increase of m affects t_q less than the increase of n

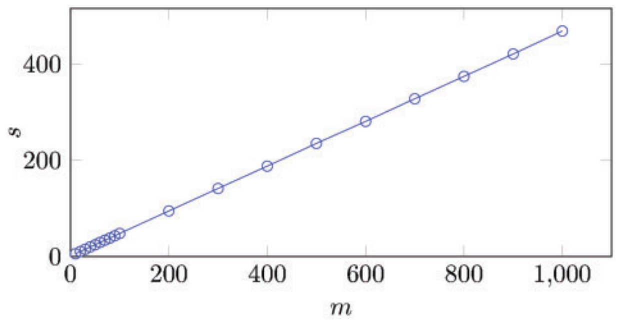

Fig. 8. Number of variable graph, where m denotes the length of the pattern and s denotes the number of variables used. For $m = 1$–10, we observe linear increase of s as m increases. The difference of s between m and $m-1$ tends to decrease as m increases, but the difference rapidly converges to 0.47, and we can easily approximate $s = 0.47m$ (Note that $s = 468.78$ when $m = 1000$.)

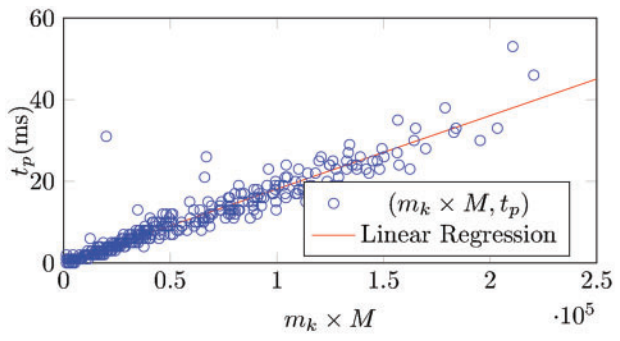

Fig. 9. Preprocessing time graph for Algorithm 5. We observe that t_p is proportional to M and m_k since $t_p = O(m_k M)$

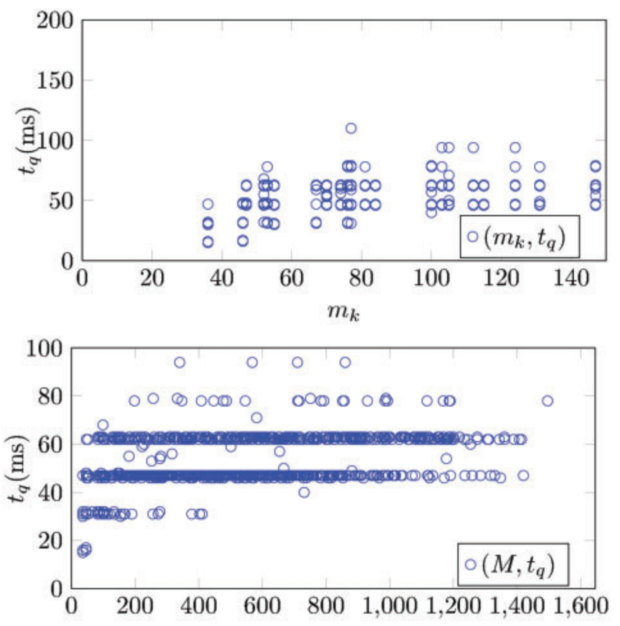

Fig. 10. Query time graph for Algorithm 5, considering m_k and M. m_k denotes the length of the longest pattern, M denotes the sum of the lengths of all patterns and t_q denotes the query time. We observe that t_q is independent from M

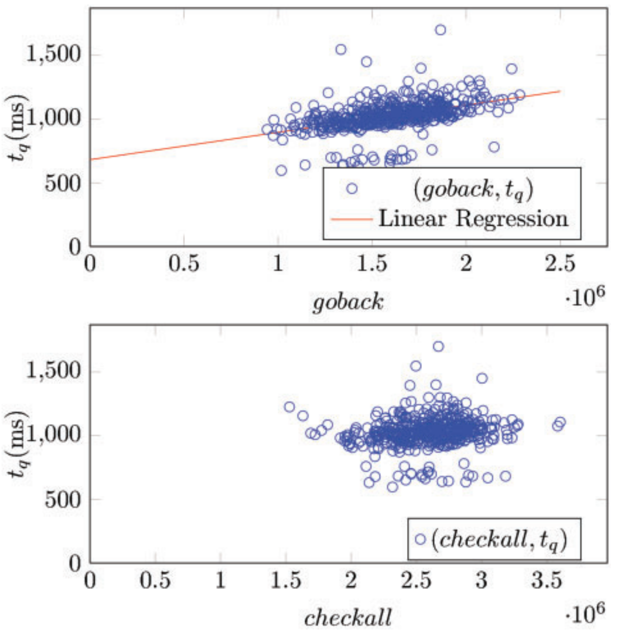

Fig. 11. Query time graph for Algorithm 5, considering *goback* and *checkall*. *goback* denotes the number of changes on the array of variables, *checkall* denotes the number of pattern suffix transitions taken, and t_q denotes the query time. This graph shows that t_q is proportional to *goback*, which is $O(m_k n)$ but the average value is far less than m_{kn} and not proportional to m_{kn}

to M but it is not clear whether or not t_q is proportional to m_k. We design another experiment to determine the factor that affects m_k most.

- T is a randomly generated text of length 100 000. We run 1000 iterations for different sets of patterns.
- For each iteration, we choose m_k between 100 and 200, and generate a set of random patterns, where M is 1000.

- We record the number of changes on the array of variables (which we call *goback*) and the number of pattern suffix transitions taken (which we call *checkall*).

Figure 11 shows the query time t_q of Algorithm 5 according to *goback* and *checkall*. Theoretically, $t_q = O(m_k n + c)$, the upper bound of *goback* is m_{kn} and the upper bound of *checkall* is c. This experiment shows that t_q is proportional to *goback*, which is $O(m_k n)$ but the average value is far less than m_{kn} and not proportional to m_{kn}. This feature makes the algorithm much more efficient than running pattern matching algorithms for individual pattern k times.

4 Conclusions

Palindromic structures are widely studied in string processing and combinatorics and have applications in the analysis of DNA, RNA and protein sequences. For a text T of length n and a pattern P of length m, we have solved the online palindrome pattern matching in $O(m^2)$ preprocessing time and $O(mn)$ query time using $O(m^2)$ space. Then we have extended the problem for multiple patterns $P_1, \ldots, P_k$ and solved the online multiple palindrome pattern matching in $O(m_k M)$ preprocessing time and $O(m_k n)$ query time using $O(m_k M)$ space, where M is the sum of all pattern lengths and m_k is the longest pattern length. Note that the algorithm for the multiple palindrome pattern matching does not increase the query time. We performed experiments to analyze the runtime of the algorithms, and found out that the runtime for the multiple pattern matching is much faster than expected. We believe that the algorithm can be efficiently used to find a structural similarity between multiple bio strings. Since the online multiple palindrome pattern matching is first proposed in the paper, our future work includes reducing time and space requirement of the algorithm. Moreover, we believe that the approach to solve the multiple pattern matching based on the Aho–Corasick automaton can be applied to pattern matching problems considering other structural equivalences.

Acknowledgements

We wish to thank the referees for the careful reading of the paper and many valuable suggestions including relevant references.

Funding

This work was supported by the Basic Science Research Program through National Research Foundation funded by MEST [2015R1D1A1A01060097], Yonsei University Future-leading Research Initiative of 2015 and the National Research Foundation Grant funded by the Korean Government [NRF-2013-Global Ph.D. Fellowship Program to H.K.].

Conflict of Interest: none declared.

References

Adebiyi,E.F. *et al*. (2001) An efficient algorithm for finding short approximate non-tandem repeats. *Bioinformatics*, **17**, S5–S12.

Ahmad,S. *et al*. (2003) RVP-net: online prediction of real valued accessible surface area of proteins from single sequences. *Bioinformatics*, **19**, 1849–1851.

Aho,A.V. and Corasick,M.J. (1975) Efficient string matching: an aid to bibliographic search. *Commun. ACM*, **18**, 333–340.

Brazma,A. *et al*. (2003) ArrayExpress–a public repository for microarray gene expression data at the EBI. *Nucleic Acids Res*., **31**, 68–71.

Buhler,J. (2001) Efficient large-scale sequence comparison by locality-sensitive hashing. *Bioinformatics*, **17**, 419–428.

Burge,S.W. *et al*. (2013) Rfam 11.0: 10 years of RNA families. *Nucleic Acids Res*., **41**, 226–232.

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

Hopcroft,J.E. and Ullman,J.D. (1979). *Introduction to Automata Theory, Languages, and Computation*. Addison–Wesley, Boston, USA.

I,T. *et al*. (2010). Counting and verifying maximal palindromes. In: Proceedings of the 17th International Conference on String Processing and Information Retrieval, pp. 135–146.

I,T. *et al*. (2013). Palindrome pattern matching. *Theor. Comput. Sci*., **483**, 162–170.

Knuth,D.E. *et al*. (1977). Fast pattern matching in strings. *SIAM J. Comput*., **6**, 323–350.

Kolpakov,R. and Kucherov,G. (2009) Searching for gapped palindromes. *Theor. Comput. Sci*., **410**, 5365–5373.

Krawinkel,U. *et al*. (1986) Palindromic sequences are associated with sites of DNA breakage during gene conversion. *Nucleic Acids Res*., **14**, 3871–3882.

Kunin,V. *et al*. (2007) Evolutionary conservation of sequence and secondary structures in CRISPR repeats. *Genome Biol*., **8**, R61.

Mali,P. *et al*. (2013) Cas9 as a versatile tool for engineering biology. *Nat. Methods*, **10**, 957–963.

Manacher,G. (1975) A new linear-time "on-line" algorithm for finding the smallest initial palindrome of a string. *J. ACM*, **22**, 346–351.

Parisi,V. *et al*. (2003) STRING: finding tandem repeats in DNA sequences. *Bioinformatics*, **19**, 1733–1738.

Paten,B. *et al*. (2009) Sequence progressive alignment, a framework for practical large-scale probabilistic consistency alignment. *Bioinformatics*, **25**, 295–301.

Prüfer,K. *et al*. (2008) PatMaN: rapid alignment of short sequences to large databases. *Bioinformatics*, **24**, 1530–1531.

Rigoutsos,I. and Floratos,A. (1998) Combinatorial pattern discovery in biological sequences: the TEIRESIAS algorithm. *Bioinformatics*, **14**, 55–67.

Wood,D. (1986). *Theory of Computation*. Harper & Row, New York City, USA.