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 $O(mkM)$ preprocessing time and $O(mkn + c)$ query time using $O(mkM)$ space, where $M$ is the sum of all pattern lengths, $mk$ 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., 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.
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 $A$ from $P$ and process $T$ using $A$. 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 $O(m(n+c))$ preprocessing time and runs in $O(m(n+c))$ query time using $O(mk)$ 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) $A$ is specified by $A = (Q, \Sigma, \delta, s, F)$, where $Q$ is a set of states, $\Sigma$ is an alphabet, $\delta : Q \times \Sigma \to Q$ 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 $A$ 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 $|w|/2$. The centre of a palindromic substring $w[i : j]$ of a string $w$ is $i + j$. A palindromic substring $w[i : j]$ is called the maximal palindromal at the centre $i + j$ if no other palindromes at the centre $i + j$ 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 $w$,

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

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

$$Pals(w) = \{(1, 0.5), (1.5, 0), (2, 0.5), (2.5, 2), (3, 0.5), (3.5, 0), (4, 0.5), (4.5, 0), (5.3, 5), (5.5, 0), (6, 0.5), (6.5, 0), (7, 0.5), (7.5, 1), (8, 2.5), (8.5, 1), (9, 0.5), (9.5, 0), (10, 0.5)\}.$$
The time complexity of the algorithm is $O(m)$ time and line 12 also 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, $A[m]$ and $P'$ requires $O(m)$ space and $B[m]$ requires $O(m^2)$ space. Therefore, the space complexity is $O(m^2)$.

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

- $Q$ is the set of states,
- $A$ is the array of variables (which is used as an alphabet in $A$) and $\#$ is a wildcard variable,
- $\delta : Q \times A \rightarrow Q$ is the transition function,
- $\epsilon$ is the start state,
- $F$ is the set of final states,
- $\Sigma$ is the alphabet of the original pattern $P$,
- $B$ is the array for inequality conditions of variables,
- $\delta_t : Q \rightarrow 2^{A \times \{\#\}}$ is the set of injective functions for variables.

Note that four parameters——$\Sigma, B, \delta_t, H$——are added to the definition of a traditional FA. The automaton $A$ 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 $A$ simulates the process when a mismatch occurs by $\delta_t$, and additionally, changes surjection of $A$ to $\Sigma$ according to $H$. Algorithm 2 constructs an automaton $A$ 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 use the longest suffix of the prefix of $P$, the time complexity is $O(m^2)$. Moreover, given a string $w$, once we have $Pals(w)$, we can construct a special automaton $A = (Q, A \cup \{\#\}, \delta, s, F, \Sigma, B, \delta_f, H)$ that finds all occurrences of $P$ in $w$. Therefore, the space complexity is $O(m^2)$.

Based on Definition 2.3, we establish the following result: after running Algorithm 1, if there is a surjection of $A$ such that $Pals(w) = Pals(P)$, then $A[k][j] \neq A[k][j]$ holds. Now we construct $P'$ as described in Algorithm 1. Figure 3 shows $P'$ and $B = PALS$. Variables $A[k]$ are written as $A_k$ in the figure for better readability.

Algorithm 1: ConstructVariablePattern

<table>
<thead>
<tr>
<th>Input</th>
<th>Pattern $P$ of length $m$ over $\Sigma$ of size $t$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output</td>
<td>Variable Pattern $P'$, array $A[m]$ of variables, $B[m]$ of inequality conditions</td>
</tr>
</tbody>
</table>

1. Construct $A[m], B[m]$ and compute $Pals(P)$ // we insert (0, 0) to $Pals(P)$ for convenience
2. $c \leftarrow 0.5, l \leftarrow 0, s \leftarrow 0$
3. for $i \leftarrow 1$ to $m$
   4. if $l \geq i$ then $P'[i] \leftarrow P'[2c-i]$ else
   5. $s \leftarrow s + 1,$ $P'[i] \leftarrow A[s]$
   6. for each $(c', r') \in Pals(P)$ where $c \leq i$
      7. if $c' + r' - 0.5 = i - 1$ then
         8. $A_l[i] \leftarrow P'[i - 1]$
         9. $A_l[i] \leftarrow P'[c' + r' - 0.5]$
         10. add $l'$ to $B [l], l$ to $B [l']$
   11. if $l \leq i$
      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 $P', A, B$

Algorithm 2: ConstructSingleAutomaton

<table>
<thead>
<tr>
<th>Input</th>
<th>Pattern $P$ of length $m$ over $\Sigma$ of size $t$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output</td>
<td>Automaton $A = (Q, A \cup {#}, \delta, s, F, \Sigma, B, \delta_f, H)$</td>
</tr>
</tbody>
</table>

1. ConstructVariablePattern($P$)
2. add $q_0$ to $Q$
3. for $i \leftarrow 1$ to $m + 1$
   4. if $i \neq m + 1$ then
      5. add $q_i$ to $Q$
      6. $\delta(q_{i-1}, A[j]) \leftarrow q_i$, for $P'[i] = A[j]$
   7. if $i = 2$ then
      8. $\delta_f(q_1) \leftarrow q_0$
      9. add $(A[1] \leftarrow \#) \rightarrow H(q_1)$
   10. else if $i > 2$ then
      11. find the smallest $i'$ such that $Pals(P'[1 : i - i']) = Pals(P'[i' : i - 1])$
      12. $\delta_t(q_{i-1}) \leftarrow q_{i-1}$
      13. for $l \leftarrow 1$ to $i - i'$ do
      15. for each $A[h]$ in $Pals(P'[1 : i - 1])$ without injective function in $H(q_{i-1})$ do
         16. add $(A[h] \leftarrow \#)$ to $H(q_{i-1})$ for $A[h]$
   17. return $(Q, A \cup \{\#\}, \delta, q_0, \{q_m\}, \Sigma, B, \delta_f, H)$
except for loops require constant time, the total time complexity is \(O(m^2}\). For the space complexity, there are \(O(m)\) states in \(A\). 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 \(A\). Based on the Knuth–Morris–Pratt algorithm, Algorithm 3 processes \(T\) in \(A\) 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 \(P_0_1, \ldots, P_0_k\), while merging all \(B[m_i]s\) to one \(B[k][m_i]\). It is straightforward to show that we call ConstructMultiVariablePattern, runs in \(O(mkM)\) time using \(O(mkM)\) space.

We define an automaton \(B = (Q, A \cup \{\#\}, \delta, s, F, \Sigma, B, \delta_f, H, \delta_p)\). The definition of \(B\) is similar to the definition of \(A\), except for an additional parameter: The pattern suffix transition function \(\delta_p: Q \rightarrow Q\) contains transitions to find multiple matching occurrences on a single state. The automaton \(B\) simulates the Aho–Corasick algorithm, using \(P_0_1, \ldots, P_0_k\) instead of \(P_1, \ldots, P_k\) as patterns. Algorithm 4 constructs \(B\) 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 \(Pals(P_i)\) in \(O(m_i)\) time and, based on \(Pals(P_i)\), lines 14
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 estimate three values—the preprocessing time \( t_p \), the query time \( t_q \), and 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 \( \frac{1}{c} \) 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 \( m = 10 \text{ to } 1000 \) 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 ≤ m ≤ 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).](http://bioinformatics.oxfordjournals.org/Downloaded from http://bioinformatics.oxfordjournals.org/at Your University on April 22, 2016)
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.

\[
t_q = O(nm)\]

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 \( n \).

\[
t_q = O(mn)\]

We observe that \( t_q \) is independent from \( M \) and \( m_k \) since \( t_q = O(mn) \). Note that \( t_q \) for \( m = 1000 \) is 468.78 when \( m = 1000 \).

\[
t_p = O(m_k M)\]

We observe that \( t_p \) is proportional to \( M \) and \( m_k \) since \( t_p = O(m_k M) \). It is not clear whether or not \( t_p \) 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 palindrom pattern matching in $O(m^2 n)$ 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 palindrom pattern matching in $O(m_k M) \times m_k n$ preprocessing time and $O(m_k M) \times m_k n$ 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 palindrom 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 palindrom 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
Hopcroft, J.E. and Ullman, J.D. (1979). Introduction to Automata Theory, Languages, and Computation. Addison-Wesley, Boston, USA.