Exact Circular Pattern Matching Using the BNDM Algorithm

K. H. Chen, G. S. Huang* and R. C. T. Lee

Department of Computer Science and Information Engineering, National Chi Nan University, Puli, Nantou, Taiwan.
{shieng, rctlee}@ncnu.edu.tw

ABSTRACT

In this paper, we discuss a problem that performs searching a circular string on text from bit-parallelism. Given a string \( X = x_1x_2 \cdots x_m \), a string \( X^i = x_ix_{i+1} \cdots x_{m}x_1 \cdots x_{i-1} \), for \( 1 \leq i \leq m \), is a circular string of \( X \). Given a text string \( T = t_1t_2 \cdots t_n \) and a pattern \( P \), the circular pattern matching problem is to find all occurrences of \( P \) in the text \( T \). Our algorithm only uses the composition of bitwise logical operations and basic arithmetic operations, and applies this technique to solve the problem. The algorithm is given name CBNDM. Our idea is based on the BNDM algorithm. We give some experiments to verify that it has good behavior for random strings.

1 Introduction

In this paper, we are interested in the following problem. Given a string \( X = x_1x_2 \cdots x_m \), a string \( X^i = x_ix_{i+1} \cdots x_{m}x_1 \cdots x_{i-1} \), for \( 1 \leq i \leq m \), is a circular string of \( X \) [16]. Given a text string \( T = t_1t_2 \cdots t_n \) and a pattern \( P \), the circular pattern matching problem is to find all occurrences of \( P \) in the text \( T \). Throughout the paper, given \( A = a_1a_2 \cdots a_n \), we use \( A(i,j) \) to denote \( a_ia_{i+1} \cdots a_j \) (notice that if \( j < i \), it does mean \( A(i,n)A(1,j) \)).

There exists a linear time algorithm to determine whether \( T \) is a circular rotation of \( P \) if \( T \) and \( P \) have the same length and \( T \) consists of a suffix of \( P \) followed by a prefix of \( P \) (see [8] for details). In our case, we further consider the length \( n \) of \( T \) and the length \( m \) of \( P \) are different such that \( P \) is a circular pattern and \( m < n \). It implies the algorithm must have shift strategy in each attempt.

For example, assume that \( T = \text{acgttaccat} \) and \( P = \text{cctta} \). We then find that a circular pattern \( P^3 = \text{ttacc} \) occurs in \( T[4, 8] \). Some special case applications of this problem, for example, in image processing analysis, or in bacterial and mitochondrial DNA analysis can be found in [8].

To solve this problem, a simple method is to transform the problem to the multiple pattern matching problem. For a pattern \( P \) of length \( m \), we produce \( m \) patterns \( P^1, P^2, \ldots, P^m \) from the original pattern \( P \). For example, in the above example, \( P = \text{cctta} \). We have \( P^1 = \text{cctta}, P^2 = \text{cttac}, P^3 = \text{ttacc}, P^4 = \text{tacct}, P^5 = \text{acctt} \).

We then can solve it by some proposed multiple-pattern matching algorithms [1, 2, 3, 4, 5, 7, 9, 10, 12, 13, 15, 17]. However, we like to point out that our problem is not a general case of the multiple pattern matching problem since in a general case multiple pattern matching problem, the patterns are not inter-related while in our case, the patterns are inter-related. Therefore, we shall point out that we can use this special property to design an efficient algorithm for our purpose.

Our algorithm is based upon a bit-parallel technique, called BNDM (Backward Nondeterministic Dawg Matching) algorithm [13]. It belongs to factor-based algorithm [14]. Therefore, we use the reverse factor algorithm [6] to introduce the basic idea of the factor-based approach. We shall show that the BNDM algorithm can be modified to suite our problem. We call our algorithm the CBNDM (Circular BNDM) algorithm.

The organization of this paper is as follows. We first introduce the basic idea of the reverse factor algorithm and using bit-parallel technique which is called the BNDM algorithm for single pattern case in Sect. 2 and Sect. 3 respectively. Section 4 introduces how to use the multiple version of the BNDM algorithm which is called the multiple BNDM algorithm to solve the circular pattern matching problem. In Sect. 5, we propose...
our algorithm for circular pattern case. The time complexity is analyzed in Sect. 6. We also provide several useful experimental results in Sect. 7. Finally, we conclude this work in Sect. 8.

2 The Single Pattern Matching Solved by the Reverse Factor Algorithm

The reverse factor algorithm was proposed in [6]. A comprehensive discussion of the algorithm can be found in [11]. Before introducing the algorithm, we need to define a term, called proper suffix. A proper suffix of a string \( A \) is a suffix of \( A \) which is not exactly \( A \). The basic principles of this algorithm can be illustrated as follows:

1. Given a text string \( T \) and a pattern string \( P \) where \(|P| = m\), we open a window \( W \) with size \( m \).

2. Let \( LSP(W,P)(LPSP(W,P)) \) denote the longest (proper longest) suffix of \( W \) which is equal to a prefix of \( P \). If \(|LSP(W,P)| < m\), we shift \( m - |LSP(W,P)| \) steps. Otherwise, \(|LSP(W,P)| = m\) and an exact match is found. We report this and shift \( m - |LPSP(W,P)| \) steps.

Consider the case where \( T = \text{acgtttga} \) and \( P = \text{tttga} \). The first window of \( T = \text{acgtttga} \) will be \( W = \text{acgtt} \) as shown below:

\[
T = \text{acgtttga} \\
P = \text{tttga}
\]

We can see that \( LSP(W,P) = \text{tt} \). We shift the pattern so that the prefix \( \text{tt} \) of \( P = \text{tttga} \) is aligned with the suffix \( \text{tt} \) of \( W = \text{acgtt} \) as shown below:

\[
T = \text{acgtttga} \\
P = \text{tttga}
\]

Now, an exact match is found.

To find \( LSP(W,P) \), we may build the suffix tree of the reverse of \( P \) in the preprocessing phase and scan the window from right to left in searching phase. However, the suffix tree approach to solve the reverse factor algorithm has a problem that it is not easy to implement. Therefore, Navarro and Raffinot proposed a bit-parallelism approach to simplify the reverse factor algorithm which is called the BNDM algorithm [13].

3 The Single Pattern Matching Solved by the BNDM Algorithm

For succinctness, we use the following notation in the bit-parallel technique. Exponentiation stands for bit repetition (e.g. \( 0^1=0001 \)). The window which we open is denoted as \( W = w_1w_2 \cdots w_m \). In the bit operators, “&” represents bitwise-AND operation. The shift left operation, “\( \ll \)”, moves the bits to the left and enters zeros from the right.

To use the bit-parallel approach, we first define an incidence matrix \( B \) of size \( r \) by \( m \) where \( r \) is the alphabet size. \( B[a] \) is a row of \( B \) that corresponds to the letter \( a \) which is set to be 1 at \( i \)-th position if and only if \( p_i = a \). We call \( B[a] \) a mask.

For example, assume that \( P = \text{acggt} \). We then have the masks \( B[\text{a}] = (10000) \), \( B[\text{c}] = (01000) \), \( B[\text{g}] = (00110) \) and \( B[\text{t}] = (00001) \). For the \( j \)-th iteration, we read \( w_{m-j+1} \) for \( 1 \leq j \leq m \). Let \( D_j = d_{1,j}d_{2,j} \cdots d_{m,j} \) be a bit vector. The bit \( d_{i,j} \) is set to 1 if and only if \( p_i \cdots p_{i+j-1} = w_{m-j+1} \cdots w_m \) for some \( j \). Let \( W = \text{acgg} \) and \( P = \text{ggca} \). Then \( D_2 = (1000) \) because \( p_1p_2 = \text{gg} = w_3w_4 \). Therefore, if \( d_{1,2} = 1 \), it means that \( p_1 \cdots p_j = w_{m-j+1} \cdots w_m \) and we have found a prefix of \( P \) which is equal to a suffix of \( W \).

This particular information is very important. We shall use a parameter \( \text{last} \) to record this information. That is, if \( d_{1,j} = 1 \) after the \( j \)-th iteration, \( \text{last} = j \).

Initially, we set \( D_0 = 1^m \) and \( \text{last} = 0 \). We scan the window \( W \) from right to left. When a new character in \( W \) was scanned at the \( j \)-th iteration, we set \( D_j' \) based on the following rule:

\[
D_j' = (D_{j-1} \& B[w_{m-j+1}])
\]

If \( D_j' = 0^m \), for some \( j \), we stop this attempt. Then \( |LSP(W,P)| = \text{last} \) and we shift \( S = m - \text{last} \) steps.

If \( D_j' \neq 0^m \) and \( j < m \), we check whether \( d_{1,j} = 1 \). We set \( \text{last} \) to be \( j \) if \( d_{1,j} = 1 \) and take no action if otherwise. After checking whether \( d_{1,j} = 1 \), we shift vector \( D_j' \) one bit to the left, fill the rightmost bit to be 0 and store this value to \( D_j \). For example, assume that \( D_j' = (0101) \), after shifting, we have \( D_j = (1010) \). This is denoted as

\[
D_j = D_j' \ll 1
\]

and continue the next iteration.

If \( D_j' \neq 0^m \) and \( j = m \), it means that we have found an exact match of \( P \) in \( T \) and \(|LSP(W,P)| = m \). We then stop the attempt,
report a match and shift $S = m - \text{last}$ steps. Algorithm 1 shows a complete description of the single pattern matching using the BNDM algorithm. Notice that in Algorithm 1, we omit the subscript $j$ for all variables such as $D_j$ and $d_{i,j}$ for the purpose to re-use memory.

Algorithm 1: The single pattern matching using the BNDM algorithm

**Input:** Two strings $T = t_1 \ldots t_n$ and $P = p_1 \ldots p_m$

**Output:** The locations of all occurrences of $P$ in $T$

1. **Begin**
   2. Set $i = m$.
   3. Compute the incidence matrix $B$ of $P$.
   4. While $i \leq n$ do
      5. $j = 0$ and last = 0.
      6. $D = 1^m$.
      7. $W = T(i - m + 1, i)$.
      8. While $D \neq 0^m$ do
         9. $j = j + 1$.
         10. $D = D \& B[w_{m-j+1}]$.
         11. If $D = 0^m$ then exit.
         12. If $j = m$ and $d_1 = 1$ then report an occurrence of $P$ at position $i - m + 1$ of $T$ and exit.
         13. If $j < m$ and $d_1 = 1$ then last = $j$.
         14. $D = D \ll 1$.
      15. **End of while**
      16. $i = i + 1 (m - \text{last})$.
      17. **End of while**
   18. **End**

For example, we assume that $T = \text{acggtgac}$, $P = \text{tgac}$. We begin with $T = \text{acggtacc}$, $D_0 = (1111)$, $B[a] = (0010)$, $B[c] = (0001)$, $B[t] = (1000)$, $B[g] = (0100)$, $m = 4$, $n = 8$, last = 0.

In the first attempt, the first window is $W = \text{acgg}$. In the first iteration, we have $D_1 = (1111)\& B[g] = (1111) \& (0100) = (0100)$. After left-shifting of $D_1$ once, we have $D_1 = (1000)$ and last = 0.

In the second iteration, we found that $D_2 = (1000) \& (0100) = (0000)$ and last = 0. Therefore, $|LSP(W, P)| = 0$. We then shift the window $S = 4 - 0 = 4$ steps right. The processing of the first attempt is shown in Figure 1.

In the second attempt, we have $W = \text{tgac}$. We found that $j = 4$ and $d_1 = 1$ in the final iteration of this attempt. It means that we have found an occurrence of $P$ in $T(5,8)$. We report a match and get $|LPSP(W, P)| = 0$ because last = 0. The shift $S = 4 - 0 = 4$. Because the end position of the window has exceeded the end of the text, we stop searching. The processing of the second attempt is shown in Figure 2.

4 The Circular Pattern Matching Problem Solved by the MultiBNDM Algorithm

We can reduce the circular pattern matching problem to the multiple matching problem, by expanding the circular pattern $P$ into ordinary patterns $P^1, \ldots, P^m$ and then checking their occurrences in the text. In the following, we shall introduce MultiBNDM algorithm [13, 14], a multiple pattern matching algorithm which is based upon the reverse factor algorithm and uses the bit-parallel technique. It concatenates $m$ patterns into a string $X = P_1^1P_2^2 \ldots P_m^m$ with length $M$ where $M = m^2$. For example, let $P = \{\text{cct, aca, gtc}\}$. Then the concatenation of $X = \text{cctacagtc}$.

The preprocessing phase of the MultiBNDM algorithm is similar to the BNDM algorithm. The incidence matrix $B$ with size $r$ by $M$, the bit vector $D = d_1d_2 \ldots d_M$ and sentinel last are still used. In each attempt, it finds last = max($LSP(W, P^i)$), $1 \leq i \leq m$. Then shift $S = m - \text{last}$. Let us assume that $T = \text{accc}$. For the set of patterns, the length of each pattern is 3. We therefore set the window to be $W = T(1,3) = \text{acc}$ with length 3. For each pattern $P^i$ in $P = \{\text{cct, aca, gtc}\}$, it can be seen that $|LSP(W, P^1)| = 2$, $|LSP(W, P^2)| = 0$ and $|LSP(W, P^3)| = 0$. Shift $S = 3 - 2 = 1$.

As before, we will use a vector $D_j$. For the $j$-th iteration, we read $w_{m-j+1}$ for $1 \leq j \leq m$. Let $D_j = d_1d_2d_3 \ldots d_{m-j+1}d_{m-j}$ be a bit vector with length $m^2$ bits. Consider the bit $d_{a,j}$ in $D_j$. Note that $a = m(k-1) + i$. Thus the bit $d_{a,j}$ in $D_j$ is set to 1 if and only if $p_1^k \cdots p_j^k = w_{m-j+1} \cdots w_m$ where $k = \lfloor a/m \rfloor$ and $i = a - m(k-1)$. Therefore, if $i = 1$ and $d_{m(k-1)+1,j} = d_{m(k-1)+1,j} = 1$ in $D_j$, this means that we have found a prefix $p_1^k \cdots p_j^k$ of $P^k$ which is equal to the suffix $w_{m-j+1} \cdots w_m$ of $W$.

Let $W = \text{tcc}$ and $P = \{\text{cct, aca, gtc}\}$. For $j = 2$, we will find that $p_1^2p_2^2 = \text{cc} = w_3w_4$. In such a situation, we have $j = 2$, $i = 1$ and $k = 1$. Thus $a = m(k-1) + i = 3(1-1) + 1 = 1$ and $D_2 = (10000000)$. Initially, we set $D_0 = 1^m$ and last = 0. We scan the window from right to left. When a new character in the window was scanned at the $j$-th iteration, we set $D'_j$ based on Equation (1) in Sect. 3.

$$D'_j = (D_{j-1} \& B[w_{m-k+1}])$$

If $D'_j = 0^{m^2}$, for some $j$, we stop the scanning.
The 1st iteration

\[ T = \text{acggtgac} \]
\[ P = \text{tgac} \]
\[
\begin{array}{l}
D_0 = 1111 \\
& \& B[c] = 0001 \\
D_1 = 0001 << 1 \\
last = 0 \quad j = 1
\end{array}
\]

The 2nd iteration

\[ T = \text{acggtgac} \]
\[ P = \text{tgac} \]
\[
\begin{array}{l}
D_0 = 0000 \\
& \& B[g] = 0100 \\
D_1 = 0100 << 1 \\
D_2 = 1000 \\
last = 0 \quad j = 2
\end{array}
\]

Figure 1: Illustration of the first attempt.

The 1st iteration

\[ T = \text{acggtgac} \]
\[ P = \text{tgac} \]
\[
\begin{array}{l}
D_0 = 1111 \\
& \& B[c] = 0001 \\
D_1 = 0001 << 1 \\
last = 0 \quad j = 1
\end{array}
\]

The 2nd iteration

\[ T = \text{acggtgac} \]
\[ P = \text{tgac} \]
\[
\begin{array}{l}
D_0 = 0000 \\
& \& B[g] = 0100 \\
D_1 = 0100 << 1 \\
D_2 = 1000 \\
last = 0 \quad j = 2
\end{array}
\]

The 3rd iteration

\[ T = \text{acggtgac} \]
\[ P = \text{tgac} \]
\[
\begin{array}{l}
D_0 = 1000 \\
& \& B[\kappa] = 0100 \\
D_1 = 0100 << 1 \\
D_2 = 1000 \\
D_3 = 1000 \\
last = 0 \quad j = 3
\end{array}
\]

The 4th iteration

\[ T = \text{acggtgac} \]
\[ P = \text{tgac} \]
\[
\begin{array}{l}
D_0 = 1000 \\
& \& B[\kappa] = 0100 \\
D_1 = 0100 << 1 \\
D_2 = 1000 \\
D_3 = 1000 \\
D_4 = 1000 \\
last = 0 \quad j = 4
\end{array}
\]

Report an occurrence.

\[ |LSP(W, P)| = 0 \]

Figure 2: Illustration of the second attempt.

Then we shift \( S = m - \text{last} \) steps.

Let \( DF = (10^{m-1})^m \). If \( D_j' \neq 0^m \) and \( j < m \), we check whether \( (D_j' \& DF) = 0^m \). Recall that \( DF \) has bit one in the highest bit position of each \( m \)-bit field, and zeros elsewhere. The operation \( D_j' \& DF \) is to find out whether there is a prefix of \( P^k \) which is equal to a suffix of \( W \), for \( 1 \leq k \leq m \). If \( (D_j' \& DF) \neq 0^m \) and \( d_{m(k-1)+1,j} = 1 \) for some \( k \), it means that we have found a prefix of \( P^k \) with length \( j \) which is equal to a suffix of \( W \). We set \( \text{last} \) to be \( j \) if \( (D_j' \& DF) \neq 0^m \) and take no action otherwise. After checking whether \( (D_j' \& DF) = 0^m \), we shift vector \( D_j' \) one bit to the left, fill the rightmost bit to be 0 for each \( m \)-bit field, and store the result to \( D_j \). For example, assume that \( D_j' = (001011001) \) and \( m = 4 \), after shifting, we have \( D_j = (010100010) \).

There is one problem here. After left shifting, the highest bit corresponding to \( P^k \) will become the lowest bit of \( P^{k-1} \). To solve this problem, we define a bit mask \( CL = (1^{m-1}0)^m \) and perform the following operation:

\[ D_j = (D_j' \ll 1) \& CL \] (3)

If \( D_j' \neq 0^m \) and \( j = m \), it means that we have found an exact match of \( P^k \) for some \( k \) in \( T \). We then report a match and shift \( S = m - \text{last} \) steps.

Algorithm 2 shows a complete description for the circular pattern matching using the MultiBNDM algorithm.

If we use the MultiBNDM algorithm directly to solve the circular pattern matching problem, the time complexity is \( O(nm \lfloor m/\kappa \rfloor) \) and space complexity is \( O(m^2) \) where \( \kappa \) is the machine word size [13, 14]. Therefore, we propose a new algorithm whose time complexity is \( O(nm \lfloor m/\kappa \rfloor) \) and space complexity is \( O(m) \) (see Sect. 6 for details). We call our algorithm the CBNDM (Circular Backward Nondeterministic Dawg Matching) algorithm which will be introduced in the next section.

5 The Circular Pattern Matching Problem Solved by the CBNDM Algorithm

In this section, we present our algorithm for solving the circular pattern matching problem. The algorithm is called CBNDM. The preprocessing phase of the CBNDM algorithm is the same as the BNDM algorithm for the single pattern matching.

To find this in a bit-parallel manner, we define an incidence matrix \( B_c \) with size \( r \) by \( m \). \( B_c[a] \) is set to be 1 at the \( i \)-th position if and only if \( p_i = a \) for letter \( a \) in the alphabet. From our example where \( P = \text{ggtca} \) and \( W = \text{accag} \), \( B_c[a] = (00001), B_c[c] = (00010), B_c[g] = (11000), B_c[t] = (00100) \).

For the \( j \)-th iteration of a window, let \( C_j = \)
Algorithm 2: The circular pattern matching using the MultiBNDM algorithm

Input: A string $T$ of length $n$ and $m$ patterns, $P^1, P^2, \ldots, P^m$ rotated from the original pattern $P$ respectively.

Output: The locations of all occurrences of $P^k$ for some $k$ in $T$

1. Begin
2. Set $i = m$, $pos = m$ and $M = m^2$.
3. Set $X = P^1 \ldots P^m$.
4. Compute the incidence matrix $B$ of $X$.
5. Set $CL = (1^{(m-1)}0^m)$.
6. Set $DF = (10^{(m-1)}m)$.
7. While $pos \leq n$ do
   8. $j = 0$ and last = 0.
   9. $D = 1^{m^2}$.
   10. $W = T(pos - m + 1, pos)$.
   11. While $D \neq 0^{m^2}$ do
       12. $j = j + 1$.
       13. $D = D \& B[w_{m-j+1}]$.
       14. If $D = 0^{m^2}$ then exit.
       15. If $j = m$ and $(D \& DF) \neq 0^{m^2}$ then report an exact match at $t_{pos-m+1}$ and exit.
       16. If $j < m$ and $(D \& DF) \neq 0^{m^2}$ then last = $j$.
   17. $D = (D \ll 1) \& CL$.
18. End of while
19. $pos = pos + (m - last + 1)$.
20. End of while
21. End

$c_1c_2 \cdots c_m$ be a bit vector. The bit $c_i$ is set to be 1 if and only if there exists a prefix of circular string $P^i$ which is of length $j$ and equal to $w_{m-j+1} \cdots w_m$, for some $i$ and $j$. Let $W = accag$ and $P = ggtca$. Note that $P^4 = p_4p_5p_6p_7p_8 = caggt$. Then $C_3 = (00010)$ because $p_4p_5p_7 = cag = w_3w_4w_5$. Thus, we have found a prefix of $P^4$ which is equal to a suffix of $W$ with length 3. To achieve this goal, we define the left-rotation operation, $\ll c_{i,j}$ which rotates vector one bit left and fill its rightmost bit to be the first bit. For instance, if $C_j = (10010)$, $C'_{j} = (000101)$. Recall that $C_j = c_{1,j}c_{2,j} \cdots c_{m,j}$. After $C'_{j} = C_j \ll 1$ operation, we have a new bit vector $C''_{j} = c'_{1,j}c'_{2,j} \cdots c'_{m,j} = c_{2,j} \cdots c_{m,j}$.

Initially, we set $C_0 = 1^m$. We scan the window $W$ from right to left. For the $j$-th iteration, we read $w_{m-j+1}$ for $1 \leq j \leq m$. Let $LSF_j(W, P)$ be the longest suffix of the window which is equal to a circular pattern of the circular pattern. When a new character in the window was scanned at the $j$-th iteration, we perform an AND operations of $B_c$ and $C_{j-1}$ based on Equation (4).

$$C_j = (C_{j-1} \ll 1) \& B_c[w_{m-j+1}] \quad (4)$$

If $C_j = 0^m$ for some $j$, we stop the scanning. Then $|LSF_j(W, P)| = j - 1$ and we shift the window $S = m - (j - 1)$ steps for the next attempt.

If $C_j = c_1c_2 \cdots c_{m-j} \neq 0^m$, $j < m$ and $c_1 = 1$, it means that the suffix $w_{m-j+1} \cdots w_m$ of $W$ is equal to the prefix of $P^1$ with length $j$ and we continue the next iteration.

If $C_j \neq 0^m$ and $j = m$, it means that we have found an exact match of $P^1$ in $T$ and $|LSF_m(W, P)| = m$. We then stop the iteration, report a match and shift the window $S = 1$ step.

Recall that the example $P = ggtca$ and $W = accag$. In the first iteration, since the rightmost character $w_5 = g$, based on Equation (4), we perform a left rotation of $C_0 = (11111)$. We then obtain $C'_0 = (11111)$. After performing an AND operations of $B_c[g] = (11000)$ and $C'_0 = (11111)$, we have $C_1 = (11000)$.

Algorithm 3 shows a complete description of the CBNDM algorithm.

Algorithm 3: The circular pattern matching using the CBNDM algorithm

Input: Two strings $T$ and $P$ with length $n$ and $m$ respectively.

Output: The locations of all occurrences of $P^i = p_1p_2p_3 \cdots p_m$ where $1 \leq i \leq m$ in $T$

1. Begin
2. Set $pos = m$.
3. Compute the incidence matrix $B_c$ of $P$.
4. While $pos \leq n$ do
   5. $j = 0$.
   6. $C_0 = 1^m$.
   7. $W = T(pos - m + 1, pos)$.
   8. While $C_j \neq 0^m$ do
      9. $j = j + 1$.
      10. $C_j = (C_{j-1} \ll 1) \& B_c[w_{m-j+1}]$.
      11. If $C_j = 0^m$ then exit.
      12. If $C_j \neq 0^m$, $j = m$ and $c_1 = 1$ then report an exact match at position $pos - m + 1$ of $T$ and exit.
   13. End of while
   14. $pos = pos + (m - j + 1)$.
15. End of while
16. End

We give a complete example that $T = atggtcac$ and $P = tcag$. In the preprocessing phase, the incidence matrix $B_c$ of $P$ is shown as follows.

$$B_c[a] = (0010),$$
$$B_c[c] = (0100),$$
$$B_c[g] = (0001),$$
$$B_c[t] = (1000).$$

Initially, $C_0 = 1^m$. In the first attempt, $j = 1$
\[ P = \text{tcag} \quad j = 1 \quad W = \text{atgg} \]

\begin{align*}
B_{t} & = 1000 \\
B_{c} & = 0100 \\
B_{a} & = 0010 \\
B_{g} & = 0001 \\
C_{0} & = 1111 \\
C_{1} & = 1111 \\
& \quad B_{c}[g] = 0001 \\
& \quad C_{2} = 0000
\end{align*}

\[ T = \text{atggcac} \]

\[ S = 4 \]

\[ C_{1} = 0010 \]

\[ C_{2} = 0100 \]

Figure 3: The first iteration in the first attempt.

\[ P = \text{tcag} \quad j = 2 \quad W = \text{atgg} \]

\begin{align*}
B_{t} & = 1000 \\
B_{c} & = 0100 \\
B_{a} & = 0010 \\
B_{g} & = 0001 \\
C_{0} & = 0100 \\
& \quad B_{c}[g] = 0001 \\
& \quad C_{1} = 0010 \\
& \quad C_{2} = 0000
\end{align*}

\[ T = \text{atggcac} \]

\[ S = 3 \]

\[ C_{1} = 0001 \]

\[ C_{2} = 0100 \]

Figure 4: The second iteration in the first attempt.

and we scan the characters of window \( W = \text{atgg} \) from right to left. In the first iteration, the rightmost character \( w_{m-j+1} = w_{4-1+1} = 4 \) is \( g \) and it is shown in Figure 3.

In Figure 3, after a bit-rotation operation and an AND operation, we found \( C_{1} = ((1111) \gtrless 1) \& (0001) = (0001) \). In \( C_{1} \), \( c_{4} = 1 \). Thus we have \( j = 1 \) and \( i = 4 \). This means that the suffix \( w_{m-j+1} = w_{4} \) is equal to the prefix of \( P^{4} = \text{gtca} \) with length \( j = 1 \). We continue the next iteration to read the next character of \( W \).

In the second iteration, \( j = 2 \), we have \( C_{1} = (0001) \) and \( w_{3} = g \). After calculating, we found that \( C_{2} = ((0001) \gtrless 1) \& (0001) = (0000) \) containing all 0’s. We stop the attempt. Note that \( |LSP_{C}(W, P)| = j - 1 = 1 \). Then the shift \( S = 4 - 1 = 3 \) steps, which is shown in Figure 4.

In the second attempt, initially \( j = 1 \) and we scan the characters of window \( W = \text{gtca} \) from right to left. We start with the rightmost character \( w_{4} = a \). The processing for the first iteration of the second attempt is shown in Figure 5.

In Figure 5, we found that \( C_{1} = ((1111) \gtrless 1) \& (0010) = (0001) \) based on Equation (4). In \( C_{1} \), \( c_{3} = 1 \). We have \( j = 1 \) and \( i = 3 \). This means that the suffix \( w_{m-j+1} = w_{4} \) is equal to the prefix of \( P^{3} = \text{agtc} \) with length \( j = 1 \). We continue the next iteration.

In the third iteration, \( j = 2 \), we have \( C_{1} = (0010) \) and the character of \( w_{3} \) is \( c \). After computing, we found that the bits in \( C_{2} = ((0010) \gtrless 1) \& (0100) = (0100) \) are not all 0’s. In \( C_{2} \), \( c_{2} = 1 \). Thus we have \( j = 2 \) and \( i = 2 \). This means that the suffix \( w_{3}w_{4} \) is equal to the prefix of \( P^{2} = \text{cagt} \) with length \( j = 2 \). We continue the next iteration. The result is shown in Figure 6.

In the third iteration, \( j = 3 \), we have \( C_{2} = (0100) \) and the character \( w_{2} = t \). After computing, we found that the bits in \( C_{3} = ((1001) \gtrless 1) \& (1000) = (1000) \) are not all 0’s. In \( C_{3} \), \( c_{1} = 1 \). Thus we have \( j = 3 \) and \( i = 1 \). This means that the suffix \( w_{2}w_{3}w_{4} \) is equal to the prefix of \( P^{1} = \text{tcag} \) with length \( j = 3 \). The process for the third iteration in the second attempt is shown in Figure 7.

In the fourth iteration, \( j = 4 \), we have \( C_{3} = (1000) \) and the character \( w_{1} = g \). We found that the bits in \( C_{4} = ((0010) \gtrless 1) \& (0001) = (0001) \) are not all 0’s and \( j = m = 4 \). We report the existence of an exact match and set \( |LSP_{C}(W, P)| = 4 \). Then the shift \( S = m - (j - 1) = 4 - 3 = 1 \), which is shown in Figure 8.

In the third attempt, \( j = 1 \) and we scan the characters of window \( W = \text{tcac} \) from right to left. We start with the rightmost character \( w_{4} = c \). The process for the first iteration in the third attempt is shown in Figure 9. In Figure 9, we get
$P = tcag \quad j = 4 \quad W = gtca$

\begin{align*}
P_1 & = tcag \quad j = 4 \quad W = gtca \\
T_1 & = \text{atg}\text{gtc} \\
C_1 & = 1000 \\
C_2 & = 0000 \\
C_3 & = 0001 \\
C_4 & = 0010 \\
& \downarrow \quad B_4,c = 0010 \\
\text{Report a match}
\end{align*}

$|LSP(P,W)| = 4$

Figure 8: The fourth iteration in the second attempt.

$P = tcag \quad j = 1 \quad W = tcac$

\begin{align*}
P_2 & = tcag \quad j = 1 \quad W = tcac \\
T_2 & = \text{atg}\text{gtc} \\
C_1 & = 0100 \\
C_2 & = 0000 \\
C_3 & = 0010 \\
C_4 & = 0010 \\
& \downarrow \quad B_4,c = 0010 \\
C_1 & = 0100
\end{align*}

$C_1 = (0100)$ and $C_2 = 1$. Thus we have $j = 1$ and $i = 2$. This means that the suffix $w_4$ is equal to the prefix of $P^2 = cagt$ with length $j = 1$.

In the second iteration, $j = 2$, we have $C_1 = (0100)$ and the character $w_3 = a$. After bit-rotation operation and AND operation, we found that $C_2 = ((0100) \otimes 1) \& (0010) = (0000)$ and then we stop the process. The shift $s = 4 - (j - 1) = 4 - 1 = 3$, which is shown in Figure 10. After shifting, the boundary of the window is larger than $m$. We stop.

Next, we show the correctness of Algorithm 3. It is not difficult to see that the following loop invariant property holds at Line 11 in Algorithm 3, by setting $\ell = m - j$ and $k = (i + \ell - 1) \mod m + 1$:

$$\text{Inv}(j) : c_k = 1 \iff P^i(\ell + 1, m) = W(\ell + 1, m)$$

for $1 \leq i \leq m$, where $c_k$ is the $k$th bit of $C_j$. When $j = m$, Inv($m$) asserts $c_i = 1$ iff

$P = tcag \quad j = 2 \quad W = tcac$

\begin{align*}
P_3 & = tcag \quad j = 2 \quad W = tcac \\
T_3 & = \text{atg}\text{gtc} \\
C_2 & = 0100 \\
C_3 & = 0010 \\
C_4 & = 0000 \\
& \downarrow \quad B_4,a = 0010 \\
\text{Report a match}
\end{align*}

Figure 10: The second iteration in the third attempt.

$P^i(1, m) = W(1, m)$. It indicates there is a match (cf. Line 12). When $C_j = 0^m$ for some $j$, it implies there is no match (cf. Line 11).

6 The Time-Complexity Analysis

The time complexity of CBNDM is analyzed as follows. We first consider its worst-case time complexity. The preprocessing phase, which computes the $B_c$ incidence matrix, takes $O(rm)$ time. If the alphabet size $r$ is a constant, it is reduced to $O(m)$. In the searching phase, there are at most $O(n)$ sliding windows in worst case since the shift of a sliding window can be as small as 1. For each sliding window, at most $m$ characters on it can be scanned from the right end to the left end. Therefore there are at most $O(m)$ iterations of Line 10 in Algorithm 3. Each execution of Line 10 takes $O(\lceil m/k \rceil)$ time. Combining all of these facts together, we have $O(n \times m \times \lceil m/k \rceil) = O(nm \lceil m/k \rceil)$ time in worst case. The worst case happens when the text is $a^n$ and the pattern is $a^m$. Notice that the worst case time complexity for MultiBNDM solving the circular pattern matching problem is $O(nm \lceil m^2/k \rceil)$, which is an order of magnitude slower than CBNDM.

The worst-case time complexity for CBNDM seems very high, as compared to other text processing algorithms. However, this does not imply that CBNDM is impractical and inefficient. Let us consider average-time complexity for CBNDM. Since the analysis is exactly the same as BNDM shown in [13], we only give a sketch of the analysis and omit the details in this paper. The average shift for a sliding window is $\Theta(m)$, which implies that there are $O(n/m)$ sliding windows on the average. The average number of characters scanned in a sliding window is $O(\log_r m)$. During each scan, it takes $O(\lceil m/k \rceil)$ time to process bit-parallel operations. Therefore, the average-time complexity for CBNDM is $O(n \log_r m \times \lceil m/k \rceil)$, which is asymptotic to $O(n \log_r m/k)$. The term $\log_r m$ is usually very small. Due to the simplicity of CBNDM, the ignored constant coefficient in the asymptotic bound is relatively small. In our case, $k = 32$, which contributes totally to the speedup. This is why CBNDM is efficient. It is easy to see that the space complexity of CBNDM is $O(m)$ if $r = O(1)$.
7 Experimental Results

In this section, we give experimental results to demonstrate the practicability of the CBNDM algorithm. The experiments are conducted as follows. We maintain two sets of testing data, one for random text searching and the other for DNA sequence searching. We also implemented the CBNDM algorithm and the MultiBNDM algorithm, which are both written in C, compiled by GCC compiler and executed on a PC with 3Ghz Intel CPU, 1GB memory installed Fedora 6 OS. Notice that a machine word has 32 bits in our environment, and this parameter is crucial for practicing bit-parallelism.

We use the overall execution time to measure the performance of the algorithms. Since there are other factors, such as multitasking and caching, that may affect the execution time, we repeat each measurement 10 times and then take their average in our final report. We are also interested in another measure that reflects the average number of bit-parallel operations needed for processing one character on the text. This measure is got by recording the number of executions of Line 13 in Algorithm 2 and Line 10 in Algorithm 3, and then dividing this number by the length of the text. Let us name this measure the CPS (Comparisons Per Site on the text). Notice that CPS is irrelevant to the length of the machine word, and in fact, it should have the same value for both CBNDM and MultiBNDM. In order to adjust them to reflect the fact that a machine word is limited to 32 bits, we shall multiply the CPS value by the length of the bit vector and then divide it by 32. Let us refer to this measure as the NCPS (Normalized CPS).

As discussed in Sect. 6, CBNDM and MultiBNDM are both factor-based algorithms, and they are especially efficient when dealing with random text searching. However, even under this preference, CBNDM should perform much better than MultiBNDM because CBNDM has smaller pattern size to process, for the simple reason that it does not concatenate all circular strings altogether as what MultiBNDM does. This is the reason why CBNDM is more efficient than MultiBNDM while dealing with circular pattern matching.

7.1 An Experiment with Random Text Searching

In our first experiment, we investigate on the performance of CBNDM and MultiBNDM when dealing with random texts. We set parameters $n = 10^6$, $m \in \{10, 20, 30, 40, 50, 60, 70, 80, 90, 100\}$ and the alphabet size $r \in \{2, 4, 8, 16\}$. For each combination, we generate 100 random samples. That is, a length $m$ pattern and a length $n$ text are composed from a uniformly distributed random source with $r$ symbols for each testing sample. Each sample is processed and repeated 10 times by CBNDM and MultiBNDM, in order to get more accurate execution time.

The final results are shown in Tables 1 and 2. Table 1 lists the execution time of these two algorithms and Table 2 shows their NCPS values. The first columns in both tables indicate the pattern lengths of the testing samples. The second and third columns list the results for CBNDM and MultiBNDM with different alphabet sizes and pattern lengths. Their performance ratios are calculated in the fourth columns. As we can see in Table 1, CBNDM runs much faster than the MultiBNDM. From the discussion in Sect. 6, we know that CBNDM is an order of magnitude faster than MultiBNDM in both worst case and average case. This fact is approved by our first experimental results. Notice that the performance ratios in Table 1 are a little smaller than the corresponding entries in Table 2. This is because additional execution time is spent for lines other than line 13 in Algorithm 2 and line 10 in Algorithm 3. Hence, the ratios of execution time would become rather mild.

7.2 An Experiment with DNA Pattern Searching

In our second experiment, we test circular pattern searching for DNA sequences. We choose the top 5 longest chromosome contigs from humans as the texts. Their NCBI reference numbers, as well as the contig lengths, are shown in Table 3. We extract the first 100 nucleotides of NT_023603.5 as the pattern throughout this experiment. Notice that the alphabet size is always 4. The final results are shown in Table 4. Notice that the performance ratios are almost a constant. This is because the pattern length $m$ is fixed to 100. Another interesting point is to compare the NCPS values in Table 4 with Table 2, which is for random cases. From Table 2, if $m = 100$ and $r = 4$, we have NCPS = 0.1751 for CBNDM and 13.6987 for MultiBNDM. The ratio of this two value is 78.25, which is exactly the same as the NCPS ratio in Table 4. The NCPS values in Table 4 are around 1.5 for CBNDM and 12 for MultiBNDM, and these values are slightly better than the ran-
Table 1: The execution time (in seconds) for searching circular random patterns in a text.

<table>
<thead>
<tr>
<th>m</th>
<th>CBNDM(A)</th>
<th>MultiBNDM (B)</th>
<th>Ratio = (B)/(A)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>0.0022</td>
<td>0.0133</td>
<td>0.1751</td>
</tr>
<tr>
<td>4</td>
<td>0.0140</td>
<td>0.0691</td>
<td>0.1722</td>
</tr>
<tr>
<td>8</td>
<td>0.0094</td>
<td>0.0360</td>
<td>0.1592</td>
</tr>
<tr>
<td>16</td>
<td>0.0100</td>
<td>0.0200</td>
<td>0.1587</td>
</tr>
<tr>
<td>32</td>
<td>0.0057</td>
<td>0.0172</td>
<td>0.1571</td>
</tr>
<tr>
<td>64</td>
<td>0.0089</td>
<td>0.0141</td>
<td>0.1562</td>
</tr>
<tr>
<td>128</td>
<td>0.0077</td>
<td>0.0135</td>
<td>0.1553</td>
</tr>
<tr>
<td>256</td>
<td>0.0085</td>
<td>0.0131</td>
<td>0.1548</td>
</tr>
<tr>
<td>512</td>
<td>0.0076</td>
<td>0.0130</td>
<td>0.1543</td>
</tr>
<tr>
<td>1024</td>
<td>0.0068</td>
<td>0.0128</td>
<td>0.1539</td>
</tr>
<tr>
<td>2048</td>
<td>0.0079</td>
<td>0.0127</td>
<td>0.1535</td>
</tr>
</tbody>
</table>

Table 2: The NCPS values for searching circular random patterns in a text.

<table>
<thead>
<tr>
<th>m</th>
<th>CBNDM(A)</th>
<th>MultiBNDM (B)</th>
<th>Ratio = (B)/(A)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>0.8361</td>
<td>0.3148</td>
<td>2.6728</td>
</tr>
<tr>
<td>4</td>
<td>0.3733</td>
<td>0.1722</td>
<td>2.1898</td>
</tr>
<tr>
<td>8</td>
<td>0.2541</td>
<td>0.1237</td>
<td>2.0468</td>
</tr>
<tr>
<td>16</td>
<td>0.3911</td>
<td>0.0924</td>
<td>2.1420</td>
</tr>
<tr>
<td>32</td>
<td>0.3167</td>
<td>0.0492</td>
<td>2.1378</td>
</tr>
<tr>
<td>64</td>
<td>0.2698</td>
<td>0.0274</td>
<td>2.0468</td>
</tr>
<tr>
<td>128</td>
<td>0.3534</td>
<td>0.0139</td>
<td>2.5562</td>
</tr>
<tr>
<td>256</td>
<td>0.2821</td>
<td>0.0092</td>
<td>2.5972</td>
</tr>
<tr>
<td>512</td>
<td>0.3447</td>
<td>0.0044</td>
<td>2.5972</td>
</tr>
</tbody>
</table>

Table 3: The top 5 longest chromosome contigs of the human chromosome.

<table>
<thead>
<tr>
<th>NCBI reference</th>
<th>Length</th>
</tr>
</thead>
<tbody>
<tr>
<td>NT_005403.17</td>
<td>84213159</td>
</tr>
<tr>
<td>NT_026437.12</td>
<td>88289540</td>
</tr>
<tr>
<td>NT_0032077.9</td>
<td>90908613</td>
</tr>
<tr>
<td>NT_005612.16</td>
<td>100537107</td>
</tr>
<tr>
<td>NT_016354.19</td>
<td>115591997</td>
</tr>
</tbody>
</table>

8 Conclusion

In this paper, we propose the circular-pattern matching algorithm named CBNDM for solving it. The performance of CBNDM is an order of magnitude faster than the MultiBNDM for solving the circular-pattern matching problem. The time complexity of CBNDM is $O(nm/\lceil m/k \rceil)$ in worst case, and it is $O(n/m \times \log m \times \lceil m/k \rceil)$ in average case. The experimental results in Sect. 7 also agree with the theoretical bound for the average cases. Finally, we claim that the performance of CBNDM processing DNA sequence searching is as competitive as processing random text searching.

One might be curious about the performance of circular-pattern matching by using the Naïve algorithm, which expands a circular pattern into $m$ ordinary patterns and then checks the occurrences of these patterns for every position on the text. A check terminates immediately when it failed. Table 5 shows parts of the experimental results for random patterns, in the same setting as for Table 1. It reveals that the Naïve algorithm is the poorest, as one might expect.

References


Table 4: Results for DNA circular pattern searching.

<table>
<thead>
<tr>
<th>NCBI reference</th>
<th>(n)</th>
<th>CBNDM(A) NCPS</th>
<th>Run time (Sec.)</th>
<th>MultiBNDM (B) NCPS</th>
<th>Run time (Sec.)</th>
<th>Ratio (=\frac{(B)}{(A)})</th>
</tr>
</thead>
<tbody>
<tr>
<td>NT_005403.17</td>
<td>842</td>
<td>0.1541</td>
<td>0.290953</td>
<td>12.0607</td>
<td>15.753600</td>
<td>78.25</td>
</tr>
<tr>
<td>NT_026437.12</td>
<td>882</td>
<td>0.1526</td>
<td>0.302227</td>
<td>11.9389</td>
<td>16.536675</td>
<td>78.25</td>
</tr>
<tr>
<td>NT_032977.9</td>
<td>909</td>
<td>0.1526</td>
<td>0.312470</td>
<td>11.9427</td>
<td>16.829435</td>
<td>78.25</td>
</tr>
<tr>
<td>NT_005612.16</td>
<td>1005</td>
<td>0.1540</td>
<td>0.347171</td>
<td>12.0517</td>
<td>18.798967</td>
<td>78.25</td>
</tr>
<tr>
<td>NT_016354.19</td>
<td>1005</td>
<td>0.1551</td>
<td>0.401382</td>
<td>12.1367</td>
<td>21.750664</td>
<td>78.25</td>
</tr>
</tbody>
</table>

Table 5: The execution time (in seconds) of the Naïve algorithm for searching circular random patterns in a text.

<table>
<thead>
<tr>
<th>(m)</th>
<th>Naïve (C)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>0.1280</td>
</tr>
<tr>
<td>4</td>
<td>0.2568</td>
</tr>
<tr>
<td>8</td>
<td>0.3844</td>
</tr>
<tr>
<td>16</td>
<td>0.5126</td>
</tr>
<tr>
<td>32</td>
<td>0.6410</td>
</tr>
<tr>
<td>64</td>
<td>0.7691</td>
</tr>
<tr>
<td>128</td>
<td>0.8973</td>
</tr>
<tr>
<td>256</td>
<td>1.0268</td>
</tr>
<tr>
<td>512</td>
<td>1.1542</td>
</tr>
<tr>
<td>1024</td>
<td>1.2827</td>
</tr>
</tbody>
</table>


