Computer Science • 22(2) 2021 https://doi.org/10.7494/csci.2021.22.2.3401

Vinod Prasad

CHARACTER FREQUENCY-BASED APPROACH FOR SEARCHING FOR SUBSTRINGS OF CIRCULAR PATTERNS AND THEIR CONJUGATES IN ONLINE TEXT

Abstract *A fundamental problem in computational biology is dealing with circular patterns. The problem consists of finding a pattern and its rotations in a database. In this paper, we present two online algorithms. The first algorithm reports all of the substrings (factors) of a given pattern in an online text. Then, without losing efficiency, we extend the algorithm to process the circular rotations of the pattern. For a given pattern P of size M and a text T of size N, the extended algorithm reports all of the locations in the text where a substring of P_c is found where P_c is one of the rotations of P. For an alphabet size σ using $O(M)$ space, the desired goals are achieved in an average $O(MN/\sigma)$ time, which is $O(N)$ for all patterns with length $M \leq \sigma$. Traditional string-processing algorithms make use of advanced data structures such as suffix trees and automaton. The experimental results we have provided show that basic data structures such as arrays can be used in text-processing algorithms without compromising efficiency.*

Keywords pattern matching in strings, circular pattern matching, similarity search, substring search

Citation Computer Science 22(2) 2021: 235–250

1. Introduction

A fundamental problem in computer science is searching for a pattern in a text. A slightly different problem ('circular-pattern-matching', or CPM) deals with circular patterns. This problem involves dealing with multiple patterns that are the rotations of a given pattern simultaneously. CPM is crucial in the context of many biological and computational geometry-related problems. The DNA of some viruses, bacteria, eukaryote, and plants is circular in shape. Circular-shaped DNA makes a potential search process highly complex. Traditional string-matching algorithms work well for linear patterns, but they are not well-suited for processing the ring-shaped DNA of these organisms. A circular pattern is best represented by a cyclic array in which the last and first indexes are adjacent to each other; i.e., for a pattern of size M, the first index 1 is followed by the last index M. Notice that we assume the initial index of the pattern to be 1 instead of 0. For example, for a circular pattern $P = ABBAAB$, there are six different rotations or conjugates: $P_1 = ABBAAB, P_2 = BBAABA, P_3 = BAABAB, P_4 = AABABB, P_5 = ABABBA$, and $P_6 = BABBAA$. Each rotation P_c of length M that starts at index c in P called a conjugate of P (where $1 \leq c \leq M$). Notice that two or more rotations of a pattern can be the same depending on the character-frequencies of the individual characters in the pattern.

2. Related work

Most of the string-processing algorithms discovered so far are covered in [1, 6, 10], and [14]. These algorithms can also be used to process circular patterns by applying some modifications. A substring or a factor of pattern P is nothing but a prefix of some of the suffixes of P. Therefore, 'suffix tree' and 'suffix automaton' have been widely used in the literature to solve pattern-matching problems [3, 10, 15], and [17]. A 'suffix tree' is a tree that contains all of the suffixes of a text. Most of the suffix tree-based algorithms maintain an array of pointers of size σ (the size of the alphabet), which may be an issue when the size of the alphabet is large. The suffix tree-based approach presented in [12] is comprised of building suffix trees for text T and $T^{'}$, where $T^{'}$ is obtained by reversing text T. Another suffix tree-based algorithm [9] consists of creating a generalized cyclic suffix tree for all of the rotations of a given set of sequences and then applying the search in $O(N)$ time. Another widely used data structure 'suffix automaton' simulates the smallest deterministic finite automaton that recognizes all of the suffixes of a given pattern [3, 6]. Therefore, suffix automatons are the best-suited data structures to recognize all of the substrings of a pattern in a text. The algorithm given in [14] is based on the simple fact that any rotation of a pattern P is a substring of PP. Therefore, a suffix automaton for PP is capable of searching for all of the rotations of P in text T in $O(N)$ time. Bit-parallel algorithms such as [4, 5], and [11] gained momentum following [16] in which fast bit-wise operations were used to speed up the search process. Note that [16] was designed to process those non-circular patterns that were later on extended to processing the circular patterns [4]

and [11]. The algorithms presented in [4] and [11] run in a worst-case $O(NM^2)$ time, whereas [5] requires average-case time $O(Nlog\sigma M/W)$, where W is the word size of the machine. The algorithm presented in [13] solves the CPM problem in $O(Nlog(\sigma))$ time and $O(N)$ space. The algorithm presented in [2] is based on the concatenation of a pattern with itself, followed by partitioning the string into fragments, and then processing each fragment individually.

The state-of-the-art solutions presented above require some serious efforts in the preprocessing phase to implement and maintain the complex data structures. These solutions are fantastic, but they may merely be overkill when a database is small and the pattern only needs to be searched once over the text. The algorithms we propose in Sections 7 and 8 simulate a suffix automaton for linear and circular patterns without actually creating it.

3. Assumptions and notations

Throughout the paper, we use symbol T for text and P for pattern. Both T and P are non-empty finite sequences of symbols that are drawn from alphabet λ of size σ. We use symbol N and M to denote the size of a text and a pattern, respectively. As most of the practical situations demand, we assume both T and P to be non-empty such that $N \geq M$. We use integers i and j to represent the indexes over T and P, respectively. The term 'index', 'shift', or 'location' represent the location of a character in a text or pattern from the beginning. We assume the initial index of the text and the pattern to be 1, which is slightly different from the traditional notation where 0 is assumed to be the initial index. So, in our case, indexes i and j are such that $1 \leq i \leq N$ and $1 \leq j \leq M$. $T[i]$ or T_i represent the i^{th} character of T, and $P[j]$ or P_j represent the j^{th} character of P. A 'substring', 'factor', or 'sub-word' of P is a string that is contained in P. For a pattern P with all distinct characters, there are $(1 + \frac{P(P+1)}{2})$ different substrings; e.g., for $P = ABCD$, the set of all substrings (factors) is $\{' ', A, B, C, D, AB, BC, CD, ABC, BCD, ABCD\}$.

4. Substring search

Consider the simple problem of searching for the word 'branch' using the Google search engine. As we type the first three letters ('bra'), the browser predicts the remaining letters. This is good enough, but try inducing a typo by replacing the first letter 'b' by some other letter; say, 'a'. Now, when typing 'ara', 'aran', 'aranc', or 'aranch', the text predictor fails miserably. In the first case, the input of just three letters was enough to predict the whole word. In the second case, however, the initial wrong input followed by five correct letters could not induce the text predictor to suggest the word 'branch'. So much is the dependence of the text predictor on the first letter that it was unable to map the word 'aranch' to 'branch' even while matching five of the six letters correctly; it completely ignored the match of the last five letters.

In this paper, we adopt a different form of 'similarity' that is based on the length of the substring of a pattern. Consider a pattern $P = ABCD$ that has ten different non-empty substrings {A, B, C, D, AB, BC, CD, ABC, BCD, ABCD}. Furthermore, let us consider all of the conjugates of P such that $P_1 = ABCD$, $P_2 = BCDA$, $P_3 = CDAB$, and $P_4 = DABC$. Now, the set of substrings grows even further {A, B, C, D, AB, BC, CD, DA, ABC, BCD, CDA, DAB, ABCD, BCDA, CDAB, DABC}. From these sets, it is clear that the larger substrings of P have a greater similarity to P or its conjugates. The algorithm discussed in Sections 7 and 8 report all of the substrings of P or their conjugates in online text stream T. Needless to say, the extended version of the algorithm works for both circular or non-circular patterns. The problem we solve can be defined as follows:

Let $T[1 \dots N]$ be a text stream of size N and $P[1 \dots M]$ be a pattern of size M such that $1 \leq M \leq N$. Let P_c represent the c^{th} rotation (conjugate) of P such that $1 \leq c \leq M$. Then, for a given integer $k(1 \leq k \leq M)$, report all of the i locations in $T(1 \leq i \leq N)$ where a substring (factor) of P_c length $\geq k$ is found.

The algorithms presented in this paper are completely different from the solutions that were developed in the past in the following aspects. First, our 'model of similarity' is based on the 'length of the substring' – a larger substring means a greater similarity. Had this been the case with the Google search engine, the text predictor would not have missed the word 'branch', as we have seen this before. Second, the proposed algorithms are online in nature; i.e., the text characters are fed to the algorithm one-by-one, and the substrings are reported as they are encountered in the text. Third, the algorithms do not make use of any complex data structure, yet remain competitive in most of the cases, which is evident from the the experimental results given in Section 10. The rest of the paper is organized as follows. In the section that follows, we provide a pattern-preprocessing algorithm that was also given in our previous work [18]. The theoretical base for the proposed algorithms is provided in Section 6. In Section 7, we provide our first algorithm that reports all of the substrings of a non-circular pattern in a text. The extended version given in Section 8 enables the algorithm to process the circular patters. In Section 9, we provide a detailed discussion on the time-space requirement of the algorithms. And finally, in Section 10, we perform experiments to compare the algorithm given in Section 7 with the state--of-the-art solutions.

5. Pattern preprocessing

To process the pattern, an array of pointers 'shift[max_ASCII+1]' is used as a hash table, where max_ASCII is the maximum possible ASCII code of a character in the alphabet (which is typically 127 or 255). Each pointer in an array points to a linked list that stores all of the 'shifts' of a character in the pattern. Initially, all of the linked lists are empty. The pattern is read from left to right. Based on the ASCII value of the character, its shift is stored in the corresponding linked list. Let us consider pattern $P = ABBAAB$. For the first character 'A' (which is at shift 1), a node that

contains the integer 1 is inserted in the link list at shift[65], where 65 is the ASCII code of the letter 'A'. Similarly for the next character 'B', a node that contains the the integer '2' is inserted in the link list beginning at 66. Figure 1 shows the algorithm and the resulting shift array. Notice that, in the shift array, the nodes are inserted in the beginning of the list, which creates the list in reverse order.

Pattern Preprocessing Algorithm: Input: pattern characters. Output: set of σ (alphabet size) linked lists. Each linked list stores shifts of a character in the pattern in reverse order.

```
'node' is a structure with two fields: integer s, and node type pointer *next
node *shift[max_ASCII+1]                    /* max_ASCII 127 or 255 */
integer i, j, M = 1                         /* M=1, initial shift of the pattern */

for  i = 0 to max_ASCII do                  /* initialize the shift array */
        shift[ i ] = NULL
end for

while ( Not end of the pattern )  do        /* read pattern */
        j = ASCII code of the pattern character
        node *ptr = new node
        ptr → s = M++
        ptr→ next = shift [j]
        shift[j] = ptr
end while
```

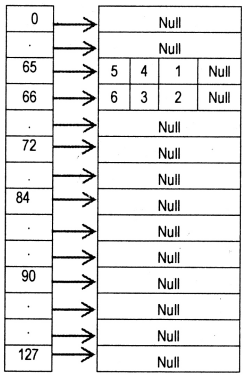


Figure 1. Pattern-preprocessing algorithm and output shift array

6. The 'i-j chain' lemma

Let $T[1,\ldots,N]$ be a text array of size N, and let $P[1,\ldots,M]$ be a pattern array of size M. Let i and j represent shifts in T and P, respectively, such that $1 \leq i \leq N$ and $1 \leq j \leq M$. Now, for each shift j in P, we define a set R_j such that $R_j = \{i \mid T[i] = P[j] \ \forall\ i's\ in\ T\}$. Now, consider sets $R_1, R_2 \ldots R_M$ as defined above. Let there be three integers (i, j, and k) such that $i \in R_j, (i+1) \in R_{j+1}, (i+2) \in R_{j+2} \ldots (i+k-1) \in R_{j+k-1}$. Then, we refer to sequence $i \in R_j, (i+1) \in R_{j+1}, (i+2) \in R_{j+2} \ldots (i+k-1) \in R_{j+k-1}$ as an 'i-j chain' of length k. Each 'i-j chain' of length k represents a k-length substring of P that is available at shift i in T.

Proof. Let $i \in R_j \rightarrow T[i] = P[j]$; i.e., each $i \in R_j$ represents a single character match of the i^{th} character of T with the j^{th} character of P, which is simply a substring of length 1 of P in T. Similarly, $i \in R_j$ and $(i+1) \in R_{j+1}$ represent a match of the i^{th} character of T with the j^{th} character of P as well as the $(i+1)^{th}$ character of T with the $(j+1)^{th}$ character of P, which is a substring of length 2, and so on. Therefore, it follows that $i \in R_j, (i+1) \in R_{j+1}, \ldots (i+k-1) \in R_{j+k-1}$ represents a substring of length k of P in T.

Observation 1. For some $P[j]$, if none of the text characters are $T[i] = P[j]$, then R_j would be empty; i.e., $R_j = \{\}$. In such a case, a chain of length M (i.e., a substring of length M) would not be possible. In other words, an exact match of P is not available in T.

Example 1. Let P be a pattern of size $M = 6$ such that $P = ABBAAB$. Let T be the text of size $N = 25$ such that $T = BAAABABBBBAABABBAABAABABB$.

In this case, we have $1 \leq i \leq 25$ and $1 \leq j \leq 6$. The 'i-j chain lemma' given in the preceding section yields the following six sets:

$R_1 = \{2, 3, 4, 6, 11, 12, 14, 17, 18, 20, 21, 23\}$;
$R_2 = \{1, 5, 7, 8, 9, 10, 13, 15, 16, 19, 22, 24, 25\}$;
$R_3 = \{1, 5, 7, 8, 9, 10, 13, 15, 16, 19, 22, 24, 25\}$;
$R_4 = \{2, 3, 4, 6, 11, 12, 14, 17, 18, 20, 21, 23\}$;
$R_5 = \{2, 3, 4, 6, 11, 12, 14, 17, 18, 20, 21, 23\}$;
$R_6 = \{1, 5, 7, 8, 9, 10, 13, 15, 16, 19, 22, 24, 25\}$.

For the given pattern and text, Algorithm 1 (given in the next section) is simulated in Figure 2. The sets given above are drawn vertically in Figure 2, which shows the online construction of the 'i-j chains'. The first column of Figure 2 shows the input text character with its shift i in the text. The next six columns represent sets $R_1, \ldots, R_6$. For each text character, some of the sets in the figure grow top-down by one element. For example, for the first input text character $T[1] = B$, the corresponding shifts in the pattern are 6, 3, and 2 (refer to Figure 1). Consequently, for the first element $i = 1$, we add 1 to sets R_6, R_3, and R_2. We call it indexes 6, 3, and 2 being hit (stamped) by timestamp $i = 1$. For the next input text character $T[2] = A$ (which is at shifts 5, 4, and 1 in the pattern), we stamp R_5, R_4, and R_1 with current timestamp $i = 2$. In the next step, the cell being hit is connected with the immediate upper-left cell if it has an immediate preceding timestamp $(i - 1)$. This forms a chain that we call the 'i-j chain'. Each 'i-j chain' in the figure represent a substring or factor of P. With respect to each incoming text character, the largest chain is chosen to show the length of the substring in the last two columns.

7. Substring search algorithm: linear patterns

In Figure 2, a table is used to create the 'i-j chains'. However, for large values of N, a table of size MN would be impractical. However, Algorithm 1 next uses a single one-dimensional array 'hit[]' of size $M + 1$ to achieve same goals. Notice that, in this case, the each upper-left cell to a cell in Figure 2 would become the immediate left cell in the one-dimensional array. Each cell of the hit[] array maintains two integer fields: 'len', and 'ts'. Field 'len' records the length of the 'i-j chain', and field 'ts' records timestamp i. We call the array 'hit[]' because each incoming text character hits some of the array cells with current timestamp i.

Observation 2. While stamping the cell hit[t] of the array, the algorithm inspects the left neighbor hit[t-1] to form a possible chain. This is why the array is stamped from right to left to ensure that the data of a cell is not updated before it is read. This is precisely why the linked lists in Figure 1 are created in reverse order.

Observation 3. For cell hit[0], there is no immediate left neighbor; so, what will happen if the algorithm hits at hit[0]? This possibility is eliminated by assuming the initial pattern index to be 1; i.e., hit[0] never gets stamped by the algorithm.

Algorithm 1 Hit-index substring search (linear patterns):

Input: Pattern in the form of a shift array as shown in Figure 1, an integer k ($1 \leq k \leq M$), where k is the minimum length of the substring to be searched, and the online text stream.

Output: All locations in the text stream where a substring of $length \geq k$ of the pattern is found. The output is shown as each text character is received.

Require: Let 'node' be structure with two fields: integer s, and node type pointer *next
Let 'cell' be structure with two fields: integer len, and integer ts

```
integer i = 0, j, t, k, M, maxlen
array cell hit[ M+1 ]

for j = 0 to M do
   hit[ j ].ts ← -1                                   /*Initialize array timestamp */
end for

while (Not end of text) do
   j ← ASCII value of input text character
   node *ptr ← shift[j]                               /*jump to jth link list in shift array*/
   maxlen ← 0, i ← i+1

   while (ptr! = null) do
      t ← (ptr→s)
      hit[ t ].ts ← i                          /* stamp index t with current timestamp i */
      if (hit[t − 1].ts == (i − 1)) then
         hit[ t ].len ← hit[ t-1 ].len + 1                /*increase chain-length */
      else
         hit[ t ].len ← 1                                 /*reset chain-length to 1*/
      end if
      if (hit[t].len > maxlen) then
         maxlen←hit[ t ].len                              /*keep track of largest chain */
      end if
      ptr ←(ptr →next)
   end while

   if (maxlen ≥ k) then
      Print Substring length and Location: maxlen, i-maxlen + 1
   end if
end while
```

Row (i), T[i]	R_1	R_2	R_3	R_4	R_5	R_6	Max Length	Common Substring(T,P)
1 B		1	1			1	1	B
2 A	2			2	2		2	BA
3 A	3			3	3		3	BAA
4 A	4			4	4		2	AA
5 B		5	5			5	3	AAB
6 A	6			6	6		2	BA
7 B		7	7			7	2	AB
8 B		8	8			8	3	ABB
9 B		9	9			9	2	BB
10 B		10	10			10	2	BB
11 A	11			11	11		3	BBA
12 A	12			12	12		4	BBAA
13 B		13	13			13	5	BBAAB
14 A	14			14	14		2	BA
15 B		15	15			15	2	AB
16 B		16	16			16	3	ABB
17 A	17			17	17		4	ABBA
18 A	18			18	18		5	ABBAA
19 B		19	19			19	6	**ABBAAB**
20 A	20			20	20		2	BA
21 A	21			21	21		3	BAA
22 B		22	22			22	4	BAAB
23 A	23			23	23		2	BA
24 B		24	24			24	2	AB
25 B		25	25			25	3	ABB

Figure 2. Algorithm 1 output for $P = ABBAAB$. Last column shows longest substring of P that is found in T

8. Substring search algorithm: circular patterns

The 'i-j chain' lemma given in Section 6 can be extended for circular patterns by assuming set R_1 to be the successor of R_M. This means the two consecutive integers in the last set R_M and the first set R_1 can be connected to form a circular chain (refer to Figure 3). Algorithm 2 given next (the extended version of Algorithm 1) does exactly this. The corresponding process is shown in Figure 3. Therefore, in Figure 3, the last cell of hit[] array in the immediate upper row becomes the immediate left neighbor of the first cell; the obtained chain is called the 'circular i-j chain'.

Observation 4. As shown in Figure 3, a circular chain is created when the algorithm hits the first cell and then inspects the upper last cell hit[M] for a possible chain. However, while using the one-dimensional array, the data in hit[M] is overwritten by the time we reach the first cell hit[1] (as we are moving from right to left). Therefore, at the end of the inner while-loop of the Algorithm 2, hit[M] is copied into unused cell hit[0] to preserve the data. This further enables us to create a circular chain in the same manner as with the one that creates a linear chain; i.e., when the algorithm

hits at hit[1], the left neighbor hit[0] is inspected rather than the last cell hit[M]. Note that neither Figures 2 nor 3 show Column 0.

Algorithm 2 Hit-index substring search (circular patterns):

Input: Pattern in the form of a shift array as shown in Figure 1, an integer k($1 \leq k \leq M$), where k is the minimum length of the substring to be searched, and the online text stream.

Output: all locations in the text where a substring $length \geq k$ of P_c is found, where P_c is one of the conjugates of P such that $1 \leq c \leq M$. The output is shown as each text character is received.

Require: Let 'node' be structure with two fields: integer s, and node type pointer *next
Let 'cell' be structure with two fields: integer len, and integer ts

```
integer i = 0, j, t, k, M, maxlen
array cell hit[ M+1 ]

for j = 0 to M do
   hit[ j ].ts ← -1                                   /* Initialize array timestamp */
end for
while (Not end of text) do
   j ← ASCII value of input text character
   node *ptr ← shift[j]                               /*jump to jth link list in the shift array*/
   maxlen ← 0, i ← i+1
   while (ptr! = null) do
      t ← (ptr→s)
      hit[ t ].ts ← i
      if (hit[t − 1].ts == (i − 1)) then
         hit[ t ].len ← hit[ t-1 ].len + 1            /*increase chain-length */
      else
         hit[ t ].len ← 1                             /*reset chain-length to 1*/
      end if
      if (hit[t].len > maxlen) then
         maxlen←hit[ t ].len                          /*Keep track of largest chain*/
      end if
      ptr ←(ptr →next)
   end while
   hit[ 0 ].ts← hit[ M ].ts                           /*copy last cell's data into unused cell 0*/
   hit[ 0 ].len←hit[ M ].len
   if (maxlen > M) then
      maxlen ← M                                      /*reset chain-length to M*/
   end if
   if (maxlen ≥ k) then
      Print Substring length and Location: maxlen, i-maxlen + 1
   end if
end while
```

Row (i), T[i]	R_1	R_2	R_3	R_4	R_5	R_6	Max Length	Common Substring(T, P)
1 B		1	1			1	1	B
2 A	2			2	2		2	BA
3 A	3			3	3		3	BAA
4 A	4			4	4		2	AA
5 B		5	5			5	3	AAB
6 A	6			6	6		4	AABA
7 B		7	7			7	5	AABAB
8 B		8	8			8	6	**AABABB=P_4**
9 B		9	9			9	2	BB
10 B		10	10			10	2	BB
11 A	11			11	11		3	BBA
12 A	12			12	12		4	BBAA
13 B		13	13			13	5	BBAAB
14 A	14			14	14		6	**BBAABA=P_2**
15 B		15	15			15	6	**BAABAB=P_3**
16 B		16	16			16	6	**AABABB=P_4**
17 A	17			17	17		6	**ABABBA=P_5**
18 A	18			18	18		6	**BABBAA=P_6**
19 B		19	19			19	6	**ABBAAB=P_1**
20 A	20			20	20		6	**BBAABA=P_2**
21 A	21			21	21		3	BAA
22 B		22	22			22	4	BAAB
23 A	23			23	23		5	BAABA
24 B		24	24			24	6	**BAABAB=P_3**
25 B		25	25			25	6	**AABABB=P_4**

Figure 3. Algorithm 2 output for $P = ABBAAB$. Last column shows longest substring ($\leq M$) of P or one of its conjugates found in T

Observation 5. The maximum possible length of a pattern substring is M. However, a circular chain can grow beyond the size of the pattern. Consider text $9B$ through $20A$ in Figure 3. Here, the pattern conjugates appear at successive text locations, which results in the chain length growing to 12. Therefore, Algorithm 2 resets the chain length to M whenever it grows beyond M, and it considers the last chain segment of $length = M$ to report the substring. The example given below clarifies this.

Example 2. In Figure 3, the initial set number reveals the identity of the conjugate. For example, the first link of chain-segment '11-12-13-14-15-16' lies within set R_4, which means that the discovered substring $AABABB$ is P_4, which is available at $T[11]$. Now, consider circular chain '9-10-11-12-13-14-15-16-17-18-19-20'. The length of the chain is 12, and the maximum possible length of a substring is $M = 6$. Therefore, the first chain segment '9-10-11-12-13-14' with a length of 6 represents substring $BBAABA$, which is the conjugate P_2 that is available at $T[9]$. Skipping the first link, the next segment of a length of 6 (10-11-12-13-14-15) represents the next conjugate

$P_3 = BAABAB$ at location $T[10]$. Continuing this pattern with the same chain, we discover $P_4 = AABABB$, $P_5 = ABABBA$, $P_6 = BABBAA$, $P_1 = ABBAAB$, and again $P_2 = BBAABA$ at successive text locations: $T[11]$, $T[12]$, $T[13]$, $T[14]$, and $T[15]$, respectively. All of the remaining chains of $length < M$ represent smaller substrings of P or its conjugates.

9. Time and space analysis

The pattern-preprocessing phase consists of creating a 'shift-array' (Fig. 1) of size σ in which a total of M nodes are inserted. Inserting a node at the beginning of the list takes $O(1)$ time. Thus, the pattern-preprocessing phase consumes $O(\sigma + M)$ space and $O(M)$ time. Both of the search algorithms require an additional array `hit[M+1]` of size $(M + 1)$ in the search phase. Hence, the total run-time memory requirement of the algorithms is $O(\sigma + M)$. Thus, the total run-time memory requirement of the algorithms is independent of N, which is ideal for large applications.

Let us discuss the execution time of the proposed algorithms in the search phase. The extended version of Algorithm 2 for circular patterns continued to have the same time and space complexity as with Algorithm 1. Therefore, the discussion that follows is applicable to both of the search algorithms given in Sections 7 and 8.

Let λ_i be the i^{th} character in alphabet λ such that $1 \leq i \leq \sigma$. Furthermore, let ft_i and fp_i represent the frequencies of λ_i in the text and pattern, respectively. These frequencies can be expressed as follows:

$$ft_1 + ft_2 + \ldots ft_\sigma = \sum_{i=1}^{\sigma} ft_i = N \ldots \tag{1}$$

$$fp_1 + fp_2 + \ldots fp_\sigma = \sum_{i=1}^{\sigma} fp_i = M \ldots \tag{2}$$

In the search phase, the algorithms read a text character and then jump to the 'shift table' to retrieve the shifts of the character in the pattern. For each shift 't', both of the algorithms stamp the array hit[t] with the current timestamp 'i'. Therefore, the execution time in the search phase is the sum of the time required to read all of the N characters and the time consumed in stamping the array cells (which we will call '*hits*'). As a rough estimation, this can be put as follows:

$$Execution\ Time = N + Total\ number\ of\ hits\ induced \ldots \tag{3}$$

Using the frequencies given above, we can precisely compute the total number of hits that are induced in the search process. For each input text character λ_i, the algorithm retrieves fp_i entries from the shift array. Because ft_i represents the frequency of λ_i in the text, the total number of hits is given by the following:

$$Total\ hits\ induced = ft_1 fp_1 + ft_2 fp_2 \ldots + ft_\sigma fp_\sigma = \sum_{i=1}^{\sigma} ft_i fp_i \ldots \tag{4}$$

Hence, using (3) and (4), the total execution Time:

$$total\ execution\ Time = N + \sum_{i=1}^{\sigma} ft_i fp_i \ldots \tag{5}$$

Let us now discuss a few particular cases.

Case 1. Consider a pattern of size M such that $M \leq \sigma$, and all of the characters in the pattern are distinct; i.e., each fp_i is either 0 or 1. *Hence, the total number of hits* $= \sum_{i=1}^{\sigma} = ft_i fp_i = ft_1 fp_1 + ft_2 fp_2 \ldots + ft_\sigma fp_\sigma \leq N$ using (1). And from (5), *the total execution time* $= N + \sum_{i=1}^{\sigma} ft_i fp_i \leq 2N$, which is $O(N)$.

Case 2. For a pattern of length M such that no character in the pattern is present in the text; i.e., for each $\lambda_i \in P, ft_i = 0$. *Then, the total hits* $= \sum_{i=1}^{\sigma} ft_i fp_i = 0$, and from (5), the total execution time $= N \approx O(N)$.

Case 3. Consider a pattern that consists of M repetitions of a single character λ_i; i.e., all of the character-frequencies in the pattern are 0 except for $fp_i = M$. In this case, all of the products in Equation (4) are equal to 0 except for $ft_i fp_i$. *Thus, the total hits* $= \sum_{i=1}^{\sigma} ft_i fp_i = M ft_i$, and using (5), *The total execution time* $= N + M ft_i$, which is $O(N + M ft_i)$.

Case 4. Consider a rare situation when the entire text and pattern are made of a single character λ_i. This means that a single character λ_i is repeated M and N times in P and T, respectively; i.e., all of the character frequencies in both P and T are 0 except for $fp_i = M$ and $ft_i = N$. Then, from (4), $\sum_{i=1}^{\sigma} ft_i fp_i = MN \approx O(MN)$.

Case 5. Let us consider the average case. In the average case, the expected frequencies ft_i and fp_i of a character λ_i in the text and pattern are given by N/σ and M/σ, respectively (where σ is the size of the alphabet). Therefore, from (4), *The expected number of hits* $= ft_1 fp_1 + ft_2 fp_2 + \ldots ft_\sigma fp_\sigma = [(N/\sigma)(M/\sigma)]\sigma = MN/\sigma$. And from (5), *The total execution time* $= N + MN/\sigma \approx O(MN/\sigma)$. Therefore, for a pattern $M \leq \sigma$, *The expected execution time is* $\leq 2N \approx O(N)$.

Observation 6. While stamping the hit array, the character 'A' of the text is merely compared to the same character 'A' of the pattern, which results in a better efficiency of the algorithm.

10. Experimental results

In this section, we perform experiments to compare Algorithm 1 (given in Section 7) with one of the state-of-the-art solutions that is based on the suffix automaton [3, 6]. The suffix automaton is a data structure that simulates a DFA that is capable of recognizing all of the suffixes of a pattern. For a given pattern P, a prefix of a suffix of P is a substring of P. This means that a suffix automaton that recognizes all of the suffixes of P automatically recognizes all of the substrings of P. For example, consider pattern $P =$ '*banana*'. The suffix set is '*a*', '*na*', '*ana*', '*nana*', '*anana*', and '*banana*' (we are not considering an empty string). The prefix of any of these suffixes

is a substring of P. For example, we have the following prefixes for suffix '*anana*': '*a*', '*an*', '*ana*', '*anana*'. All of these are valid substrings of P. We now briefly explain the experimental setup as well as the hardware, software, and test data that is used to perform our experiments.

Hardware & Software: We performed our experiments on an ASUS laptop with an $Intel\ Core\ i7\ CPU@2.5GHz,\ 8GB\ RAM,\ 64\ bit$ Windows 10 Operating System x64-Based processor. To compile our programs, we used $MinGW$ – a native Windows port of the GNU Compiler Collection (GCC) $g++$ compiler.

Suffix Automaton: We used the forward directed acyclic word graph ($DAWG$) matching algorithm given in [6] to create the suffix automaton. The 'C' language version of the algorithm was downloaded from [7].

Test Data: We downloaded three text files from the 'string matching algorithm research tool' ($SMART\ tool$) [8]. A brief description of these files is given below.

'*hs.txt*': The protein sequence database consists of repeated sequences of 20 letters $(A, C, D, E, F, G, H, I, K, L, M, N, P, Q, R, S, T, V, W, Y)$, which represent amino acids.

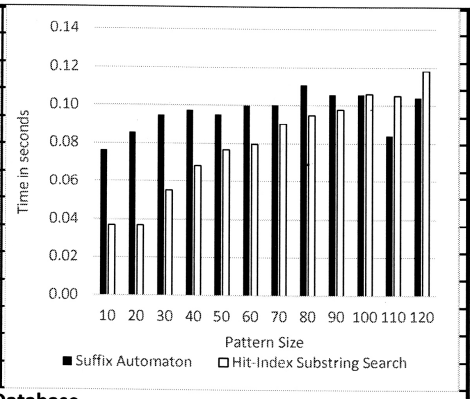
'*ecoli.txt*': The DNA database consists of repeated sequences of four letters (A, C, G, and T), which represent the four nucleotide bases of a DNA strand.

'*bible.txt*': Natural English language text characters ($[a-z][A-Z]$) with spaces and punctuation.

Experimental Setup: In our experiments, we first stored the text and pattern characters in arrays before applying the search. This was done to avoid reading from the file during the search process, as the reading time from the files may vary for different program runs. We used an array of a size of two million to read the first two million characters of the biological databases into the array. However, for 'bible.txt' (which is small in size), the entire file of 30,382 characters was stored in the array. Next, three different patterns of a particular size were chosen from the text file. Then, a search was applied in the same file to record the search time for each pattern. Following this, the mean search time was calculated. For example, we first read 20 million characters of protein database 'hs.txt' in an array. Then, three different samples of patterns of the same size 10 (P1, P2, P3) were randomly chosen from the same file, and three program runs were applied to search each sample using the respective algorithms to record the search time. Following this, the mean search time for P1, P2, and P3 was calculated (refer to Figures 4a, 4b, and 4c). A similar process was adopted for the remaining patterns of sizes from 20 to 120.

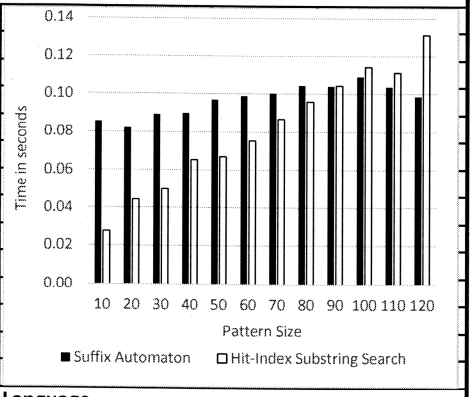
The corresponding graphs show a comparison between the mean run-times taken by the algorithms for each pattern size (in seconds). Please note that the execution time does not include the preprocessing time nor the time consumed in reading the characters from the file in the array.

	Suffix Automaton				Hit Index Sub. Search			
M	P1	P2	P3	Mean	P1	P2	P3	Mean
10	0.078	0.08	0.071	0.076	0.047	0.032	0.032	0.037
20	0.088	0.089	0.08	0.086	0.032	0.04	0.039	0.037
30	0.109	0.08	0.096	0.095	0.078	0.048	0.04	0.055
40	0.109	0.096	0.088	0.098	0.093	0.056	0.056	0.068
50	0.11	0.096	0.08	0.095	0.094	0.072	0.064	0.077
60	0.109	0.096	0.095	0.100	0.094	0.072	0.073	0.080
70	0.109	0.096	0.096	0.100	0.109	0.089	0.073	0.090
80	0.125	0.104	0.103	0.111	0.109	0.088	0.088	0.095
90	0.125	0.096	0.096	0.106	0.11	0.088	0.096	0.098
100	0.109	0.104	0.104	0.106	0.109	0.105	0.104	0.106
110	0.104	0.087	0.062	0.084	0.112	0.109	0.094	0.105
120	0.104	0.104	0.104	0.104	0.12	0.121	0.112	0.118



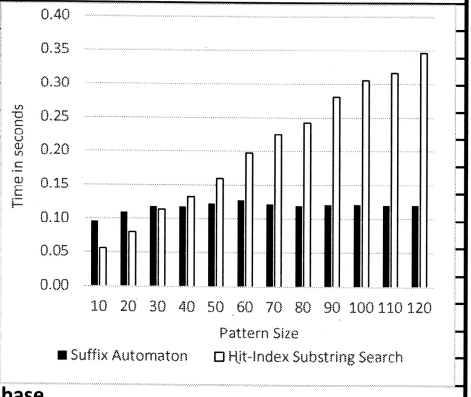
a: Protein Database

	Suffix Automaton				Hit Index Sub. Search			
M	P1	P2	P3	Mean	P1	P2	P3	Mean
10	0.088	0.081	0.088	0.086	0.027	0.024	0.033	0.028
20	0.083	0.083	0.081	0.082	0.043	0.048	0.042	0.044
30	0.092	0.088	0.088	0.089	0.050	0.050	0.049	0.050
40	0.093	0.089	0.088	0.090	0.059	0.065	0.072	0.065
50	0.096	0.098	0.096	0.097	0.065	0.072	0.064	0.067
60	0.098	0.098	0.101	0.099	0.076	0.075	0.075	0.075
70	0.099	0.100	0.102	0.100	0.088	0.086	0.087	0.087
80	0.105	0.100	0.108	0.104	0.096	0.096	0.096	0.096
90	0.105	0.103	0.104	0.104	0.104	0.105	0.104	0.104
100	0.104	0.118	0.105	0.109	0.110	0.124	0.109	0.114
110	0.104	0.110	0.097	0.104	0.113	0.109	0.112	0.111
120	0.096	0.104	0.096	0.099	0.128	0.137	0.128	0.131



b: Natural Language

	Suffix Automaton				Hit Index Sub. Search			
M	P1	P2	P3	Mean	P1	P2	P3	Mean
10	0.096	0.096	0.097	0.096	0.056	0.056	0.055	0.056
20	0.104	0.112	0.112	0.109	0.080	0.080	0.081	0.080
30	0.121	0.112	0.120	0.118	0.111	0.112	0.119	0.114
40	0.120	0.120	0.113	0.118	0.127	0.135	0.135	0.132
50	0.120	0.127	0.120	0.122	0.159	0.161	0.160	0.160
60	0.143	0.119	0.120	0.127	0.192	0.192	0.209	0.198
70	0.122	0.121	0.122	0.122	0.217	0.219	0.238	0.225
80	0.120	0.118	0.120	0.119	0.246	0.240	0.242	0.243
90	0.121	0.120	0.121	0.121	0.276	0.296	0.272	0.281
100	0.121	0.121	0.122	0.121	0.319	0.302	0.299	0.307
110	0.120	0.120	0.120	0.120	0.320	0.328	0.304	0.317
120	0.121	0.120	0.120	0.120	0.336	0.334	0.369	0.346



c: Ecoli Database

Figure 4. Algorithm 1 vs Suffix Automaton

The graph given in Figure 4a shows that, for protein sequence database 'hs.txt', the 'Hit-index substring search' algorithm given in Section 7 performs better against the suffix automaton for those patterns with sizes of less than 100. Following

this, the suffix automaton starts to take over (less time-consuming). For natural languages such as 'bible.txt' a similar trend is shown in Figure 4b. Finally, Figure 4c shows that, for DNA databases such as 'ecoli.txt', our algorithm remains competitive until the pattern size reaches 30; beyond this, the suffix automaton takes the lead. Notice that the 'ecoli' DNA database is drawn from just 4 letters, the protein database is drawn from 20 letters, and the natural language database uses around 55 letters. From the given results, it is evident that the 'Hit-Index substring search' algorithm works well when the size of the pattern is less than 120 and the alphabet size is reasonably large (refer to Case 6 of the 'time space analysis' section). Even for those DNA databases with a small alphabet size of four (A, C, G, T), the algorithm remains competitive until the pattern size reaches 40. These results are in line with the theoretical 'time space analysis' discussion given in the preceding section.

11. Conclusion

The suffix automaton is a magical data structure when all of the substrings (factors) of a pattern need to be searched. However, this is one of the complex data structures that needs some serious effort to implement and maintain. The use of such data structures for a one-time search is a bit of overkill, as the implementation cost outweighs the benefits that they provide. In such situations, the proposed algorithms can be a good alternative; they provide easier solutions to output the substrings of a pattern as well as its conjugates without making a serious investment in the preprocessing phase. The algorithms that we have developed in this paper use simple data structures; therefore, they are easy to implement and yet remain competitive in most cases. We have shown that simple arrays can also prove to be tough competition for state-of-the-art data structures without compromising efficiency.

Acknowledgements

We thank the anonymous reviewers for their careful reading of our manuscript and their insightful comments and suggestions.

References

[1] Apostolico A., Gali Z.: *Pattern Matching Algorithms*, Oxford University Press, 1997.

[2] Barton C., Iliopoulos C., Pissis S.: Fast algorithms for approximate circular string matching, *Algorithms for Molecular Biology*, vol. 9, 2014.

[3] Blumer A., Blumer J., Haussler D., Ehrenfeucht A., Chen M.T., Seiferas J.: The smallest automation recognizing the subwords text, *Theoretical Computer Science*, vol. 40, pp. 31–55, 1985.

[4] Chen K., Huang G.S., Lee R.C.T.: Exact Circular Pattern Matching Using the BNDM Algorithm. In: *28th Workshop on Combinatorial Mathematics and Computation Theory*, pp. 152–161, 2011.

[5] Chen K., Huang G.S., Lee R.C.T.: Bit-Parallel Algorithms for Exact Circular String Matching, *Computer Journal*, vol. 57, pp. 731–743, 2014.

[6] Crochemore M., Rytter W.: *Jewels of Stringology: Text Algorithms*, World Scientific Press, Singapore, 2003.

[7] Faro S., Lecroq T.: Forward Dawg Matching algorithm, https://www-igm.univ-mlv.fr/~lecroq/string/fdm.html#SECTION00220, 2016.

[8] Faro S., Lecroq T.: A String Matching Algorithms Research Tool, https://smart-tool.github.io/smart, 2016.

[9] Fernandes F., Pereira L., Freitas A.: CSA: An efficient algorithm to improve circular DNA multiple alignment, *BMC Bioinformatics*, vol. 10, pp. 1–13, 2009.

[10] Gusfield D.: *Algorithms on Strings, Trees and Sequences. Computer Science and Computational Biology*, Cambridge University Press, 1999.

[11] Hirvola T., Tarhio J.: Approximate Online Matching of Circular Strings. In: *Proceedings of the 13th International Symposium on Experimental Algorithms*, pp. 315–325, 2014.

[12] Iliopoulos C.S., Rahman M.S.: Indexing Circular Patterns. In: *Proceedings of the 2nd International Conference on Algorithms and Computation*, pp. 46–57, 2008.

[13] Lin J., Adjeroh D.: All-Against-All Circular Pattern Matching, *Computer Journal*, vol. 55, pp. 897–906, 2012.

[14] Lothaire M.: *Applied Combinatorics on Words*, Cambridge University Press, 2005.

[15] McCreight E.: A Space-Economical Suffix Tree Construction Algorithm, *Journal of ACM*, vol. 23, pp. 262–272, 1976.

[16] Navarro G., Raffinot M.: Fast and flexible string matching by combining bit-parallelism and suffix automata, *ACM JEA*, vol. 5, 2000.

[17] Ukkonen E.: On-line construction of suffix trees, *Algorithmica*, vol. 14, pp. 249–260, 1995.

[18] Vinod P.: A Novel Algorithm for String Matching with Mismatches. In: *5th International Conference of Pattern Recognition Applications and Methods, Rome*, pp. 638–644, 2016.

Affiliations

Vinod Prasad
University of Technology and Applied Sciences, Department of IT, Sur, P.O. Box: 484, Zip Code: 411, Sultanate of Oman, GSM:+968-92171389, vinod.sur@cas.edu.om

Received: 12.08.2019
Revised: 09.09.2020
Accepted: 21.09.2020