Transactions on Engineering and Computing Sciences - Vol. 11, No. 4
Publication Date: August 25, 2023
DOI:10.14738/tecs.114.15065.
Prasad, V. (2023). A Real-Time Algorithm for Searching for Substrings of a Pattern in Online Text Stream: An Alternative to Suffix Trees and Finite Automaton. Transactions on Engineering and Computing Sciences, 11(4). 158-171.



A Real-Time Algorithm for Searching for Substrings of a Pattern in Online Text Stream: An Alternative to Suffix Trees and Finite Automaton

Vinod Prasad
UTAS Sur, Sultanate of Oman

ABSTRACT

In the field of similarity search, very little effort has been made to process the text without using suffix trees or automata. Most of the algorithms developed in the past use data structures and methods outlined above to accelerate the search process. However, their costly maintenance has always been a cause of concern. Let, T be a text of size N, and P be a pattern of size M. We present highly practical algorithms to process a real-time stream of text characters. The algorithm reports all substrings of pattern P in the text stream T as they appear in the stream. This is highly useful in detecting patterns in the data stream in real-time, for example, viruses and malware can be detected as they appear in the data stream. Additionally, the algorithm can be used to find the longest common substring(s) between two given strings. We achieved the stated goals without using any sophisticated data structure such as a suffix tree or finite automaton. For an alphabet size σ the algorithm runs in $O(M)$ space, and $O(MN/\sigma)$ expected time, which simplifies to $O(N)$ for $M \leq \sigma$. We also provide experimental results to support our claim.

Keywords: Approximate String Matching, Similarity Search, Substring Search.

INTRODUCTION

Most string processing algorithms developed before the 1990's such as those of: - Morris and Pratt [1], Aho and Corasick [2], Knuth, et al. [3], Boyer and Moore [4], Horspool [5], and Karp-Rabin [6] were designed to search for exact pattern matches in the text. However, with some modifications, they can also be used to find approximate matches. Computational biology, web searches, data mining, signal processing, and digital dictionaries are several areas in which similarity searches have become a topic of considerable interest. String manipulating algorithms have made extensive use of suffix trees, suffix arrays, and automata. The algorithms based on automata preprocess the pattern to create a finite state machine (FSM). A finite state machine is a mathematical model of computation comprising a finite number of states and transitions. Each transition determines the next state of the machine. Some of the automaton-based algorithms are Wu and Manber [20], and Wu et al. [21], Melichar [22]. Algorithms based on text filtering accelerate the search process by filtering the text for the higher similarity regions, some of them include Tarhio and Ukkonen [23], Chang and Lawler [24], Takaoka [25], and Yates and Navarro [26]. In a suffix-tree-based algorithm, the text is preprocessed in O(N) time to construct a tree that stores all suffixes of the text. Once the tree is constructed, most strings-related queries can be resolved in O(M) time. Because of their high space consumption, suffix trees have not been widely used in string processing applications despite their broad

acceptance in theory. Even the most space-efficient main-memory suffix-tree construction algorithm requires 45 GB of memory to store the 700 MB size of the human genome according to Kurtz [37]. In the literature, there is a scarcity of practical solutions that apply searches for pattern substrings in a database without using suffix trees, suffix arrays, automata, or dynamic programming. The methods discussed above provide faster search time but still suffer various space-time tradeoffs. Suffix trees are not practical for single searches; neither are they viable options when the alphabet size is large. Moreover, the real-time construction and maintenance of a suffix tree pose further challenges: Amir et al. [38], and Breslauer and Giuseppe [39]. To reduce the space overhead of the suffix trees Manber and Myers [40] introduced the Suffix arrays. A suffix array stores the starting positions of all suffixes of the text in sorted order. Compressed versions of the suffix tree or suffix array also exist; see Grossi and Vitter 2005[41], Sadakane [42], and Russo et al. [43], and Alimaki et al. [44]. Similar problems arise in automaton-based algorithms, particularly when the alphabet is considerably large. For a detailed survey of these algorithms refer to Crochemore et al. [45], Navarro [46], and Boytsov [47]. Breaking the trend outlined above, in this paper, we follow a novel approach to process an unlimited amount of real-time text in O(M) space to respond to a variety of string-related queries without using any sophisticated data structure. The algorithm presented in this paper reports all substrings of a given pattern in the online text stream of data in real-time as the pattern or any of its substrings appear in the stream. Needless to say, the same algorithm can also be applied to offline databases. The work presented in this paper has a direct relation with our previous work [49], [50], and [51]. Therefore, for a prompt reference, we provide some theoretical discussion such as pattern preprocessing, lemma, and some of the examples given in [49], [50], and [51]. The rest of the paper is organized as follows: - In section II, we introduce several terms and notations. Section III presents a lemma that provides a theoretical base for the algorithm. In section IV, we present the pattern preprocessing algorithm, which is a prerequisite for the algorithm in the search phase. In section V, we present another lemma that extends the discussion given in section III. In section VI, we present the algorithm to search for patterns and their substrings in the text stream. A brief discussion of the time and space requirements of the algorithms is provided in section VII. To support our theoretical claims, experimental evidence is presented in section VIII. Finally, we conclude our work in section IX.

PRELIMINARIES

We use the following assumptions and notations. 'λ' represents a finite non-empty ordered set of characters called an alphabet, such that $|\lambda|=\sigma$ is the size of the alphabet. 'λ_i' is the i^{th} character in λ such that $1 \le i \le \sigma$. String S is a finite sequence of characters defined over the alphabet λ. $S[i]$ or s_i represents the i^{th} character of string S, where, 'i' is referred to as the shift, location, or index in S. A substring (or factor) S' of a string $S=s_1s_2\ldots\ldots s_M$ is a contiguous sequence of characters in S such that $S'=s_{i+1}\, s_{i+2}\ldots s_{i+k}$, where, $i \ge 0$ and $i+k \le M$. A prefix of string S is a substring of S that includes the first character of S, and a suffix of string S is a substring of S that includes the last character of S. Throughout the paper, T and P represent non-empty text and pattern strings of sizes N and M respectively. Both T and P are defined over the alphabet λ, typically, $M << N$. Because we are concerned with index locations, all of the variables used in this paper represent integers. Throughout the paper, we use the phrase *"Number of matches of P in T at shift t",* which refers to the number of character matches between P and T when pattern P is aligned with location t in T. In the algorithms, we refer to this as the number of hits in T at shift t by pattern P.

LEMMA

Consider the text string $T = t_0\ t_1\ t_2\ t_3\ t_4\ t_5 = DBCDAB$ of size N, and the pattern string $P = DABCD$ of size M. It is easy to see that one-character match is present, provided that P is aligned at location -4 in T (assume that there is such a location). Similarly, 3-character matches may be found when P is aligned at locations -1 and 3 in T. Traditionally, pattern P is aligned with all locations i in T such that $0 \leq i \leq N\text{-}M$, ignoring all other locations. However, while searching, considering all *i's* in the range $(1\text{-}M) \leq i \leq (N\text{-}1)$ may provide useful information, particularly when the pattern is considerably large and character matches exist at opposite ends of the pattern and the text.

Lemma

Let T and P be non-empty text and pattern arrays defined over an alphabet λ such that: $T = T[0....N\text{-}1]$, and $P = p_0\ p_1....p_{M-1}$. For each shift j in P, we define a set $R_{j\ such}$ that $R_j = \{i - j \mid T[i] = p_j, \forall\ 0 \leq i \leq N\text{-}1\}$. Further, let $R_0 \cap R_1 \cap R_2 ..... \cap R_{M-1} = S$. Then, every element $t \in S$ represents an exact match of P at shift t in T, and the cardinality $|S|$ represents the number of occurrences of P in T.

Proof: The proof of the lemma is given in our previous work [].

Example 1: Let $T = GCABABABCBA$ be a text array and $P = ABAB$ be a pattern array of size 4. For the given text we have i such that $0 \leq i \leq 10$. For each shift j ($0 \leq j \leq 3$) in P we create a set R_j such that $R_j = \{i - j \mid T[i] = p_j, 0 \leq i \leq 10\}$. Which gives: $R_0 = \{2, 4, 6, 10\}$, $R_1 = \{2, 4, 6, 8\}$, $R_2 = \{0, 2, 4, 8\}$, and $R_3 = \{0, 2, 4, 6\}$. Hence, $R_0 \cap R_1 \cap R_2 \cap R_3 = S = \{2, 4\}$. This shows P occurs in T twice (|S|) at locations *2* and *4*.

Corollary 1: Given the M sets R_j defined as in the lemma given above. Let f_t be the frequency of the occurrence of an integer *'t'* in all sets. Then, f_t represents the number of character matches when P is aligned with shift t in T. We have already seen that $(1\text{-}M) \leq t \leq (N\text{-}1)$.

Proof: Please refer to [].

Example 2: Consider example 1, the integer *6* appears in three sets: R_0, R_1 and R_3. Hence, the frequency of the integer 6 is equal to $f_6 = 3$. Which shows three-character matches of P when P is aligned with shift *6* in T. Similarly, $f_8 = f_0 = 2$ reveals two-character matches of P when P is aligned with locations *8* and *0* in T.

PATTERN PREPROCESSING

A pattern must be processed before it can be used by the algorithms presented in this paper. Pattern processing is performed using a well-known technique called hashing with separate chaining. For this, we use an array of pointers *'shift []'* of size σ. Each pointer in the array points to a linked list that stores all shifts of a character in the pattern. Initially, all linked lists are empty. The pattern is read from left to right one character at a time. Based on the *ASCII* value of the pattern character, the character shift is stored in the corresponding linked list. For example, let *'s'* be the shift of a character in the pattern that is being read, and let *'j'* be its *ASCII* value. For this character, a node containing integer *'s'* is created, and then inserted at the beginning of the linked list *shift[j]*. The following example and figure a clarifies this process. Let *P=FCTHZCTZCF* be a pattern of size 10. The *ASCII* codes of the pattern characters are- *F=70, C=67, T=84, H=72,* and *Z=90*. For the first character, *'F',* which is at shift *0*, a node containing *'0'* is created, and then

inserted in the link list at *shift [70]*. Note that nodes are inserted at the beginning of the linked list. Thus, all linked lists are created in reverse order, which is a prerequisite for the substring search algorithm. We assume the maximum possible *ASCII* code of a character in the alphabet λ to be 127.

Pattern Preprocessing Algorithm: Input: pattern characters. Output: A set of σ linked lists. Each linked list stores all shifts of a character in the pattern in reverse order. Note: the variable 'm' represents the 'shift' of a character in the pattern. The substring search algorithm presented in section VI requires the initial index of the pattern to be 1. Hence, for the substring algorithm variable m must be initialized to 1.

```
'node' is a structure with two fields: integer s, and node type pointer *next
node *shift[σ]                          /* array of node type pointers, σ is the size of the alphabet*/
integer i, j, m = 0                             /* character shift, initialize m=1 for the substring search algorithm */
for i = 0 to σ-1 do
        shift [ i] = NULL               /*in the beginning, all linked lists are empty*/
end for
while (Not end of the pattern) do       /* read pattern */
        j = ASCII code of the pattern character
        node *ptr = new node            /* create a node */
        ptr → s = m++                   /* store 'shift' in the node, and then increment m */
        ptr→ next = shift [j]           /* add the node to the beginning of the list pointed by shift[j] */
        shift[j] = ptr
end while
```

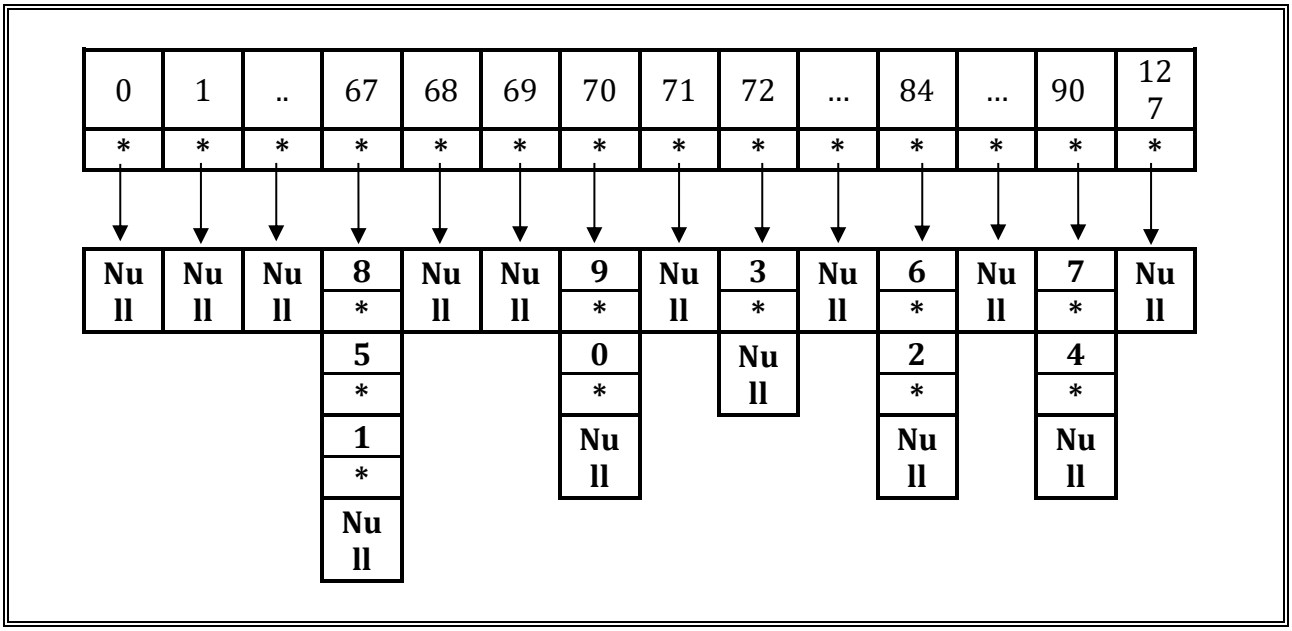


Figure 1: The Shift Array (Each pointer in the array points to a linked list)

SUBSTRING SEARCH

A pattern of size *M* can have *M(M+1)/2* non-empty substrings at most. In this section, we attempt to identify all locations in the text where any substring of the pattern is found. Such problems are common in computational biology, speech recognition systems, anti-virus

systems, and many other applications. In practical situations, for a given pattern, we may be interested in portions of the database where pattern substrings of a least certain length are found. Therefore, the problem we consider here is as follows:

Let T[1…N] be a text array of size N, and P[1…M] be a pattern array of size M. Then, for a given integer k (1 ≤ k ≤ M), report all locations 1 ≤ i ≤ N in T where a substring of P length ≥ k is found. For reasons explained later, we assumed the initial index of both the text and the pattern to be 1.
A naïve algorithm for the extraction of all pattern substrings would align the pattern at each location in the text to search for consecutive character matches. This would require an array of size *N* and *MN* character comparisons. Another approach is to divide the pattern into its possible *M(M+1)/2* substring, and then, for each substring apply the *KMP* algorithm [3]. In this manner, the given task can be accomplished in $O(M^2N)$ time. Using a two-dimensional array, dynamic programming can be applied to search for pattern substrings in *O(MN)* time and *O (NM)* space. Other methods include creating indexes over the text using suffix trees in *O(N)* time and space and then applying search for each of the *M-1* suffixes of the pattern, which would make execution time in the search phase independent of *N*. However, the difficulties associated with these methods have already been discussed in Section I. In the following discussion, we first introduce the term *"i-j chain"*. Later, we shall demonstrate how '*i-j chains*' can be used to search for the pattern substrings without making use of any complex data structures.

Lemma: The 'i-j Chain'
Let *T[1….N]* be a text array of size *N,* and let *P[1….M]* be a pattern array of size *M.* For each shift *j* in *P*, we define a set R_j such that: $R_j = \{ i \mid T[i] = P[j] \ \forall 1 \le i \le N \}$. Then, for a common substring of length *k* between *P* and *T* ($1 \le k \le M$), there must be an index *i* such that: $i \in R_j$, $(i+1) \in R_{j+1}$, $(i+2) \in R_{j+2}$…… $(i+k-1) \in R_{j+k-1}$, which we refer to as an '*i-j chain'* of length *k.* Notice, the term 'i-j chain' is named after the initial location of the chain.

Proof: Each $i \in R_j \Rightarrow T[i]=P[j]$ 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 common substring of length 1. Similarly, $(i+1) \in R_{j+1}$ represents a match for the $(i+1)^{th}$ character of *T* with the $(j+1)^{th}$ character of *P,* which indicates the existence of a common substring of length *2* and so on. Therefore, it follows that: $i \in R_j$, $(i+1) \in R_{j+1}$, ……$(i+k-1) \in R_{j+k-1}$ represents a common substring of length *k* between *P* and *T*.

Example 3: Let the pattern **P** and the text T be as follows:

T	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
	B	A	B	A	B	B	A	A	B	B	A	B	B	A	B	A	B	B	A	D	B	B	A

`

P	1	2	3	4	5	6	7	8
	A	B	A	B	B	A	C	B

Now, for each *j* in *P* the sets are given as follows:
$\boldsymbol{R_1}$ *= {2, 4, 7, 8, 11, 14, 16, 19, 23},*
$\boldsymbol{R_2}$ *= {1, 3, 5, 6, 9, 10, 12, 13, 15, 17, 18, 21, 22},*
$\boldsymbol{R_3}$ *= {2, 4, 7, 8, 11, 14, 16, 19, 23},*

$\boldsymbol{R_4}$*={1, 3, 5, 6, 9, 10, 12, 13, 15, 17, 18, 21, 22},*
$\boldsymbol{R_5}$*={1, 3, 5, 6, 9, 10, 12, 13, 15, 17, 18, 21, 22},*
$\boldsymbol{R_6}$ *= {2, 4, 7, 8, 11, 14, 16, 19, 23},*
$\boldsymbol{R_7}$ *= { },*
$\boldsymbol{R_8}$ *= {1, 3, 5, 6, 9, 10, 12, 13, 15, 17, 18, 21, 22}.*

To discover '*i-j chains*' we begin with all elements of set R_1, and then proceed through the remaining sets. In the process, we find an index *i=2* in R_1 such that: $2 \in R_1$, $(2+1) \in R_2$, $(2+2) \in R_3$, $(2+3) \in R_4$, $(2+4) \in R_5$, and $(2+5) \in R_6$, which corresponds to a *'2-1 chain'* of length 6. The *2-1 chain'* so discovered reveals the presence of 6 consecutive matches between the pattern and the text, which is '*ABABBA'* equivalent to *T[2…7]* or *P[1….6].* To view all such '*i-j chains'* refer to Figure 2. Each chain of length *k* in the figure corresponds to a common substring of length *k* between *P* and *T*. The length of an '*i-j chain'* can vary between *1* and *M*.

Let us discuss the construction of these chains. Corresponding to each unique pattern character, character shifts in the pattern are A = 6→3→1, B = 8→5→4→2, and C = 7. Consider Figure 2 in a real-time sense moving from top to bottom. Each row of the figure contains a text character and its shift *'i'* in the text. With each incoming text character, set R_j grows vertically downwards. For example, for the first text character *T [1] = B,* we retrieve all shifts of *'B'* from the shift array, which are: *8, 5, 4,* and *2.* Because $R_j = \{ i \mid T[i] = P[j] \}$, we add the first element *i=1* to the sets R_8, R_5, R_4, and R_2. As the sets grow, we connect the current cell with the immediately upper leftward cell if a contiguous sequence of *i* is found, which creates a diagonal chain. The superscript of each *i* represents the length of the '*i-j chain'* thus far constructed. In the last column, for each row, the largest superscript *k* in the row is chosen to report the length of the common substring that is present at the location *i-k+1* in *T*. The substring so found is indicated in boldface in the last column.

(i) Text Char	R_1	R_2	R_3	R_4	R_5	R_6	R_7	R_8	Length of the largest chain ending at *i*	Substring & Starting Position
1 B		1^1		1^1	1^1			1^1	1	B (at 1)
2 A	2^1		2^2			2^2			2	BA (at 1)
3 B		3^2		3^3	3^1			3^1	**3**	**BAB at (1)**
4 A	4^1		4^3			4^2			3	ABA (at 2)
5 B		5^2		5^4	5^1			5^1	4	ABAB (at 2)
6 B		6^1		6^1	6^5			6^1	5	ABABB (at 2)
7 A	7^1		7^2			7^6			**6**	**ABABBA (at 2)**
8 A	8^1		8^1			8^1			1	A (at 8)
9 B		9^2		9^2	9^1			9^1	2	AB (at 8)
10 B		10^1		10^1	10^3			10^1	3	ABB (at 8)
11 A	11^1		11^2			11^4			**4**	**ABBA (at 8)**
12 B		12^2		12^3	12^1			12^1	3	BAB (at 10)
13 B		13^1		13^1	13^4			13^1	4	BABB (at 10)
14 A	14^1		14^2			14^5			**5**	**BABBA (at 10)**
15 B		15^2		15^3	15^1			15^1	**3**	**BAB (at 13)**
16 A	16^1		16^3			16^2			3	ABA (at 14)
17 B		17^2		17^4	17^1			17^1	4	ABAB (at 14)
18 B		18^1		18^1	18^5			18^1	5	ABABB (at 14)
19 A	19^1		19^2			19^6			**6**	**ABABBA (at 14)**
20 D									0	--------
21 B		21^1		21^1	21^1			21^1	1	B (at 21)
22 B		22^1		22^1	22^2			22^1	2	BB (at 21)
23 A	23^1		23^2			23^3			**3**	**BBA (at 21)**

Figure 2: The *'i-j Chains' (*common substrings between *P = ABABBACB* and *T)*

THE Hit-INDEX SUBSTRING SEARCH ALGORITHM

To construct *chains* in Figure 2 we need a table of size *MN*. If we could visualize the process using a one-dimensional array, then each 'upper left' cell to a cell in Figure *2* would become the immediate left cell in the one-dimensional array. However, this approach presents further challenges, which we will discuss in detail below. In the algorithm presented below, we use an array *'hit[]'* of size *M +1*. Each cell of the array has two integer fields: *'len'* and *'ts'*. The field *'len'* records the length of the '*i-j chain'*, and the *'ts'* records the time-stamp that is used to determine whether a sequence '*i-j chain*' can be created. The *'length'* variable tracks the longest chain formed thus far with respect to the most recent input text character.

'Hit-Index Substring Search' Algorithm: Input: Pattern in the form of a shift array, and a stream of text characters of any size > 0. Output: all locations in the text where a substring of P length ≥ k is found. We assume the initial text index to be 1. Output is shown in real-time. Please refer to Figure 3. Correctness of the algorithm follows from the lemma 'i-j chain'.

```
integer i =0, j, t, length, k                    /* k is the minimum length of the substring to be searched */

Let 'node' be the structure with two fields: integer s, and node type pointer *next
Let "cell" be the structure with two fields: - integer len, and integer ts
cell hit[M+1]
for j = 0 to M do
hit[j].ts = -1                                   /* initialize the hit array time-stamp */
end for
while (Not end of the text) do
j = ASCII value of the input text character
        node *ptr = shift[j]                     /* jump to the j^th link list in the array */
        length = 0,      i++
        while(ptr != NULL) do                    /* retrieve all shifts one by one  */
                t = ptr→s
                hit[t].ts = i                    /* stamp index t with the current time stamp i */
                if(hit[t-1].ts == (i-1))   /*look immediately left whether sequence (i- chain) can be formed */
                        hit[t]. len = hit[t-1]. len + 1      /*if yes, increase the chain length */
                else
                        hit[t]. len = 1          /* too old time stamp, set chain length to 1 */
                if(hit[t]. len > length)         /* keep track of the largest chain in the cluster */
                        length = hit[t].len
                ptr = ptr→next                   /* retrieve next shift from the list*/
        end while
        if(length ≥ k)                           /* display substring length and location*/
Print" Substring length and Location="  length,  i-length + 1
end while
```

Run Time Behavior: 'Hit-Index Substring Search' Algorithm

In Figure 3, as the text characters arrive, each row shows the most recent status of the hit array. Note that except first row and column, the two cells *(i, j)* and upper-left *(i-1, j-1)* in Figure 2 correspond to the cells *j* and (*j-1)* in Figure 3. Thus, to construct a chain, we now look to the cell to the immediate left rather than the upper left. Let us assume that *i* is a global timestamp that increases by one with the arrival of each text character. Each cell of the array maintains its own local timestamp *'ts*'. The local time-stamp for each cell of the array cells is initialized to -1. With the arrival of the text character *T[i],* depending on the linked list in the shift array, some of the cells in the '*hit[]'* array is stamped with the current timestamp *i*, for the other cells, the timestamps remain the same. In other words, some of the cells in the '*hit[]'* array have been hit. In Figure 3, cells receiving hits are indicated in underlined boldface. Following a hit at '*hit[t]'*, the algorithm looks to the immediate left neighbor of '*hit[t]'* that is '*hit [t-1]'.* If the time-stamp at *hit[t-1]* is exactly one less than the current global time-stamp *i*, then the chain length at '*hit[t]'*, which is indicated by a superscript, is updated. Otherwise, the algorithm concludes that the time-stamp at index '*hit[t-1]'* is too old, and sets the chain length at '*hit[t]'* to 1. Once all cells in the row *i* have been hit, the length of the largest substring ending at *i* is reported, which is simply the largest superscript among the cells that have been hit. In row *20 D,* no cell receives a hit; hence, the length remains at zero. Note that the cells in the '*hit[]'* array are time-stamped from right to left in reverse order. This is done to ensure that the time stamp of the left cell is not

updated before it is read. This is also precisely why the linked lists are created in reverse order during the pattern preprocessing phase. This results in two benefits: –each insertion into a linked list require *O(1)* time in the pattern processing phase, and the shifts can be read in reverse order during the search phase. As described before, following a hit at *hit[t]* algorithm looks for *hit[t-1],* which is not possible in case *t = 0*. Therefore, to avoid the shift *0*, in the pattern preprocessing algorithm presented in section IV, the variable '*m'* must be initialized to 1.

(i) **Text Char**	**Indexes to be hit**	***0***	***1***	***2***	***3***	***4***	***5***	***6***	***7***	**8**	**Length of the largest chain ending at *i***	**Substring & Starting Position**
	Initialize ->	-1	-1	-1	-1	-1	-1	-1	-1	-1		
1 B	8, 5, 4, 2	-1	-1	**1**[1]	-1	**1**[1]	**1**[1]	-1	-1	**1**[1]	1	B (at 1)
2 A	6, 3, 1	-1	**2**[1]	1[1]	**2**[2]	1[1]	1[1]	**2**[2]	-1	1[1]	2	BA (at 1)
3 B	8, 5, 4, 2	-1	2[1]	**3**[2]	2[2]	**3**[3]	**3**[1]	2[2]	-1	**3**[1]	**3**	**BAB at (1)**
4 A	6, 3, 1	-1	**4**[1]	3[2]	**4**[3]	3[3]	3[1]	**4**[2]	-1	3[1]	3	ABA (at 2)
5 B	8, 5, 4, 2	-1	4[1]	**5**[2]	4[3]	**5**[4]	**5**[1]	4[2]	-1	**5**[1]	4	ABAB (at 2)
6 B	8, 5, 4, 2	-1	4[1]	**6**[1]	4[3]	**6**[1]	**6**[5]	4[2]	-1	**6**[1]	5	ABABB (at 2)
7 A	6, 3, 1	-1	**7**[1]	6[1]	**7**[2]	6[1]	6[5]	**7**[6]	-1	6[1]	**6**	**ABABBA (at 2)**
8 A	6, 3, 1	-1	**8**[1]	6[1]	**8**[1]	6[1]	6[5]	**8**[1]	-1	6[1]	1	A (at 8)
9 B	8, 5, 4, 2	-1	8[1]	**9**[2]	8[1]	**9**[2]	**9**[1]	8[1]	-1	**9**[1]	2	AB (at 8)
10 B	8, 5, 4, 2	-1	8[1]	**10**[1]	8[1]	**10**[1]	**10**[3]	8[1]	-1	**10**[1]	3	ABB (at 8)
11 A	6, 3, 1	-1	**11**[1]	10[1]	**11**[2]	10[1]	10[3]	**11**[4]	-1	10[1]	**4**	**ABBA (at 8)**
12 B	8, 5, 4, 2	-1	11[1]	**12**[2]	11[2]	**12**[3]	**12**[1]	11[4]	-1	**12**[1]	3	BAB (at 10)
13 B	8, 5, 4, 2	-1	11[1]	**13**[1]	11[2]	**13**[1]	**13**[4]	11[4]	-1	**13**[1]	4	BABB (at 10)
14 A	6, 3, 1	-1	**14**[1]	13[1]	**14**[2]	13[1]	13[4]	**14**[5]	-1	13[1]	**5**	**BABBA (at 10)**
15 B	8, 5, 4, 2	-1	14[1]	**15**[2]	14[2]	**15**[3]	**15**[1]	14[5]	-1	**15**[1]	**3**	**BAB (at 13)**
16 A	6, 3, 1	-1	**16**[1]	15[2]	**16**[3]	15[3]	15[1]	**16**[2]	-1	15[1]	3	ABA (at 14)
17 B	8, 5, 4, 2	-1	16[1]	**17**[2]	16[3]	**17**[4]	**17**[1]	16[2]	-1	**17**[1]	4	ABAB (at 14)
18 B	8, 5, 4, 2	-1	16[1]	**18**[1]	16[3]	**18**[1]	**18**[5]	16[2]	-1	**18**[1]	5	ABABB (at 14)
19 A	6, 3, 1	-1	**19**[1]	18[1]	**19**[2]	18[1]	18[5]	**19**[6]	-1	18[1]	**6**	**ABABBA (at 14)**
20 D	-----	-1	19[1]	18[1]	19[2]	18[1]	18[5]	19[6]	-1	18[1]	0	--------
21 B	8, 5, 4, 2	-1	19[1]	**21**[1]	19[2]	**21**[1]	**21**[1]	19[6]	-1	**21**[1]	1	B (at 21)
22 B	8, 5, 4, 2	-1	19[1]	**22**[1]	19[2]	**22**[1]	**22**[2]	19[6]	-1	**22**[1]	2	BB (at 21)
23 A	6, 3, 1	-1	**23**[1]	22[1]	**23**[2]	22[1]	22[2]	**23**[3]	-1	22[1]	**3**	**BBA (at 21)**

Figure 3: Run Time Behavior Algorithm Substring Search P=ABABBACB

TIME & SPACE REQUIREMENTS

In the pattern preprocessing phase, we use an array of size σ to hold multiple linked lists. For each pattern character, a node is created and then mapped to the appropriate array index based on its *ASCII* value. Each insertion at the beginning of a linked list costs *O (1)* time. For a pattern of size *M,* this implies pattern processing time to be *O(M),* and required memory to be *O (σ+M).*. In the search phase, the algorithms require an array of size *M* as an additional workspace*.* Therefore, the total run-time memory requirement for the substring search algorithm is

O(σ+M) + *O(M),* which is *O(M).* Thus, the total run time memory requirement is independent of *N,* meaning that the algorithms are ideal for large real-time applications. Note the length of a linked list in the shift array represents the frequency of a character in the pattern. Now, the outer while-loop of each algorithm reads *N* text characters one by one, and for each incoming text character, the inner while-loop runs *fp* times where *fp* is the frequency of the text character in the pattern. For the average case analysis, we can assume the average frequency of a character in the pattern to be M / σ, which yields an average execution time of *O(NM / σ).* For a detailed discussion on execution time please refer to [49].

EXPERIMENTAL RESULTS

As we discussed previously, to search for pattern substrings in text, dynamic programming requires *O(MN)* space and time, whereas suffix trees and suffix arrays provide execution time that is independent of *N*. However, with these data structures, text preprocessing is a prerequisite, which is a costly phase. Preprocessing phase requires *O(N)* time and consumes space that is proportional to the size of the text, i.e. *O(N)*. Here, the asymptotic notation *O(N)* may not fully convey the actual memory requirement because of the potentially large value of the missing constant. As stated previously, for a text of *700 MB* in size, a suffix tree may consume approximately *45 GB* of space. Hence, working with such a tree might need disk accesses, which may have a significant influence on the total run-time. By contrast, our algorithms consume space linear to the size of the pattern, i.e., *O(M),* meaning that the total run-time space requirement is independent of *N*. Broadly speaking, for a pattern of size *100*, the total run-time space requirement should be no more than *0.5 MB*. Hence, one of the performance measurement parameter 'space' is the primary aspect in which our algorithm distinguishes itself from other known techniques. However, if we could afford to sacrifice *O(N)* time and a huge space to process the text using these data structures, the execution time they provide turn out to be independent of *N*. Therefore, performing experiments for a comparative study of our algorithms with those described above is neither justifiable, nor it will reveal any hidden information. However, to provide a rough estimation of the execution time for our algorithms, we present experimental results in Table 3, and a corresponding graph is provided in Figure 4.

The total execution time required in the search phase predominantly depends on the inner while-loop of the algorithms, in which the 'hit-indexes' are generated. Note that the inner 'while-loop' is skipped when the incoming text character is not present in the pattern, which helps to accelerate the search process. To perform our experiments, we used the biomedical databases [48]. These databases and their supporting tools are freely available to the research community. We used six different protein sequence databases of various sizes ranging from approximately *0.5* million to *1.5* million characters: 'gvrt4.fsa_aa', 'gbtsa57.fsa_aa', 'gbtsa69.fsa_aa', 'gbcon287.fsa_aa', 'gbtsa68.fsa_aa', and 'gbtsa78.fsa_aa'. These files are available in FASTA format. In FASTA, each sequence begins with a description, followed by the protein sequence. A protein sequence is represented by a series of English letters. Each letter in the sequence represents a particular amino acid. From each database file, 10 patterns of various sizes were randomly chosen from *10* different locations. Each pattern was then searched in the same file, and the total number of induced hits was recorded. The results are summarized in Table 1. The first row and first column of the table represent the number of characters in the text and the pattern respectively. Each other entry indicates the number of induced hits for the corresponding pattern and text sizes. The corresponding graph is provided in Figure 4. Consider the hit count in the two rows corresponding to *M=10,* and *M=100*. For the same text file, for a

factor of *10* increase in the size of the pattern, the number of hits also increased by almost a factor of *10*. Similarly, consider the file "gbtsa68.txt" which is nearly three times larger than the file "gbvrt4.txt". For each pattern, the number of hits in the "gbtsa68.txt" column is nearly three times larger than the number of hits in the "gbvrt4.txt" column. The character frequencies in all files were found to be highly non-random, which is an unfavorable situation for the algorithm. For instance, in the file "gbvrt4.txt" the character distribution is as follows: total characters 486318, with an individual character count of A(28142), B(1602), C(5705), D(13007), E(14504), F(21828), G(23022), H(9837), I(24878), J(305), K(13565), L(48910), M(11880), N(15522), O(142), P(20372), Q(10952), R(11157), S(27287), T(24824), U(24), V(21123), W(8602), X(110), Y(11892), Z(3), and other characters (117139).

Table 1: Number of Hits (Text Size N, and the Pattern Size M)

	gbvrt4 (N=486318)	gbtsa57 (N=694145)	gbtsa69 (N=967719)	gbcon287 (N=1121767)	gbtsa68 (N=1394033)	gbtsa78 (N=1508004)
M=10	195383	311119	336314	488064	657774	728660
M=20	550185	554434	758637	994580	1253402	1591960
M=30	713868	802477	1142839	1541309	2036019	2110028
M=40	1094220	1147052	1538355	2045358	2339446	3009621
M=50	1218372	1464215	2131982	2562122	3342183	3690275
M=60	1427690	1857907	2505460	3285277	3448562	4777869
M=70	1736579	1927084	2698948	4171947	4492603	5237806
M=80	2035106	2299393	2887593	4400426	4956497	5981492
M=90	2372188	2514785	3436054	4929523	6306204	6653915
M=100	2273617	2748085	3678333	5640917	6615246	7451683

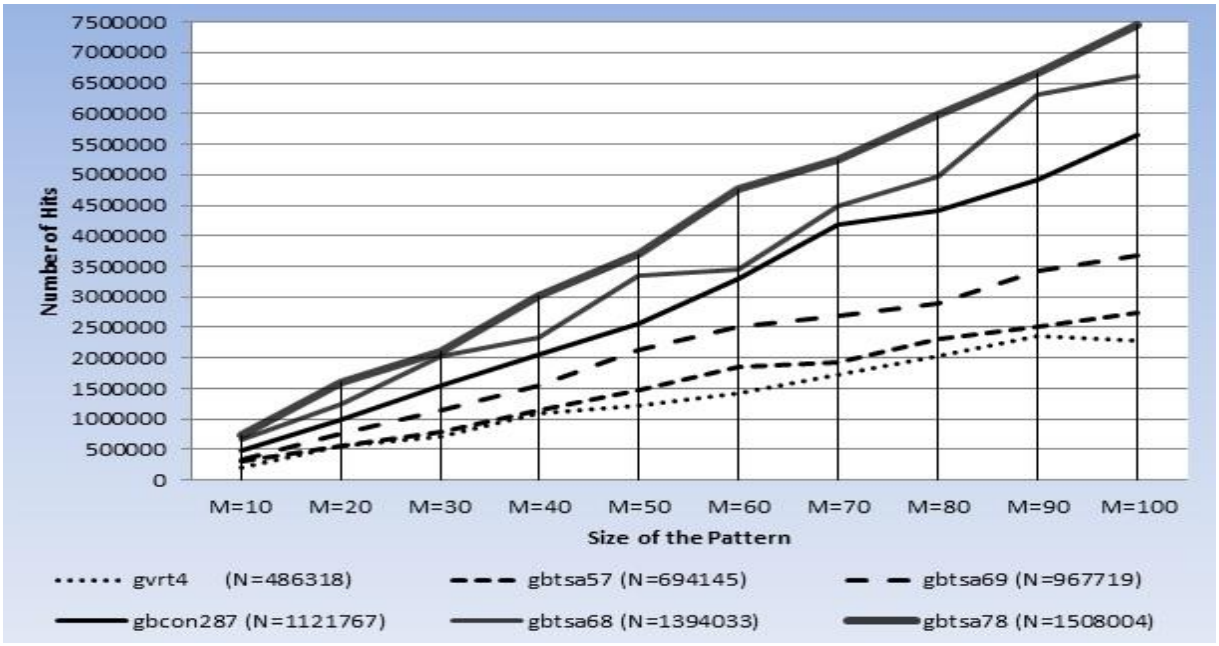


Figure 4: Number of Hits Induced: Text size *N*, and the Pattern Size *M*

CONCLUSION

Developing string processing algorithms without creating indexes over the text has been a considerable challenge. This paper provides highly practical solutions that do not use complex data structures or methods. The proposed solutions can be easily implemented with minimal effort and resources. Furthermore, proposed solutions can be used in filtering large databases to determine the regions of best pattern alignment.

References

[1] Morris J., and Pratt V. A linear pattern matching algorithm. Technical Report 40, Computing Center, University of California, Berkeley, 1970

[2] Aho A., and Corasick M. Efficient string searching: An aid to bibliographic search. Communications of the ACM, 18, pp. 333-40, 1975

[3] Knuth D., Morris J., and Pratt V. Fast pattern matching in strings. SIAM Journal on Computing, 6, pp. 323-50, 1977.

[4] Boyer R., and Moore S. A fast string searching algorithm. Communications of the ACM, 20, pp. 762–772, 1977.

[5] Horspool N. Practical fast searching in strings. Software Practice and Experience, 10, pp. 501–506. 1980

[6] Karp R., and Rabin M. Efficient randomized pattern-matching algorithms. IBM Journal Research and Development, 31, pp. 249-60. 1987.

[7] Levenshtein I. Binary codes capable of correcting deletions, insertions, and reversals, Soviet Physics Doklady, 10 (8), pp. 707–710. 1966

[8] Hamming R. Error detecting and error correcting codes, Bell System Technical Journal, 29 (2), pp. 147–160, 1950

[9] Lowrance R., and Wagner A. An extension of the string-to-string correction problem, Journal of the ACM 22, pp. 177–183, 1975

[10] Sellers P. The theory and computation of evolutionary distances. Journal of Algorithms, 1, pp. 359–373, 1980

[11] Masek W., and Paterson M. A faster algorithm for computing string edit distances. Journal of Computer System Sciences, 20, pp. 18–31, 1980

[12] Ukkonen E. Finding approximate patterns in strings. Journal of Algorithms, 6, pp. 132–137, 1985.

[13] Landau G., and Vishkin U. Fast string matching with k differences. Journal of Computer System Sciences, 37, pp. 63–78, 1988

[14] Galil Z., and Giancarlo R. Data structures and algorithms for approximate string matching. Journal of Complexity, 4, pp. 33–72, 1988

[15] Landau G., and Vishkin U. Fast parallel and serial approximate string matching. Journal of Algorithms, 10, pp. 157–169, 1989

[16] Galil Z., and Park K. An improved algorithm for approximate string matching. SIAM Journal on Computing, 19, pp. 989–999, 1990.

[17] Ukkonen E., and Wood D. Approximate string matching with suffix automata. Algorithmica, 10, pp. 353–364, 1993.

[18] Cole R., and Hariharan R. Approximate string matching: a simpler faster algorithm. Proceedings of the 9th ACM-SIAM Symposium on Discrete Algorithms, pp. 463–472, 1998.

[19] Myers G. A fast bit-vector algorithm for approximate string matching based on dynamic programming. Journal of the ACM, 46(3), pp. 395–415, 1999.

[20] Wu S., and Manber,U. Agrep—a fast approximate pattern-matching tool. Proceedings of the USENIX Technical Conference. USENIX Association, Berkeley, CA, USA. pp. 153–162. 1992.

[21] Wu S., Manber U., and Myers E. A subquadratic algorithm for approximate limited expression matching. Algorithmica 15(1), pp. 50–67, 1996

[22] Melichar B. String matching with k differences by finite automata. Proceedings of the International Congress on Pattern Recognition (ICPR '96), pp. 256–260, 1996.

[23] Tarhio J., and Ukkonen E. Approximate Boyer–Moore string matching. SIAM Journal of Computing, 2, pp. 243–260, 1993

[24] Chang W., and Lawler E. Sublinear approximate string matching and biological applications. Algorithmica, 12, pp. 327–344, 1994

[25] Takaoka T. Approximate pattern matching with samples. Proceedings of the ISAAC 1994, L.N.C.S Springer Verlag, pp. 234–242, 1994.

[26] Baeza-Yates R., and Navarro G. Faster approximate string matching. Algorithmica, 23(2), pp. 127-158, 1999.

[27] Landau G., and Vishkin, U. Efficient string with k mismatches. Theoretical Computer Science, 43, 239-249, 1986

[28] Weiner P. In the proceedings of the 14th Annual IEEE Symposium on Switching and Automata Theory, pp. 1–11, 1973.

[29] McCreight E. A space-economical suffix tree construction algorithm, Journal of ACM 23, pp. 262-272, 1976.

[30] Ukkonen E. On-line construction of suffix trees, Algorithmica 41. pp. 249-260, 1995.

[31] Farach M. Optimal suffix tree construction with large alphabets. 38th IEEE Symposium on Foundations of Computer Science, pp. 137–143, 1997.

[32] Gusfield D. Algorithms on strings, trees and sequences. Computer Science and Computational Biology, Cambridge University Press, ISBN 0-521-58519-8, 1999.

[33] Galil Z., Giancarlo R. Improved string matching with k mismatches, SIGACT News 17 (4) pp. 52–54, 1986.

[34] Amir A., Lewenstein M., and Porat E. A Faster algorithms for string matching with k mismatches. Journal of Algorithms, pp. 257-275, 2004.

[35] Abrahamson K. Generalized string matching, SIAM Journal of Computing, 16 (6), pp. 1039–1051, 1987.

[36] Baeza-Yates, R., Perleberg C.H. Fast and practical string matching. Information Processing Letters, 59, pp. 21-27, 1996.

[37] Kurtz S. Reducing the space requirement of suffix trees, Software—Practice & Experience, 29(13),pp.1149-1171, 1999.

[38] Amir A., Kopelowitz T., Lewenstein M., and Lewenstein N. Towards real-time suffix tree construction string processing and information retrieval. 12th International conference SPIRE 2005, Lecture Notes in Computer Science, pp 67-78, 2005.

[39] Breslauer D., Giuseppe F. Near real-time suffix tree construction via the fringe marked ancestor problem. Journal of Discrete Algorithms, 18, pp. 32-48, 2013

[40] Manber U., and Myers E. Sufix arrays: A new method for on-line string searches, SIAM Journal on Computing, 22(5), pp. 935–948, 1993.

[41] Grossi R., and Vitter J. Compressed suffix arrays and suffix trees, with applications to text indexing and string matching, SIAM Journal on Computing, 35(2), pp. 378-407, 2005.

[42] Sadakane K. Compressed suffix trees with full functionality. Theory of Computer Systems, 41(4) pp. 589-607, 2007.

[43] Russo L., Navarro G., and Oliveira A. Proceedings of the 8th Latin American conference on Theoretical informatics, Springer-Verlag, pp. 362-373, 2008.

[44] Alimaki V., Gerlach W., Dixit K., and Makinen V. Engineering a compressed suffix tree implementation. In the Proceedings of the 6th WEA, pp. 217-228, 2007.

[45] Crochemore M., Hancart C., and Lecroq T. Algorithms on Strings, Cambridge University Press, 2007.

[46] Navarro G. A guided tour to approximate string matching. ACM Computing Surveys, 33 (1), pp. 31–88, 2001.

[47] Boytsov L. 2011. Indexing methods for approximate dictionary searching: Comparative analysis, ACM JEA 16 (1), pp. 1–9, 2011.

[48] http://www.ncbi.nlm.nih.gov/protein/

[49] Vinod Prasad. A Character Frequency based Approach to Search for Substrings of a Circular Pattern and its Conjugates in an Online Text, Computer Science Journal AGH University of Science and Technology, Poland, April 2021

[50] Vinod Prasad. A Novel Mathematical Model for Similarity Search in Pattern Matching Algorithms, Journal of Computer and Communications, Sept. 2020.

[51] Vinod Prasad. A Novel Algorithm for String Matching with Mismatches, 5th International Conference on Pattern Recognition Applications and Methods, Rome, ISBN 978-989-758-173-1, Feb 24-26, 2016.