String Matching Algorithm Using Multi-Characters Inverted Lists

CHOUVALIT KHANCOME
Computer Science Department, Faculty of Science
Ramkhamhaeng University
282 Ramkhamhaeng Road , Huamark , Bangkapi Bangkok
THAILAND.

Abstract:-This research article introduces a new string-matching algorithm that utilizes the multi-character inverted lists structure, implemented using a perfect hashing method. The new solution processes the pattern input string in a single pass. In the searching phase, the algorithm takes a linear time complexity. This approach demonstrates efficient and rapid performance in practical experiments when a binary character set is used. It also handles fewer characters and shorter lengths, even when the pattern length is increased. In some cases, this algorithm exhibits superior text search performance compared to other algorithms. It has been observed that this superiority is particularly evident when searching for smaller amounts of text. This algorithm can be explored more quickly, especially when the search factor is between 0.25 to 0.75 times the length of the pattern, approaching the performance of the traditional algorithm.

Key-Words: - String Matching Algorithm, Inverted Lists, Inverted Index, Pattern Matching, Exact String Matching.

Received: May 29, 2022. Revised: July 23, 2023. Accepted: September 6, 2023. Published: October 3, 2023.

1 Introduction

String matching is the process of finding keywords or character sequences in large texts. It is a fundamental technique in computer science with a wide range of applications, including search, data filtering, analysis, validation, data access, and artificial intelligence. In search tasks, string matching is used to find words or sentences in text documents, such as keyword-based website searches. In data filtering, it is used to filter out spam emails, censor offensive language, and sift through online shopping data. It can also be used to analyze sentiments and uncover trends or patterns in text. In validation, string matching is used to check passwords, detect copyright infringement, and identify privacy violations. In data access and manipulation applications, it can be used to replace and edit words in word processing programs. String matching is also essential for artificial intelligence, where it can be used to create versatile tools for a variety of situations.

Focusing on the principles of computer science, string matching is a principle that locates all occurrences of a pattern string, denoted as $p = c_1c_2c_3\ldots c_m$, in a given text string $T = t_1t_2t_3\ldots t_n$, where m is the length of p and n is the length of T. Essentially, the pattern p is transformed into a suitable data structure during the processing phase. Then, the searching methods scan the text T using appropriate techniques, which [8] are divides into the prefix search, the suffix search, and the factor search.

Existing solutions, as cited in references [1]-[4], [9]-[10], [15]-[19], and [22], use various data structures such as automata, shift tables, or bit-parallel techniques to generate the pattern *p*. During the search phase, these methods are used to reduce time complexity. The optimal solution [4] achieves a pre-processing time complexity of *O(m)* and a space complexity of *O(1)* while attaining an average search time complexity of *O(n)* and a best-case search time complexity of *O(n/(m+1))*. For a comprehensive overview of solutions to this problem, shown in the articles [7] and [8].

The inverted index has been used to address information retrieval challenges in a variety of applications, as described in references [2], [10], [17]-[19], and [23]. By focusing on keywords and their positions, the inverted index, as discussed in references [5], [20], and [21], can be adapted to different data structures, enabling faster searches. This raises the question of whether the inverted index can be used to develop a new data structure that can provide even faster solutions. The findings presented in this article provide an answer to this question.

Thus, this research article introduces a new exact string matching algorithm using a data structure that extends the inverted index concept discussed in references [5], [20], [21], and [23]. The data structure is called multi-character inverted lists (*m-cIVL*). The

algorithm has a pre-processing time complexity of *O(m/k)* and a searching time complexity of *O(n + nocc)*, where *m* is the pattern length, *k* is the factor by which a single pattern is divided, *n* is the length of the text being searched, and *nocc* is the matching time.

Empirical experiments reveal the effectiveness and speed of this approach when applied to binary characters. In certain instances, it excels in text search efficiency, particularly when employing fewer characters, shorter character lengths, and configuring the search factor (character splitting) within the range of 0.25 to 0.75 times the character length. When dealing with small search texts, such as 100 bytes, the new algorithm consistently yields search results that are either faster or equivalent to those produced by the traditional algorithm.

The remaining sections are organized as follows. Section 2 discusses related works and derives the algorithm's principles. Section 3 introduces the basic definitions and construction of the inverted lists. Section 4 explains the search algorithm and provides an illustrative example. Section 5 presents the experimental results. Section 6 discusses the findings. Section 7 concludes the paper, and Section 8 suggests algorithm improvements and future work.

2 Related Works

Since the inverted index structure has been used for the information retrieval problem, it has been applied to many applications. This structure represents words in the target documents in the form of <*documentID, word:pos*>, where *documentID* is the assigned number used to refer to the document, *word* is the keyword known as the vocabulary, and *pos* is the position of the word's occurrence in the *documentID*. The original sources [20] and [21] consider all documents as $D=\{D_1 \ldots D_n\}$, where D_i represents each document containing various keywords in multiple positions and $1 \le i \le n$. Surprisingly, each document *D* can be replaced by the multiple pattern $P=\{p^1, p^2, p^3, \ldots, p^r\}$, and each p^i can be represented as D_i. The characters and their occurrence positions in each p^i are referred to as individual posting lists. Subsequently, all posting lists are organized into a hashing table. The single inverted lists data structure can be represented as follows.

Example 1: An example with *p=aabcz* demonstrates an inverted list illustrated in Table 1.

Table 1. Example of single inverted lists.

∑)single-Character(	(IVL:inverted lists)
a	*<1:0>,<2:0>*
b	*<3:0>*
c	*<4:0>*
z	*<5:1>*

The efficient hashing table, known as perfect hashing, requires *O(n)* space and operates in *O(1)* time (as demonstrated in [11], [12], and [13]), where *n* is the size of the data. Typically, the perfect hashing principle is suitable for static keywords, such as reserved words in programming languages. This structure comprises: 1) A universal key, denoted as *U* and $f(\lambda)$, which accommodates all keys for accessing data within the table and 2) Two levels of implementation. The first level consists of k keys for accessing the second level using a function *f(k)*. The second level contains data items associated with the corresponding key, k. In this research, we assign U as the universal key and use *f(k)* for the first level of the perfect hashing table. The data items in the second level are groups of posting lists. This research applies the perfect hashing principle to accommodate multi-character inverted lists. An illustrative example is provided in the next section.

3 Multi-Characters Inverted Lists

Building upon the ideas presented above, this section provides fundamental definitions of inverted lists, accompanied by illustrative examples for each definition. Additionally, it explains the pre-processing phase, which outlines how to create and validate the inverted lists table.

Definition 1: Let *k* be a multiple of *m*, denoted as *k-m*, representing the character factor of the string pattern *p*, where m is the length of the pattern and *k=1,2, 3,... , m*.

Definition 2: The inverted lists, constructed using the *k-m* factor, are represented in the form of <*appearing position: termination status*> and are referred to as *m-cIVL*.

Example 2: For a pattern *p = aabcz*, when *k* is set to 2, 3, 4, and 5 as shown in Definition 1.

Table 2. The *m-cIVL* example.

∑ (2-m)	m-cIVL
aa	*<1:0>*
bc	*<2:0>*
z	*<3:1>*
∑ (3-m)	

aab	*<1:0>*
cz	*<2:1>*
∑ (4-m)	
aabc	*<1:0>*
z	*<2:1>*
∑ (5-m)	
aabcz	*<1:1>*

Definition 3. Any key word $w_{pos,ter}$ is a part of *p* derived representing any *k-m* factor, where *pos* is the position that appears in *p* from 1 to *m / k* by rounding an integer. If there is a fraction; *ter* is *1*, if the end character(s) of *p*, or *0* if not the last character(s).

Definition 4. Any keyword of *p* representing $w_{pos,ter}$ contains the following keywords: $w_{1,0}$, $w_{2,0}$, $w_{3,0}$, $w_{m/k-3,0}$ …, $w_{m/k-2,0,}$, $w_{m/k-1,0,}$ $w_{m/k,1}$.

Example 3: The 2-m of *p = aabcz*, as defined in definition 4.

$w_{1,0} = aa_{1,0}$
$w_{2,0} = bc_{2,0}$
$w_{m/k,1} = z_{3,1}$

Definition 5. Any multiple-character inverted lists structure is a form of representing definitions 3 and 4 in the form of $w_{pos,ter}$:<*pos,ter*>.

Example 4: The multiple-character inverted lists of *p = aabcz* from 2-m of definition 4.
$w_{1,0} = aa_{1,0}$ written with *aa: <1,0>*
$w_{2,0} = bc_{2,0}$ written with *bc: <2,0>*
$w_{m/k,1} = z_{3,1}$ written with *z: <3,1>*

Definition 6. Given that <*pos,0*> is the index of the multiple characters inverted lists as denoted by $I_{pos,0}$, and <*pos, 1*> is represented by $I_{pos,1}$. Therefore, $W_{pos,ter} : I_{pos,0}$ when *k* <*m / k* or $I_{pos,1}$ when *k = m / k*.

Definition 7. Let *HT* be a hashing table for the *m-cIVL* of Definition 6 with two columns, containing $W_{pos,ter}$ and either $I_{pos,0}$ or $I_{pos,1}$.

Table 3 illustrates the example.

Table 3. Any *m-cIVL* of *HT*.

$W_{pos,ter}$	$I_{pos,0}$ / $I_{pos,1}$
$W_{1,0}$	$I_{1,0}$
$W_{2,0}$	$I_{2,0}$
$W_{3,0}$	$I_{3,0}$
…	…
$W_{m/k-3,1}$	$I_{m/k-3,0}$
$W_{m/k-2,1}$	$I_{m/k-2,0}$
$W_{m/k-1,1}$	$I_{m/k-1,0}$
$W_{m/k,1}$	$I_{m/k,1}$

Theorem 1: Accessing I_{pos} or $I_{pos,1}$ in the *HT* table has a time complexity of O(1).
Proof: Assuming that *f(x)* is a hash function, $W_{pos,0}$ serves as a key for accessing any $I_{pos,0}$ and $W_{pos,1}$ serves as a key for accessing any $I_{pos,1}$.

Based on the properties of a perfect hashing table, accessing information within it typically has a complexity of *O(1)*. Therefore, using $f(W_{pos,0})$ to access $I_{pos,0}$ and $f(W_{pos,1})$ to access $I_{pos,1}$ in *HT* also has a complexity of *O(1)*. #

4 String Matching Algorithms

There are two phases of the algorithm: pre-processing and searching. These phases are characterized as follows.

The pre-processing algorithm creates a multiple inverted lists table and adds all $W_{pos,ter}$ and $I_{pos,0}$ / $I_{pos,1}$ pairs to the *HT* table. Then, the algorithm reads *p* as a *k-m* factor and generates characters for $W_{pos,ter}$ and $I_{pos,0}$ / $I_{pos,1}$ pairs, adding them to the *HT* table. Algorithm 1 shows the steps of the pre-processing algorithm.

Algorithm 1: Pre-processing phase
Input: $p[c_1c_2c_3…c_m]$ *of m lenth, k-m*
Output: table *HT*
1. Create empty *HT*
2. *pos=1, begin=1, end=k, terminate=0*
3. For *i=1* To m Do
4. $W_{pos,\ terminate} \leftarrow p[c_{begin}…c_{end}]$
5. $I_{pos,ter} \leftarrow W_{pos,\ terminate}$
6. $HT \leftarrow I_{pos,ter}$
7. *begin = begin +(k+1)*
8. *end = begin+k*
9. IF *end>= m* Then
10. *end = m*
11. End of IF
12. IF *end= m* Then
13. *terminate=1*
14. End of IF
15. *pos=pos+1*
16. End of For
17. Return *HT*

The time complexity of Algorithm 1 is *O(m/k)*, where *m* is the length of a pattern *p*, and *k* is the *k-m* value of the selected pattern *p*. The maximum space

required is $O(|\sum| + m)$ and, on average, it takes $O(2(m/k))$, as proven by the following theorems.

Theorem 2: The time complexity to create *HT* is $O(m/k)$ when $k>1$ or $O(m)$ when $k=1$.

Proof: By choosing k and defining the lengths of the multiple characters for each $W_{pos,ter}$, we establish that p has a length of m.

When k is greater than 1, the primary source of complexity arises from the 'For' loop (lines 3-16). Each iteration of this loop processes the creation of multiple inverted lists $I_{pos,ter}$ for the substring $p[c_{begin} \ldots c_{end}]$.

Next, $I_{pos,ter}$ is inserted into *HT*, taking $O(1)$ time as per Theorem 1. This loop operates m/k times, where k varies from 1 to m/k. Therefore, the complexity of the 'For' loop is $O(m/k)$. Other sections, both inside and outside the 'For' loop, have $O(1)$ processing time. When k equals 1, this loop operates m times, resulting in a time complexity of $O(m)$.

So, the most complex of the pre-processing algorithm is $O(m / k)$ when $k> 1$ or $O(m)$ when $k = 1$.#

Theorem 3: The space complexity of *HT*, processed by algorithm 1, takes $O(|\sum|+m)$ in the worst case and $O(2(m/k))$ in the average case scenario.

Proof: The space complexity is determined by the method used to create the multiple characters' inverted lists in the HT table.

First, an empty *HT* is created in line 1. Then, the 'For' loop in lines 3-16 is responsible for generating all of the *m-cIVLs*. Each inverted list is generated by creating a $W_{pos,\ terminate}$ value from $p[c_{begin} \ldots c_{end}]$ (as seen in line 4) and associating it with a unique key in the first column of the table. At the same time, the space required to store the inverted lists $I_{pos,ter}$ in lines 5-6 is $O(1)$ (single inverted list). Every inverted list is created uniquely with an existing key.

So, the number of inverted lists to be stored in *HT* is $2(m/k)$. In the average case, the space complexity is $O(2(m/k))$ when $k>1$. However, in the worst case ($k=1$), the first column of the table takes up $|\sum|$ space, and the second column takes up m space. Therefore, *HT* has a space complexity of $O(|\sum|+m)$ in the worst case and $O(2(m/k))$ in the average case.#

The searching algorithm starts with the default value given to the navigator and then scans the search window from front to back to match the pair. For each attempt to find a match, the algorithm obtains the characters from the $T[t_{begin} \ldots t_{end}]$ as a key to access *HT* using the hash function to find the termination status pos. If *pos* matches and *terminate* is *1*, then a correct match is found. The search algorithm is as follows.

Algorithm 2: SMCIVL
Input : *HT, k,* $p=c_1c_2,c_3 \ldots c_m$ $T = t_1t_2t_3 \ldots t_n$
Output : All occurrences are reported, and *T* is scanned.

```
1. end=k, terminate=0, j=1, matchwindow=1
2. While j<=(n-(k*m) Do
3.      pos=1, begin=j, end=begin+k, terminate=0,
      matchwindow=1
4.      While matchwindow=1 AND end<=n Do
5.         st=text[t_begin...t_end]
6.         IF HT(f(st)) Then
7.            h ← I_pos, ter of st
8.            IF h contains pos and terminate Then
9.               IF terminate=1 Then
10.                 report occurrence at text[t_begin...t_end]
11.                 matchwindow=0
12.              Else
13.                 pos=pos+1
14.                 begin=begin+k, end=begin+k
15.                 IF pos=m/k Then
16.                    terminate=1
17.                    IF (m mod k) !=0 Then
18.                       end=end-1
19.                    End of IF
20.                 End of IF
21.              End of IF
22.           Else
23.              matchwindow=0
24            End of IF
25.        Else
26.           matchwindow=0
27.        End of IF
28.     End of While
29.     j=j+1
30. End of While
```

This new search idea can compare multiple characters in a single comparison, making searching faster than ever. The search window can be more varied, and the search factor can be set to the width of the search window (*1, 2, 3, ..., m*), which will also make searching faster.

The time complexity of the algorithm is primarily determined by two loops: the first loop has a complexity of $O(n)$, and the inner loop iterates a number of times equal to $O(nocc)$, where *nocc* is the

number of successful matches. The proof is illustrated in the following Theorem 4.

Theorem 4: The multi-character string matching algorithm, utilizing multiple inverted lists, has a time complexity of *O(n + nocc)*.

Proof: The proof is divided into three parts. In the first part, starting from the second line, the entire complexity of the algorithm is controlled by the variable *j*, which ranges from *1* to *n - (k * m) - 3, n - (k * m) - 2, n - (k * m) - 1, to n - (k * m)*. In this case, the normal search window moves from one location to the next until it reaches *n - (k * m)*, resulting in a time complexity of *O(n)*.

The second part (lines 4-28) considers two cases: when $k = 1$ and when $k > 1$. In the first case, comparisons are made for each window, resulting in *m* comparisons per window. In the second case ($k > 1$), there are m / k comparisons per window. The overall complexity of this part is determined by the number of successful matches, '*nocc*,' with HT access having a constant complexity of *O(1)* as per Theorem 1. The second case occurs when all search attempts are mismatched. The operation will only access *HT* once in each *j* with a constant time according to Theorem 1. Furthermore, the other case involves comparing only a portion of p that does not result in a match. Each of these external loops is compared for each j value. The number of attempts required to find a match is less than *m* when $k = 1$ or less than *nocc*. Therefore, the maximum time complexity of the second part remains *O(nocc)*, as demonstrated in the second case.

Part three involves a detailed examination of each line of work. The loops and conditional (IF) functions have a constant factor of *O(1)* complexity because they only depend on configurable variables.

Therefore, the inner While loop operates at most *nocc* times, while the outer loop operates at most *n - (k * m)* times. The overall complexity is *nocc + (n - (k * m))* times, which is equal to *O(n + nocc)*.#

5 Experimental Results

The researcher developed a computer program using the Java language and conducted experiments on a 16-GB Intel Core i7 processor running Windows 10. The experiments involved using different random text as pattern characters and search data, with varying sizes for each.

Next, famous algorithms from the Handbook of Exact String Matching [7] were implemented: Brute Force (BF), Knuth-Morris-Pratt (KMP), Boyer-Moore (BM), Shift-Or (SO), Karp-Rabin (KR), and Quick Search(QS). Additionally, the PFIVL algorithm from [23] and the q-gram algorithms BNDM and BNDMq from [15] were implemented for comparison with the new solution, SMCIVL.

The size and quantity of random text were 100 bytes, 1 KB, 10 KB, 100 KB, 1 MB, and 10 MB. The length of characters (L) was 2, 4, 8, 16, 32, or 64 characters. The number of active characters (Σ) was 2 (binary), 4, 8, 16, 32, or 64 characters.

With respect to the amount of search text, the newly developed algorithm delivers faster search times when searching for a small amount of text, for example, 100 bytes. The search times match or closely approach those of traditional algorithms, especially when the search factor is large, ranging from 0.25 to 0.75 times the length of the pattern.

Considering the same hash algorithm, KR [7], and PFIVL [23], the new solution is expected to deliver better overall performance. Fig. 1 displays the results for scenarios with a small volume of text, a short pattern length, and a small Σ. Additional experimental results are presented in Fig. 2 for scenarios involving a large volume of text and a large Σ.

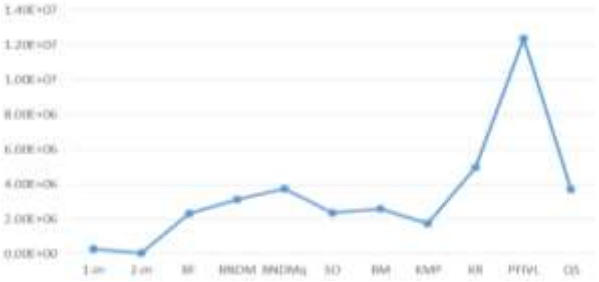

Fig. 1: Experimental results when Σ = 2, text size= 100 KB, and Length = 2.

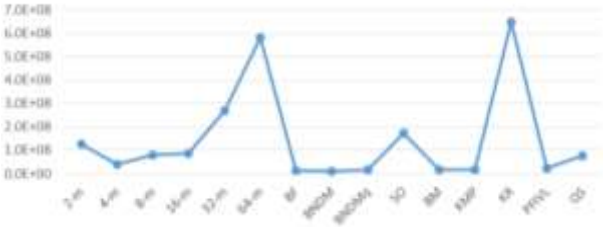

Fig. 2: Experimental results when Σ = 64, text size= 10 MB, and Length = 64.

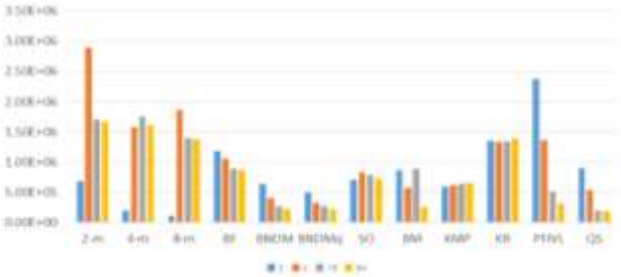

Fig. 3: Experimental results when Σ = 2, 4, 16, 64, text size= 10 KB, and Length = 8.

Furthermore, the real-world experiments were conducted using data obtained from three open sources:

1. DNA Sequences from the GenBank database (https://www.ncbi.nlm.nih.gov/genbank/) with a volume of 1.8 MB.
2. Data from the British National Corpus (BNC) word bank (BNC Consortium - OUP, Longman, UCREL, OUCS, Chambers) (http://ota.ox.ac.uk/desc/2553), which consists of 380,640 words.

Data from the American National Corpus (ANC) word bank (http://www.anc.org/data/masc/downloads/data-download/), which consists of 201,488 words.

The experiment measured the time taken to search real data (on the x-axis) in nanoseconds, as shown in Fig. 4 to 6. The experiments were conducted with ANC, BNC, and DNA data, varying the factors from 1-m to 64-m based on the character string length from 2 to 64 (on the y-axis).

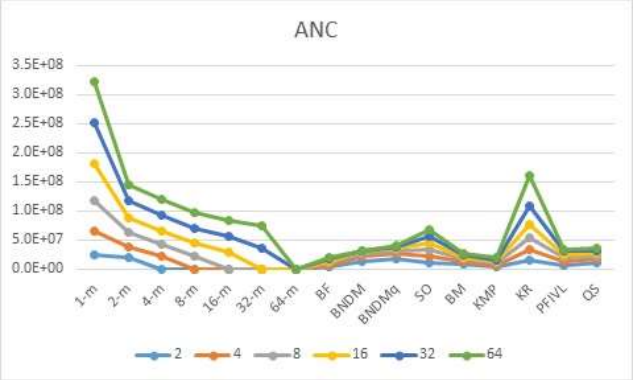

Fig. 4: Experimental results of ANC.

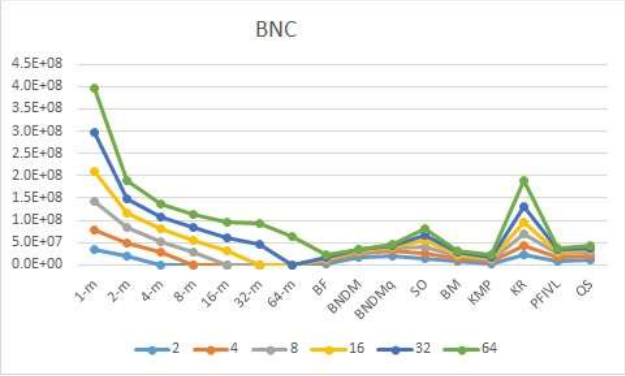

Fig. 5: Experimental results of BNC.

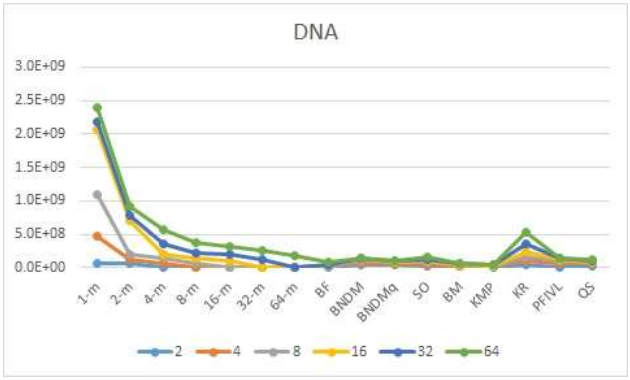

Fig. 6: Experimental results of DNA.

Figures 4-6 confirm that the new algorithm is faster than the original algorithm, especially when splitting the pattern into more than one character. This is particularly true for k-m values between 0.25 and 0.75 times the length of the pattern. In these cases, the new algorithm is at least as fast as the original algorithm, and often faster.

6 Discussions

The following discussion will analyze and evaluate complexity. It will examine experimental results, including character length and text quantity specifications, and compare them to algorithms with similar hashing designs presented in the article.

In evaluating method complexity, the pre-processing step demonstrates a superior time complexity of $O(m/k)$, surpassing other methods. Moreover, the search step exhibits a linear time similar to that of the KMP method, with a complexity of $O(n + nocc)$.

Based on the experimental results, the new method demonstrates excellent efficiency when the character set size is binary, particularly for scenarios with small character counts and short lengths. However, as the character set size increases to 4, 16, and 64, the new method exhibits consistent search speeds due to unchanged values in the mismatch table. The primary factor influencing performance is the length of the characters.

When examining the quantity of text employed for searches, it becomes clear that the recently devised method demonstrates superior or nearly equal search performance in comparison to the conventional method. This advantage is particularly noticeable when adjusting the search factor parameters (character splitting) to encompass larger values, spanning from 0.25 to 0.75 times the character length.

In contrast to methods employing similar hash table construction principles, such as KR and PFIVL, the novel approach generally demonstrates superior efficiency. This advantage stems from its utilization

of character subdivision into factors, resulting in a relatively faster search process.

In scenarios where search performance is evaluated with longer characters and larger text volumes, the new algorithm's search outcomes do not consistently match those of other algorithms. This discrepancy arises from the need for continuous analysis of longer characters in each search operation, resulting in a higher frequency of analyses. Furthermore, the actual search process can be influenced by computer programming, as character segmentation based on factor values directly affects search time. Additionally, it's worth noting that the new algorithm lacks a shift table, unlike algorithms such as BM or KMP, which allows for shifting the search window by more than one character.

7 Conclusion

This research introduces a novel data structure known as multi-character inverted lists (*m-cIVL*) designed for searching string data. The development of this structure enhances the ability to compare character strings, going beyond the limitations of one-to-one character matching. Theoretical findings suggest that this newly created data structure exhibits a time complexity of $O(m / k)$ for constructing string matches, a worst-case space complexity of $O(|\Sigma| + m)$, and an average-case space complexity of $O(2(m / k))$. The search algorithm operates with a time complexity of $O(n + nocc)$. In experimental evaluations, this new solution proves to be a good fit and performs efficiently, especially when utilizing a binary character set with a limited number of characters and shorter lengths. It is noteworthy that this algorithm adapts well to longer patterns and outperforms other approaches in terms of text search performance.

8 Suggestions for future work

Although the new algorithm is effective, there is room for improvement, especially in algorithms for finding or comparing pairs, which only require a single shift of the search window. Developing strategies or algorithms that allow multiple characters to be scrolled simultaneously would significantly improve the efficiency of the search process. Additionally, the following suggestions could be considered to further improve the algorithm's efficiency and open up new avenues for future research:

1. Develop processes for generating shift tables that accommodate multiple characters in the search window.
2. Optimize the search direction, including exploring reverse searches within the search window.
3. Utilize simultaneous access factors across various segments of a character pattern during searches.
4. Investigate techniques to expedite the verification of the inverted lists continuity.

Acknowledgement:
Special thanks to Ramkhamhaeng University for providing funding for this research. Extensive gratitude goes to the Department of Computer Science for their support, including the provision of necessary time and research space. Finally, appreciation is extended to Assoc. Prof. Dr. Veera Boonjing for collaborating on the creation and expansion of the inverted lists in the initial version of this research structure.

References:

[1] Boyer R.S. and Moore J. S., A fast string searching algorithm, *Communications of the ACM*. *20*, pp. 762-772, 1997.

[2] Chrochemore M. and Handcart C., Automata for Matching Patterns, *Handbook of Formal Languages, Volume 2, Linear Modeling: Background and Application, G. Rozenberg and A.Salomaa ed.,Springer-Verlag,Berlin.*, Ch. 9, pp. 399-462, 1997.

[3] Chrochemore M., Off-line serial exact string searching, *Pattern Matching Algorithms, A. Apostolico and Z. Galil ed., Oxford University Press*. Chapter 1, pp. 1-53, 1997.

[4] Crochemore M., Gasieniec L., and Rytter W., Constant-space string-matching in sublinear average time, *Compression and Complexity of Sequences 1997*, pp. 230 – 239, 1997.

[5] Monz C. and Rijke M. de. (2006, August, 12) *Inverted Index Construction*. Available: http://staff.science.uva.nl/~christof/courses/ir/transparencies/clean-w-05.pdf.

[6] Escardo M., (2008, October 15), *Complexity considerations for hash tables* Available: http://www.cs.bham.ac.uk/~mhe/foundations2/node92.html.

[7] Charras C. and Lecroq T.. (2008, October 10). *Handbook of Exact String Matching*. Available: www-igm.univ-lv.fr/~lecroq/string/string.pdf.

[8] Navarro G. and Raffinot M., *Flexible Pattern Matching in Strings*, The press Syndicate of The University of Cambridge., pp. 15-40, 2002.

[9] Galil Z., Giancarlo R., On the exact complexity of string matching upper bounds, *SIAM Journal on Computing*, 21(3)., pp. 407-437, 1992.

[10] Kesong H., Yongcheng W. and Guilin C., Research on A Faster Algorithm for Pattern Matching, *Proceedings of the fifth International workshop on Information retrieval with Asian languages*. 2000, pp. 119-124.
[11] wikipedia, (2020, November 15), *Hash table*. Available:http://en.wikipedia.org/wiki/Hash_table.
[12] Loudon K., (2020, November 24), *Hash Tables*. Available:www.oreilly.com/catalog/masteralgoc/chapter/ch08.pdf.
[13] DINH V. H., (2020, November 24), *Hash Table*. Available:http://libetpan.sourceforge.net/doc/API/API/x161.html.
[14] Law J., Book reviews: Review of Flexible pattern matching in strings: practical on-line algorithms for text and biological sequences by Gonzolo Navarro and Mathieu Raffinot, Cambridge University Press 2002". *ACM SIGSOFT Software Engineering Notes*, vol. 28 Issue 2, pp. 1-36, 2003.
[15] Navarro G., Raffinot M., Fast and flexible string matching by combining bit-parallelism and suffix automata, *December 2000 Journal of Experimental Algorithmics (JEA)*, Vol. 5, 2000.
[16] Knuth D.E., Morris J. R., and Pratt J. H., Fast pattern matching in strings, *SIAM Journal on Computing 6(1)*, pp. 323-350, 1977.
[17] Ager M. S., Danvy O. and Rohde H. K., Fast partial evaluation of pattern matching in strings, *ACM Transactions on Programming Languages and Systems (TOPLAS)*, No. 28 Issue 4, pp. 3-9, 2006.
[18] Ager M. S., Danvy O. and Rohde H. K., On obtaining Knuth, Morris, and Pratt's string matcher by partial evaluation, *Proceedings of the ASIAN symposium on Partial evaluation and semantics-based program manipulation*, pp. 32-46, 2002.
[19] Morris J. R., and Pratt J. H., A linear pattern-matching algorithm, *Technical Report 40, University of California, Berkeley*. 1970.
[20] Zaïane O. R. (2001, September 15), "CMPUT 391: Inverted Index for Information Retrieval", University of Alberta. Available: http://www.cs.ualberta.ca/~zaiane/courses/cmput39-03/.
[21] Yates R. B. and Neto B. R., Mordern Information Retrieval, *The ACM press. A Division of the Association for Computing Machinery, Inc*, pp. 191-227, 1999.
[22] Simon I., String matching and automata, *Results and Trends in Theoetical Computer Science, Graz, Austria, J. Karhumaki, H. Maurer and G. Rozenerg ed., Lecture Notes in Computer Science 814, Springer-Verlag, Berlin*, pp. 386-395, 1994.
[23] Khancome C. and Boonjing V. Inverted lists string matching algorithms, *International Journal of Computer Theory and Engineering* Vol.2 , No.3, pp.352–357, 2010.

Contribution of Individual Authors to the Creation of a Scientific Article (Ghostwriting Policy)

The author contributed in the present research, at all stages from the formulation of the problem to the final findings and solution.

Sources of Funding for Research Presented in a Scientific Article or Scientific Article Itself

Special thanks to Ramkhamhaeng University for providing funding for this research. Extensive gratitude goes to the Department of Computer Science for their support, including the provision of necessary time and research space. Finally, appreciation is extended to Assoc. Prof. Dr. Veera Boonjing for collaborating on the creation and expansion of the inverted lists in the initial version of this research structure.

Conflict of Interest

The author has no conflict of interest to declare that is relevant to the content of this article.