Available online at http://icsejournal.com/index.php/JCSE
Journal of Computer Science an Engineering (JCSE)
e-ISSN 2721-0251
Vol. 3, No. 1, February 2022, pp. 49-59

Performance study of the Memory Utilization of an Improved Pattern Matching Algorithm using Bit-Parallelism

Oladunjoye John Abiodun[1], Moses Timothy[2], Okpor James[3] and Baku Agyo Raphael[4]

[1,3,4]Computer Science Department, Federal University Wukari, Taraba State, Nigeria.
[2]Computer Science Department, Federal University Lafia, Nasarawa State, Nigeria.
E-mail: oladunjoye.abbey@yahoo.com[1], moses.timothy@science.fulafia.edu.ng[2], okporjames@fuwukari.edu.ng[3], bakuraph@fuwukari.edu.ng[4]

ARTICLE INFO

Article History:
Received December 04, 2022
Revised February 20, 2022
Accepted February 28, 2022

Keywords:
Bit-parallelism,
pattern matching,
string patter matching,
memory utilization,
improved pattern matching.

Correspondence:
E-mail:
oladunjoye.abbey@yahoo.com

ABSTRACT

The strategy of packing several data values in a single computer word and refreshing them all in a solitary operation is referred to bit parallelism. It assumes a significant part in pattern matching because it can handle in parallel the length of pattern sizes. In this paper, an Improved Pattern Matching model (IPM) proposed, which makes searching process quicker and decreases how much memory used in processing input data. C# was used for the development of the model. With a computer word size of 64bits and pattern length ranging from 8 characters to 72 characters, the system decides how much memory is used. The developed model was evaluated and contrasted with the existing model using 64bits computer word size (cws) and the pattern length of 72 characters. The assessment showed that the IPM had minimal worth of MU contrasted with the existing model (BNDM, SBNDM, and FSBNDM). This IPM model can be embraced for improvement of the size of string data stored in computer word because of its capacity to diminish memory space usage.

1. Introduction

Bit-parallelism is one of the main techniques in the field of computer science for the search of patterns in a given text [1]. Because of the length of pattern size that can be handled in parallel, bit parallelism assumes a significant part in pattern matching lately [2]. To achieve bit parallelism, you make bit vector *s* of the pattern length of characters, and afterward do a parallel match with the help of bit operation *s*. Test review, notwithstanding, shows that bit-parallelism performs better when contrasted with other character-based or non-bit parallel algorithms. However, it forces a restriction on the pattern size [3]. Notable algorithms that utilize the bit-parallelism strategy to find occurrences of patterns in a huge text are Shift-AND, Shift-OR, Shift-ADD, Backward Non-Deterministic matching calculation (BNDM) [4].

Pattern matching involves having a text or sequence and tracking down the occurrences of specific pattern of characters in it. Pattern matching is a vital strategy utilized in numerous applications such as spell checkers, Information retrieval frameworks, web indexes, copyright/similarity index detection, bioinformatics, content tools, and word processors. There are different sorts and characterizations of string matching algorithms. Two kinds exist; approximate/exact string matching and inexact string matching [5]. While inexact string matching searches specific occurrence of the pattern, inaccurate searching established on specific applications is permitted in approximate string matching [6]. String matching has two classifications in light of the number of patterns; multiple and single pattern string matching. While multiple string matching searches multiple patterns in a text, single string matching involves searching a solitary pattern in a text. String matching has four groupings in light of the order of searching; specific order matching, no order matching, right to left, and left to right matching [7]. The

http://dx.doi.org/10.36596/jcse.v3i1.460

pattern matching strategies are intending to further improve the search process and allow for quicker response [8]. This is possible by lessening the number of comparisons. As indicated by [9,10], improvements are made on certain algorithms on the search process and how comparisons are made between the considered text and the pattern. Other algorithms made enhancements on the processing stage, which is utilized to decide how much shift in the event that there is a mismatch between the pattern and the text [11,12]. As far as comparisons are concern, as indicated by [13], a few algorithms made correlations from the left side of the text by aligning one pattern to the text. Other algorithms made comparisons from the right side of the text by aligning one pattern to the text [14]. In term of sliding window, some algorithms utilize two sliding windows to make comparisons utilizing two patterns; one pattern lined up with the text from the left, while from the right side, the subsequent pattern lined up, with the two correlations happening at the same time [15, 16]. Enhancements have likewise been made on memory usage. Memory usage is the complete space taken by the algorithm with regard to the input data. This improvement is simply critical to diminish the memory used to manage the shift values in the pre-processing stage. Other enhancements have been made to the strategy for the search process; to make it quicker in some algorithms either expanding the shift value that the pattern should move in situation where there is a mismatch between the pattern and the text or by involving an alternate strategy for the search process [17,18].

The algorithm that utilizes bitwise tasks is known as the bit-parallelism algorithm [19]. The shift-AND algorithm reproduces the behaviour of the nondeterministic string matching automation that recognizes from a given text, pattern *p* of length *m*. It is a solitary pattern string matching algorithm. Shift-AND algorithm is quick and simple to carry out. The Shift-OR algorithm utilizes the corresponding procedure of the Shift-AND algorithm. It utilizes the shift-OR strategy for matching the pattern. Specifically, an active state of the automation is addressed with zeros (0s) bit while one (1s) addresses the non-active state. The algorithm is for both single and multiple pattern matching [20]. Backward Non-Deterministic Matching (BNDM) algorithm was proposed by [4] as a single pattern exact string matching algorithm. BNDM is a simulation of Backward Deterministic matching with the utilization of bit-parallelism. BNDM matching idea depends on a shift-OR calculation and suffix automata from the Backward Deterministic matching strategy. In BNDM, we can check a given text from right to left in a given pattern length. BNDM involves two stages; the preprocessing stage and searching stages. In the preprocessing stage, the algorithm observes bit vectors of each character that will conceivably come in the text. While in the search stage, the pattern is looked with the assistance of shift-AND operation. BNDM algorithm is quicker than the previous algorithms: shift-OR, shift-AND, BDM, and other character based algorithms [6]. It is consuming next to no space and performs different activities in parallel. It is an extremely straightforward and adaptable algorithm. Every one of the patterns is thought to be not exactly or equivalent to the computer word size [4]. BNDM algorithm was modified by [21] into a simplified version of BNDM algorithm. This version of BNDM is the very BNDM algorithm with the exception that it has a few changes from ascertaining the shift value in the loop; this makes the algorithm to skip the examination of the longest prefixes. In SBNDM, the average length of the shift is diminished. Despite this fact, the innermost loop of the algorithm becomes more straightforward prompting better execution in practice. SBNDM is quicker and consumes less space in correlation with past bit-parallelism algorithms. Another variation of BNDM algorithm was presented by [1] called Forward Simplified Backward Non-Deterministic Matching (FSBNDM) algorithm, which is the bit-parallel rendition of the Forward-BOM algorithm. To consider character next to the right of the present window of the text, FSBNDM employs non-deterministic automation (*NDaWg (p))* augmented of a new initial state. The outcome automaton (*m+1)* unmistakable states need *m+1* bits to be addressed. A bit vector of size *m +1* is instated in the preprocessing stage for each character. The i^{th} bit is one (1) in this vector in the event that *c* happens in the reversed pattern in the position i – 1; otherwise, it is 0. The bit

http://dx.doi.org/10.36596/jcse.v3i1.460

of location 0 is generally set to 1. FSBNDM algorithm ends up being quicker in practical cases especially for small alphabets and short patterns. As far as memory usage is concern, FSBNDM consumes less space contrasted with other variants of BNDM. One more variation of BNDM called Backward Non-deterministic matching algorithm with *q* gram was proposed by [22]. The BNDMq is likewise founded on the BNDM algorithm and it can likewise reproduce by bit-parallelism. In BNDMq, the q characters are read at every alignment prior to testing the state variable. The loop in this algorithm has been made to rapidly move *m-q+1* position, where *q* can be differed by our prerequisite and *m* is the length of the pattern. The creator of this algorithm improves the search process and makes it quicker. This paper proposes a superior pattern matching algorithm utilizing bit-parallelism to minimize the amount of memory space utilized in computer word and increment the size of string data stored in a computer memory.

IPM was developed to reduce the memory space utilized in searching for a pattern in a text. There are five sections in this paper. The first section is the introduction and the second reviews related literatures. Detailed design and method used were clearly described in the third section. The fourth section explains the performance evaluation between the developed model and the existing model. Section five is the conclusion.

2. Method

2.1. The design of IPM algorithm

Two phases are involved in the developed model, which are the pre-processing phase and the searching phase. An alignment matrix was worked on in the pre-processing stage by shifting the window of pattern *m* over the text. It then processes the shift *ch* to move values and mask the matrix of input pattern. To do this, it encodes the alignment matrix in binary numbers. This now serves as input to the next stage (the searching stage). The position of the pattern in a given text is worked on by the searching stage utilizing bitwise AND operations.

2.2. Pre-processing Stage

At this stage, three fundamental steps which are alignment matrix, shift value and the mask matrix are utilized.

2.1.1. The Alignment Matrix

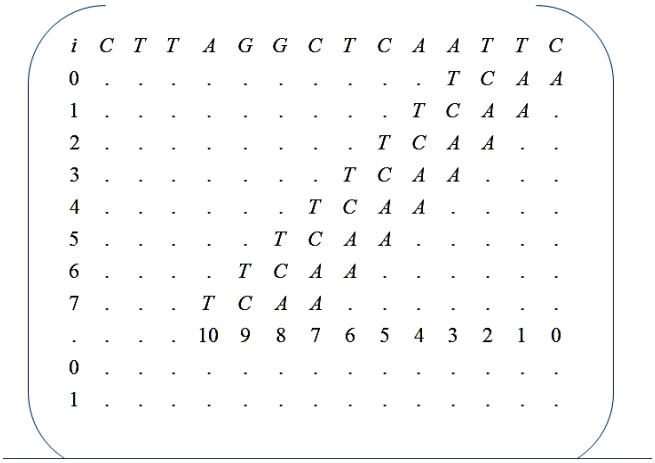
Rows and columns were created in this step. The row represents the computer word size *cws* while the window size is represented in the column. A pattern left-shifted by *i* characters ($0 \leq i \leq cws$) was carried out on each row while the column has characters of the pattern. To execute this algorithm, a window of length m was shifted over the text (where *m* is the quantity of characters in the pattern).This means that the algorithm searches the pattern by examining the present window from right to left for every window alignment. While implies that the algorithm peruses the window in reverse order as portrayed in Figure 1.

Algorithm A: Procedures for executing alignment matrix (P, m, T, n) where P = pattern, m = length of the pattern, T = text and n = length of text.

```
Start
Import Tn, P
Get m, n
for i = 1 to Tn
        Read T(i)
for j = 1 to Pn
        Read P(j)
```


http://dx.doi.org/10.36596/jcse.v3i1.460

```
Shift = n
Get cws
Ws = cws + m – 1
While (shift >= Ws)
Shift = Getwindows (shift, Ws , Tn , Pn )
Stop
```



i	*C*	*T*	*T*	*A*	*G*	*G*	*C*	*T*	*C*	*A*	*A*	*T*	*T*	*C*
0	.	.	.	.	.	.	.	.	.	.	*T*	*C*	*A*	*A*
1	.	.	.	.	.	.	.	.	.	*T*	*C*	*A*	*A*	.
2	.	.	.	.	.	.	.	.	*T*	*C*	*A*	*A*	.	.
3	.	.	.	.	.	.	.	*T*	*C*	*A*	*A*	.	.	.
4	.	.	.	.	.	.	*T*	*C*	*A*	*A*	.	.	.	.
5	.	.	.	.	.	*T*	*C*	*A*	*A*	.	.	.	.	.
6	.	.	.	.	*T*	*C*	*A*	*A*	.	.	.	.	.	.
7	.	.	.	*T*	*C*	*A*	*A*	.	.	.	.	.	.	.
.	.	.	.	10	9	8	7	6	5	4	3	2	1	0
0	.	.	.	.	.	.	.	.	.	.	.	.	.	.
1	.	.	.	.	.	.	.	.	.	.	.	.	.	.

Figure 1: Results showing alignment matrix of developed algorithm using DNA sequence

Where *T = C T T A G G C T C A A T T C (column)*
m = T C A A (column)
cws i = 0 1 2 3 4 5 6 7 (row)
W_S *= cws + m-1*
= 8+4-1
= 11

Window Size Length

To know the number of the shift in searching a specific pattern in a given text or sequence, W_s (window size length) was utilized. This was accomplished utilizing eqn(1).

$$W_s = cws + m - 1 \qquad eqn(1)$$

Where
window size length = W_s
computer word size = *cws*
number of character in a pattern = *m*

2.2.2. The Shift Value

The idea of shift instrument in the quick search algorithm developed by [23] was used to for this step. To get the shift value, the developed algorithm uses the left side of the sliding window to quickly look at the characters in the specified text. The shift value is determined by shift $[ch] = W_s - x$ whenever a character is included in the pattern. In this case, x represents the outermost position of the character in the pattern. However, the shift value is determined by shift $[ch] = W_s + 1$ if otherwise; the character is then rejected in the pattern. Whenever the character *(ch)* is observed at the position *(pos)*, for each character visited, the shift value should be selected as the maximum value of the current value of the

http://dx.doi.org/10.36596/jcse.v3i1.460

character visited. Below are algorithms A and B that shows the window size (shift, W_s, T_n, P_n) and shift value (W_s, P_n, T_n, C_h) respectively.

Algorithm B: Getwindows (shift, W_s, P_n, T_n)

```
Start
        for k = 0 to (W – 1)
                {
                        for c = 1 to m
                                print P(c)
                        shift = shift – 1
                }
        Get mask matrix (shift, Ws, Pn, Tn)
        Shift = Getshift value (Ws, Pn, Tn, Ch)
        Stop
```

Algorithm C: Getshiftvalue (Ws, Pn, Tn, Ch)

```
Start
for k = 1 to m
If P(k)  = ch
                {
R = m – k
Return r
                }
Stop
```

3.1.3 The Mask Matrix

There are *w* rows in the matrix. The bit vector of *cws* is represented by *w* (i.e. *bw_1bw_2 ... b2b1b0*) and window size column (W_s). Each row represents characters of the pattern while characters of the text are represented in the column. Binary numbers set at 0 and 1 as displayed in Figure 2 are used to encode the mast matrix. The binary number 0 is set for the i^{th} bit in the mask[character][position] of the patterns and the given text. Whenever 'we see that the character *[ch]* of the text at position *(pos)* is fundamentally not the same as the character *i*, the left-shifted alignment of *p* (the input pattern) is executed. In any case, it is comparable to 1, which shows that the character *(ch)* of the text at this position *(pos)* is the same *i* character left-shifted version of the pattern. Algorithm D shows the procedures expected to achieve the Mask matrix (Shift, W_s, P_n, T_n)

A lgorithm D: Getmaskmatrix (Shift, Ws, Pn, Tn)

```
Start
for i = 0 to (W- 1)  {
for k = 0 to (Ws – 1) {
        If Ws(k) = Empty
print 1 If Ws(k) = Tn (k)
print 1 If Ws(k) ≠ Tn (k)
print 0
        }
    }
Stop
```


http://dx.doi.org/10.36596/jcse.v3i1.460

3.2. Searching Stage

As shown in Figure 3, the algorithm at this stage observes the position of the pattern with the help of data collected during the pre-processing stage.

a. Determination of bit value of each character letter for each window size

To determine the bit value of every alphabet for every window size, mask matrix was encoded with binary numbers in the pre-processing stage. To achieve this, the column of every alphabet with relating bit vectors was checked.

b. Determination of first bit values of the indicator

The pointer addresses the current condition inside the computer system on which the decision is based. The first bit value of the pointer corresponds to the first bit value of the first alphabet of the window size determined during the preprocessing stage.

i	*C*	*T*	*T*	*A*	*G*	*G*	*C*	*T*	*C*	*A*	*A*	*T*	*T*	*C*
0	.	.	.	1	1	1	1	1	1	1	0	0	0	0
1	.	.	.	1	1	1	1	1	1	0	0	0	0	1
2	.	.	.	1	1	1	1	1	0	0	1	0	1	1
3	.	.	.	1	1	1	1	1	1	1	1	1	1	1
4	.	.	.	1	1	1	0	0	0	1	1	1	1	1
5	.	.	.	1	1	0	1	0	0	1	1	1	1	1
6	.	.	.	1	0	0	0	0	1	1	1	1	1	1
7	.	.	.	0	0	0	0	1	1	1	1	1	1	1
.	.	.	.	10	9	8	7	6	5	4	3	2	1	0
0	.	.	.	.	.	.	.	.	.	.	.	.	.	.
1	.	.	.	.	.	.	.	.	.	.	.	.	.	.

Figure 2: The IPM algorithm for the implementation of mask matrix

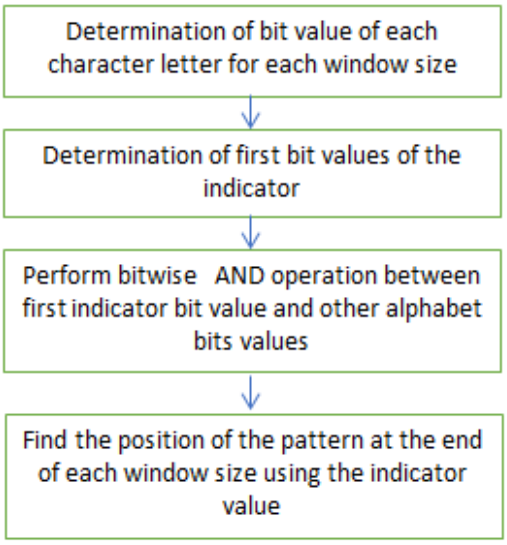


Figure 3: The IPM algorithm showing hierarchy diagram of the searching phase for finding a pattern in a given string using bit-parallelism

http://dx.doi.org/10.36596/jcse.v3i1.460

c. Performing bitwise AND operation between First Indicator Bit Value and other Alphabet Bits Value

To obtain the second pointer value, the method performs bitwise AND operations between the first-bit value of the pointer and the second-bit value of the second letter obtained from the mask matrix of the window size. This cycle is repeated for the following letter in order to obtain the next pointer value until the bit value of the alphabet of the window size has been processed.

d. Finding the position of the pattern at the end of each window size using Indicator Value

The bit values of the pointer can be used to calculate the pattern's location. The bits in the pointer that are set to 1 after the inspection on the current text window depict which pattern from the set is observed at certain points. Assuming the pointer becomes 0 at the end of the window size under consideration, this indicates that the pattern does not exist on the window. Algorithm E describes the procedures involved in the searching state (W_S , T_n)

```
Searching Phase (WS , Tn )
Start
for i = 1 to Tn
    {
for j = (Ws – 1) to 0
Ch(j) = Tn (j)
    }
for k = (Ws – 1) to 0
          {
Msk(k) = Ws(k)
If k = (W – 1)
Ind(k) = msk(k)
else
Ind(k) = msk(k + 1) & Ind(k)
          }
// Finding the position of the pattern //
for x = 0 to (Ws + 1)
a = 0;
for y = 0 to (W-1)
If Ind(y) = 1
a = a + 1
If a = Ws
Pattern detected at T[x]
Stop
```

4. Result and Discussion

4. Experimental setup performance evaluation

4.1. Setup

The created method was tested alongside various existing algorithms, including BNDM, SBNDM, and FSBNDM, using DNA characters as a test-bed. In the C# programming language, an execution code was written. For this experiment, an Intel(R) Pentium(R) CPU B960 running at 2.2GHz, 4.00GH main memory, and the 64bits Microsoft Windows 7 operating system were employed.

4.2. Performance Evaluation

Memory Usage (MU) was used as a measure to examine the existing algorithms (BNDM, SBNDM, and FSBNDM) as well as the new (IPM) method. Figures 4–7 demonstrate the memory utilization of the

http://dx.doi.org/10.36596/jcse.v3i1.460

known methods and the proposed approach using a 64bit computer word size. Table 1 summarizes the assessment outcomes. The algorithms were evaluated utilizing DNA characters that constituted a 150-character text and character pattern lengths ranging from 8 to 72 characters.

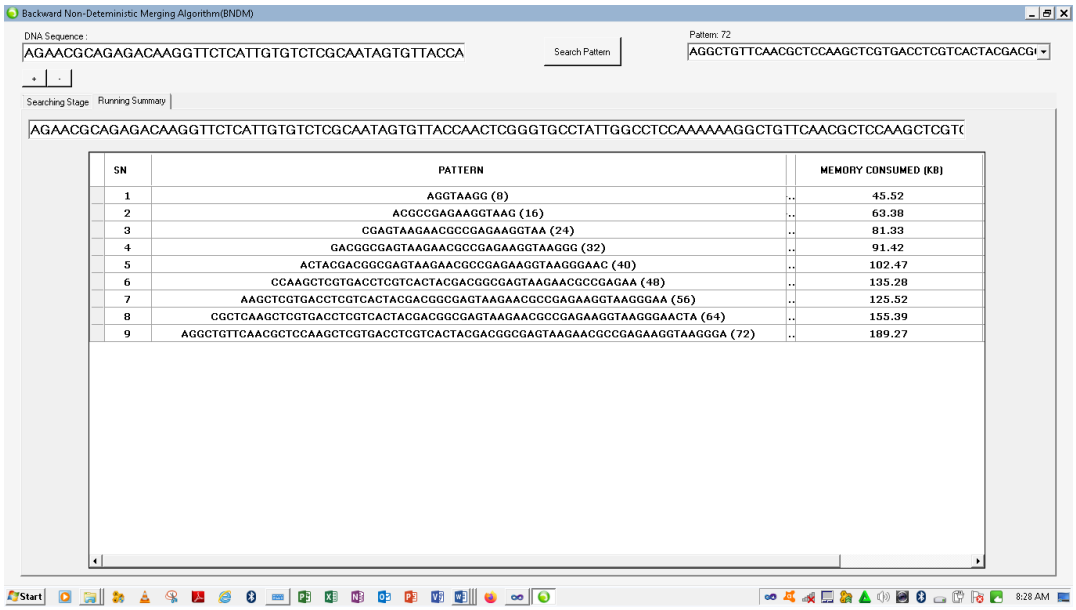


Figure 4: The outcomes of running BNDM

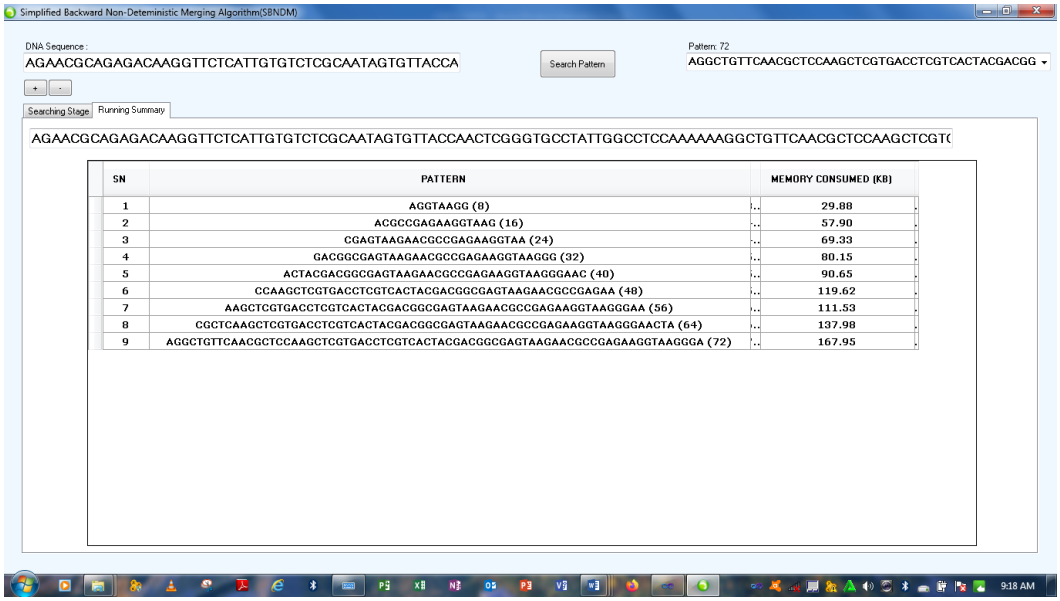


Figure 5: The outcomes of running SBNDM

http://dx.doi.org/10.36596/jcse.v3i1.460

 e-ISSN 2721-0251

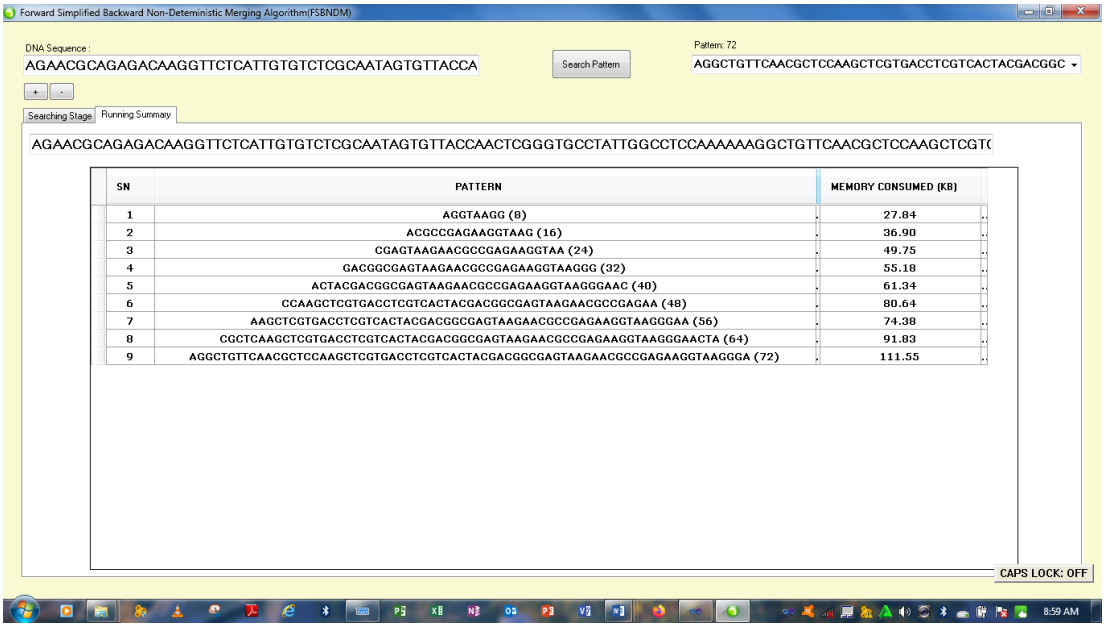


Figure 6: The outcomes of running FSBNDM

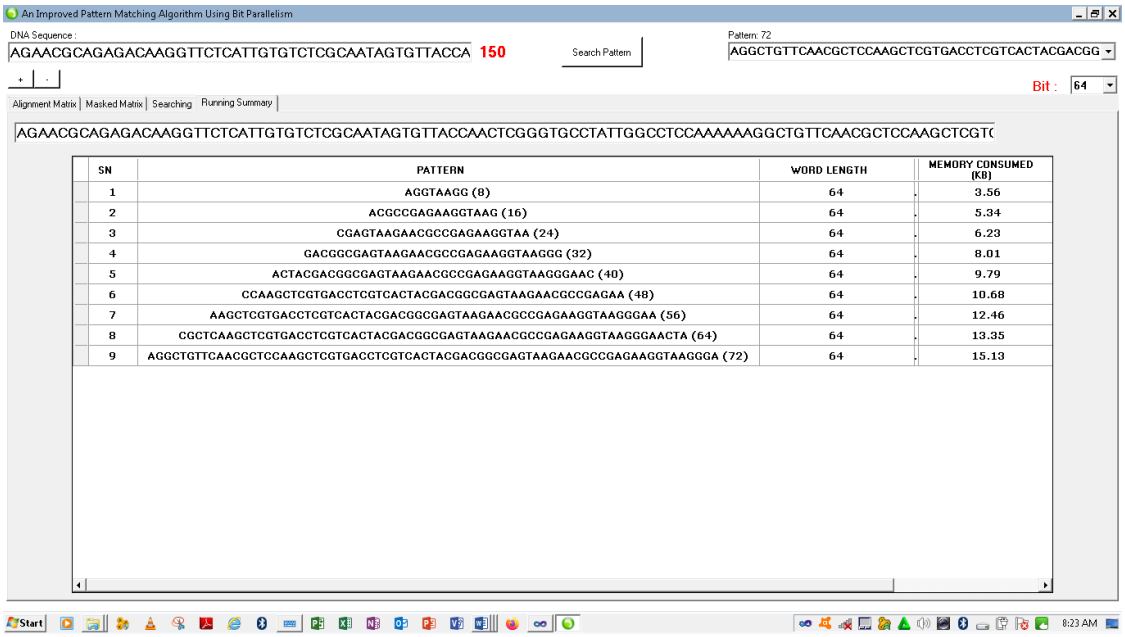


Figure 7: The outcomes of running the developed algorithm (IPM)

Table 1: Performance evaluation using memory utilization (in kb)

PATTERN SIZE	DEVELOPED(IPM)	BNDM	SBNDM	FSBNDM
8	3.56	45.52	29.88	27.84
16	5.34	63.38	57.90	36.90
24	6.23	81.33	69.33	49.75
32	8.01	91.42	80.15	55.18
40	9.79	102.47	90.65	61.34

http://dx.doi.org/10.36596/jcse.v3i1.460

48	10.68	135.28	119.62	80.64
56	12.46	125.52	111.53	74.38
64	13.35	155.39	137.98	91.83
72	15.13	189.27	167.98	111.55

Table 1 shows how much memory (in kb) is used by the created method and existing techniques for varying lengths of patterns in DNA sequences. The lengths of the patterns used range from 8 to 72 characters. The values in the table clearly illustrate that the proposed IPM method uses less memory than current techniques for any pattern length in a given text. According to the table, the memory use for the IPM method for a 72-character pattern is 15.13kb, which is less than that of the existing approaches. This suggests that the proposed technique required less memory than other algorithms to discover a pattern. The DNA data set that was utilized to assess our created technology is adequately enormous, and enough to conclude that IPM is efficient in terms of memory utilization that the existing algorithms.

5. Conclusion

This paper proposed a superior pattern matching algorithm utilizing the bit-parallelism strategy to enhance memory usage which limits the amount of memory utilized in computer word and to expand the size of string data stored in computer memory. The results obtained from the experiment shows that IPM out-perform existing algorithms in terms of memory utilization in a computer word, when searching pattern in a given text. This shows that IPM is a better algorithm to reduce memory usage in any computer word as little as could really be expected

References

[1] Faro, S., and Lecroq, T. The Exact Online String Matching Problem: A Review of the most Recent Results". *ACM Computing Surveys*, 2013; 45,2, 1-42.
[2] Gulfishsan, F.A., and Nilay, K. String matching algorithms using bit parallelism. *International Journal of Advanced Engineering and Global Technology*, 2014; 2, 5, 667-671.
[3] Alina, G. Bit parallel string matching. *International Journal of Soft Computing and Engineering*, 2006; 215-224.
[4] Navarro, G., and Raffinot, M. Fast and flexible string matching by combining bit-parallelism and suffix automata. *Journal of Exp. Algorithmics* (JEA), 2000; 5, 4, 1-36.
[5] Nimisha, S., and Deepak, G. String matching algorithms and their Applicability. *Application International Journal of Soft Computing and Engineering* (IJSCE), 2012; 1, 6, 2231-2307.
[6] Kapil K.S, Rohit V, and Vivek S. Efficient string matching using bit-parallelism. *International Journal of Computer Science and Information Technologies*, 2015; 6, 1, 265-269.
[7] Alberto Apostolico and ZviGalil. Pattern matching algorithms. Oxford University Press, USA, 1st edition, May 29, 1997.
[8] Hudaib, A., Suleiman, D. and Awajan, A. Fast pattern matching algorithm using changing consecutive characters. *Journal of Software Engineering and Applications*, 2016; 9, 399-411.
[9] Suleiman, D., Hudaib, A., Al-Anani, A., Al-Khalid, R. and Itriq, M. ERS-A Algorithm for Pattern Matching. *Middle East Journal of Scientific Resarech*, 2013; 15, 1067-1075.
[10] Pendlimarri, D. and Petlu, P.B.B. Novel pattern matching algorithm for single pattern matching. *International Journal on Computer Science and Engineering*, 2010; 2, 2698-2704.
[11] Hudaib, A., Al-Khalid, R Al-Anani, A., Itriq, M. and Suleiman, D. Four Sliding Windows Pattern Matching Algorithm (FSW). *Journal of Software Engineering and Applications*, 2015; 8, 154-165.
[12] Faro, S. and Kulekci, M.O. Fast Packed String Matching for Short Patterns. 2012; ArXiv:1209.6449v1[cs.IR].

http://dx.doi.org/10.36596/jcse.v3i1.460

[13] Berry, T., and Ravindran, K. A fast string matching algorithm and experimental results. *Proceedings of the prague Stringology Club Workshop '99'Collaborative Report DC-99-05, Czech Technical University, Prague*, 2001; 16-26.
[14] Senapati, K.K., Mal, S. and Sahoo, G. RS-A Fast Pattern Matching Algorithm for Biological Sequences. *International Journal of Engineering and Innovative Technology (IJEIT),* 2012; 1, 116-118.
[15] Suleiman, D. Enhanced Berry Ravindran Pattern Matching Algorithm (EBR). *Life Science Journal*, 2014; 11, 395-402.
[16] Faro, S. Efficient variants of the Backward-Oracle-Matching Algorithm. *International Journal of Foundations of Computer Science*, 2009; 20, 967-984. http://dx.doi.org/10.1142/S0129054109006991
[17] Salmela, L., Tarhio, J. and Kalsi, P. Approximate Boyer-Moore String Matching for Small Alphabets. *Algorithmica*, 2010; 58, 591-609. http://dx.doi.org/10.1007/s00453-009-9286-3
[18] Suleiman, D., Hudaib, A., Al-Anani, A., Al-Khalid, R. and Itriq, M. ERS-A Algorithm for Pattern Matching. *Middle East Journal of Scientific Resarech*,2013; 15, 1067-1075.
[19] Farro, S. and Lecroq, T, Twenty Years of Bit-parallelism in String Matching, 2000.
[20] Rajesh, P. ,Suneeta, A., Ishadutta, Y. and Bharat, S. Efficient Bit-Parallel Multi-Patathens String Matching Algorithms for Limited Expression" *ACM*, 2010; J an. 22-23.
[21] Peltola, H., and Tarhio, J. Alternative algorithms for bit-parallel string matching. *In M.A Nascimento, E. Silvad e Moura, and A.L Oliveira, editors, proceedings of the 10th International Symposium on string processing and Information Retrieval SPIRE*, 2003; 2857, 80-94. Manaus, Brazil, 2003. Springer-Verlag, Berlin.
[22] Faro, S., and Lecroq, T. Efficient variations of the Backward-Oracle-Matching algorithm. *In Holub Jan and Zdarek Jan, editors, Proceedings of the Prague stringology conference*, 2008; 146-160, Czech Technical University in Prague, Czech Republic.
[23] Sunday, D.M. A Fast Substring Search Algorithm. *Communications of the ACM,* 1990; 33, 8, 132-142.

http://dx.doi.org/10.36596/jcse.v3i1.460