Algorithms for String Comparison in DNA Sequences

Conference Paper · July 2019

DOI: 10.1007/978-981-13-7564-4_29

3 authors:

Dhiman Goswami
Daffodil International University
2 PUBLICATIONS **0** CITATIONS
SEE PROFILE

Nishat Sultana
Daffodil International University
2 PUBLICATIONS **0** CITATIONS
SEE PROFILE

Warda Ruheen Bristi
Daffodil International University
4 PUBLICATIONS **3** CITATIONS
SEE PROFILE

Chapter 29
Algorithms for String Comparison in DNA Sequences

Dhiman Goswami, Nishat Sultana and Warda Ruheen Bristi

1 Introduction

In context of working in the diversified field of computational biology, string comparison is a key concept applied to determine different properties of DNA or genome. Here three proposed improvements of the existing three approaches is discussed and named as SSAHA for exact or partial match finding in sequences of DNA database, finding MUMmer using BWT and FM index and improvement in the tool Sibelia to find synteny block.

SSAHA (Sequence Search and Alignment by Hashing Algorithm) [9], for performing efficient searching procedure on very large DNA database, sequences are preprocessed by dividing them consecutively in k-tuples of bases. Hash table method is used for the purpose of storing the positions of each k-tuple occurred. We search in the database of query sequences by retriving "hits" for each query sequence's every k-tuple from the hash table and then find the sorted order of the results. Tuple length k has the impact on the speed of searching, usage of memory and algorithm's sophistication. A dynamic programming approach is proposed to solve this problem which will give faster output for lower k-mers.

D. Goswami (✉) · N. Sultana · W. Ruheen Bristi
Department of Computer Science and Engineering, Daffodil International University, Dhaka 1207, Bangladesh
e-mail: dhimangoswami17@gmail.com

N. Sultana
e-mail: nishatsultana241@gmail.com

W. Ruheen Bristi
e-mail: wardaruheen39@gmail.com

D. Goswami · N. Sultana · W. Ruheen Bristi
Department of Computer Science and Engineering,
Bangladesh University of Engineering and Technology, Dhaka 1000, Bangladesh

M. S. Uddin and J. C. Bansal (eds.), *Proceedings of International Joint Conference on Computational Intelligence*, Algorithms for Intelligent Systems, https://doi.org/10.1007/978-981-13-7564-4_29

MUMs are the Maximal Unique Matches between two genomes of different organisms/species. It shows the common sub-sequences between two genomes and the sub-sequences must be unique in both the genomes. That is each MUM can be only once in each genome, as it is unique. Many significant algorithms are there to align two genomes. For example, Needleman and Wunsch (Global alignment) [8] and Smith and Waterman (Local alignment) [11] algorithms. In this paper, we propose an approach using BWT matrix [1] and FM index [5] to find MUMs by comparing two genomes by ensuring the uniqueness of the MUMs.

Synteny Block finding which is the undertaking of breaking down genomes into non-covering profoundly rationed portions has turned out to be essential in genome correlation. The motive is to explore synteny blocks.The genome sequences are closely related. Exploring blocks from various microbial genome sequences has some obstacles as they have less change in their characteristics. A general technique of building or generating graph from the existing sequence is using the de Bruijn graph. But as the real genome sequence has many hidden portions which can hide many characteristics. In this case, using the general version of de Bruijn graph could not capture those characteristics so in that case iterative de Bruijn graph is about to be implemented. As duplications are imprecise, de Bruijn graph structure needed an introductory stride of graph simplification to identify the accord of duplications, and next, fortifying the genome over the simplified graph to obtain the points of duplications.

2 Related Works

2.1 SSAHA

Formal Definition To determine the complete or specific portion's match of a query sequence Q in a subject sequences' database $D = \{S_1, S_2, \ldots, S_d\}$, where all the sequences in D is marked with a specific value i which is an integer used for the reference of index, the term k-tuple is used to represent a consecutive sequence of DNA nucleotides of length k. A DNA sequence S having length n includes (n–k+1) k-tuples [9] which overlaps in different positions. All the k-tuple's offsets within S is the positional difference as regards to the first base of S. We use $w_j(S)$ to express the k-tuple of S having offset value j. Thus, the location within D of every instance of all k-tuple is demonstrated by (i, j). Four types of bases can be mapped with two binary digits which can be shown in the following way: $f(A) = 00_2$, $f(C) = 01_2$, $f(G) = 10_2$, $f(T) = 11_2$. In this mapping procedure, 2k bit integer is needed to uniquely map any k-tuple $w = b_1 b_2 \ldots b_k$. So from the above binary representation we get the hash table position of a particular k-tuple by the following function: E(w) = $\sum_{i=1}^{k} 4^{i-1} f(b_i)$.

Hash Table Construction For example, we have database where we have three elements S_1, S_2, and S_3. Suppose, $S_1 = ACCGTTGTAGCTA$, $S_2 = ACCTTGTT$

Table 1 Hash table for k-mer value 1

W	E(W)	Positions												
A	0	(1,0)	(1,8)	(1,12)	(2,0)	(2,10)	(2,13)	(3,10)	(3,11)	(3,12)				
C	1	(1,1)	(1,2)	(1,10)	(2,1)	(2,2)	(2,8)	(2,14)	(2,15)	(3,0)	(3,6)			
G	2	(1,3)	(1,6)	(1,9)	(2,5)	(2,12)	(3,1)	(3,4)	(3,7)	(3,14)	(3,15)			
T	3	(1,4)	(1,5)	(1,7)	(1,11)	(2,3)	(2,4)	(2,6)	(2,7)	(2,9)	(2,11)	(3,2)	(3,3)	(3,5)	(3,8)	(3,13)

Table 2 Hash table query sequence for k = 1

t	$W_{t(Q)}$	Positions	H	M
0	A	(1,0)	(1,0,0)	(1,−2,0)
		(1,8)	(1,8,8)	(1,−1,0)
		(1,12)	(1,12,12)	(1,0,0)
		(3,12)	(3,11,12)	(1,6,8)
		(2,0)	(2,0,0)	(1,7,8)
		(2,10)	(2,10,10)	(1,8,8)
		(2,13)	(2,13,13)	(1,10,12)
		(3,10)	(3,10,10)	(1,11,12)
		(3,11)	(3,11,11)	(1,12,12)
1	A	(1,0)	(1,−1,0)	(2,−2,0)
		(3,12)	(3,12,12)	(2,−1,0)
		(1,8)	(1,7,8)	(2,0,0)
		(1,12)	(1,11,12)	(2,8,10)
		(2,0)	(2,−1,0)	(2,9,10)
		(2,10)	(2,9,10)	(2,10,10)
		(2,13)	(2,12,13)	(2,11,13)
		(3,10)	(3,9,10)	(2,12,13)
		(3,11)	(3,10,11)	(2,13,13)
2	A	(1,0)	(1,−2,0)	(3,8,10)
		(1,8)	(1,6,8)	(3,9,10)
		(1,12)	(1,10,12)	(3,9,11)
		(2,0)	(2,−2,0)	**(3,10,10)**
		(2,10)	(2,8,10)	**(3,10,11)**
		(2,13)	(2,11,13)	**(3,10,12)**
		(3,10)	(3,8,10)	(3,11,11)
		(3,11)	(3,9,11)	(3,11,12)
		(3,12)	(3,10,12)	(3,12,12)

$CTATGACC$ and $S_3 = CGTTGTCGTCAAATGG$ and we have a Target Sequence: AAA. So the hash table for the database for k-mer size equal to 1 (Table 1).

Now we observe the hash table of the query sequence in the following table and there the solution can be found from the bold indices (Table 2).

Now the approach is extended to k-mer value equal to 2 and the database is
$S_1 = ACCGTTGTAGCT$,
$S_2 = ACCTTGTTCTATGACC$ and
$S_3 = CGTTGTCGTCAAATGG$
and Target Sequence is $AAATGG$.
Hash table for the database for this case is (Table 3).

After that, we observe the hash table of the query sequence in the following table and there the solution can be found from the bold indices (Table 4).

Table 3 Hash table for k-mer value 2

W	E(W)	Positions		
AA	0	(3,11)		
AC	1	(1,1)	(2,1)	
AG	2	(1,9)		
AT	3	(2,11)	(3,13)	
CA	4			
CC	5	(2,15)		
CG	6	(1,3)	(3,1)	(3,7)
CT	7	(1,11)	(2,3)	(2,9)
GA	8	(2,13)	(1,8)	(1,12)
GC	9			
GG	10	(3,15)		
GT	11	(1,7)	(3,7)	
TA	12			
TC	13	(3,9)		
TG	14	(2,5)		
TT	15	(1,5)	(2,7)	(3,3)

Table 4 Hash table query sequence for k = 2

t	$W_{t(Q)}$	Positions	H	M
0	AA	(3,11)	(3,11,11)	(2,2,5)
1	AA	(3,11)	(3,10,11)	(2,9,11)
2	AT	(2,11)	(2,9,11)	(3,10,11)
		(3,13)	(3,11,13)	**(3,11,11)**
3	TG	(2,5)	(2,2,5)	**(3,11,13)**
4	GG	(3,15)	(3,11,15)	**(3,11,15)**

2.2 MUMs Finding

An established system is available to align the whole genome [3]. Two sequences as genome A and genome B act as input to the system. At first MUMmer is found out by suffix tree structure. All the suffixes are found out from both the genomes A and B where genome A is "gaacctca" and genome B is"aaaaccgtt", for example. Suffix tree is drawn from the suffixes of genome A. Later on suffixes of genome B is added to that tree. Figure 1 shows the suffix tree of genome A and genome B. By doing this the portion of suffixes which is common to both the genomes, MUMs, could easily be sorted out. In Fig. 1, red colored alphabets indicate the genome from which the string comes. After finding the MUMs, those are aligned by a variation of LIS algorithm. Lastly gaps among the MUMs are closed by four different processes by identifying many biological features like SNPs, large insertions, repeats, tandem repeats, etc. In this way, the whole genome is aligned (Table 5).

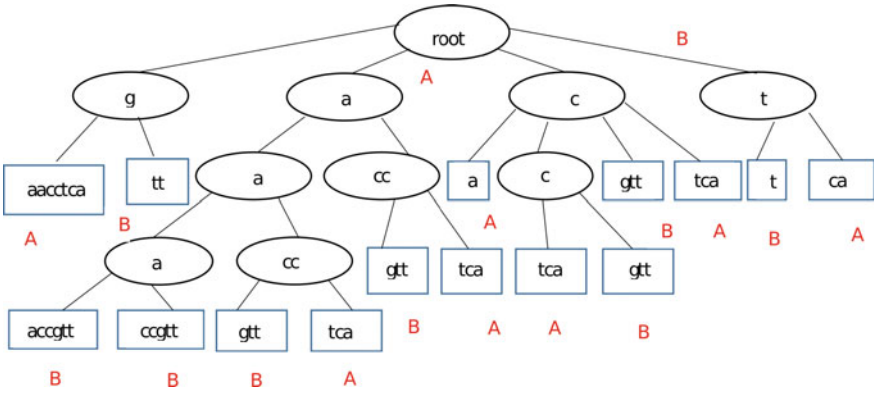

Fig. 1 Suffix tree on genome A and B

Table 5 Suffices of genome A and genome B

Genome A	Genome B
gaacctca	aaaaccgtt
aacctca	aaaccgtt
acctca	aaccgtt
cctca	accgtt
ctca	ccgtt
tca	cgtt
ca	gtt
a	tt
	t

2.3 Synteny Block Finding

Cycles in de Bruijn Graph Given String $S = s_1 \ldots s_n$ is roundabout which is a genome of length n. It is roundabout over the nucleotide letters in order A,G,C,T. A string having length k is denoted as k-mer. The de Bruijn graph diagram DB(S,k) speaks to each node in the graph that is defined as per the k-mer and associates both nodes by a coordinated arc on the off chance that they are compared to a couple of back to back k-mers in the genome sequence. The de Bruijn graph observed as a weighted multigraph where adjoining nodes can be associated by various arcs and with the assortment of an edge (a,b) defined as the occasions that the k-mers show up continuously in S. Cycles in de Bruijn diagrams are normally classified into two kinds [7]: swells (Fig. 2) and circles (Fig. 2). Instinctively, swells are formed by befuddles/indels between two homologous successions, and circles are caused by firmly found k-mer rehashes. To uncover rehashes in de Bruijn charts, these little cycles ought to be expelled. To maintain a strategic distance from the threading methodology or in other words, we receive a succession alteration way to deal

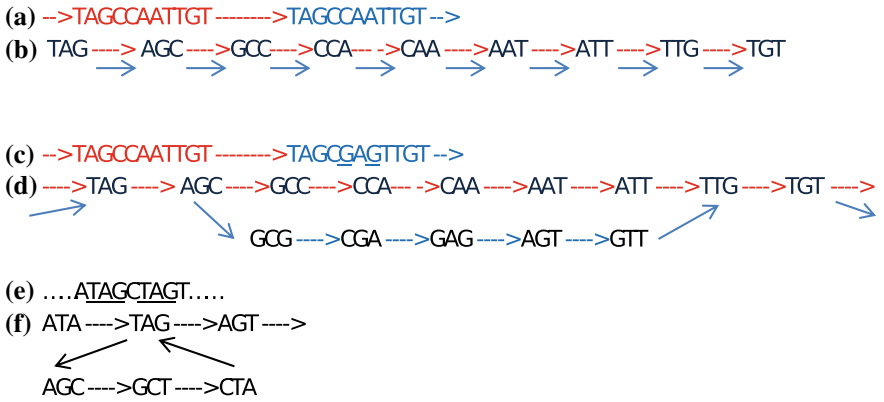

Fig. 2 Figure containing genome sequence de Bruijn graph generated from the sequence and cycles in the de Bruijn graph. **a** Genome sequence S = TAGCCAATTGT...TAGCCAATTGT. **b** The two branches of the de Bruijn graph is drawn using different color (red and blue) using the sequence provided in **a**. **c** A sequence having two different streams of strings the dissimilarities are underlined. **d** Having two different branching De Bruijn graph of the sequence in **c**. Small differences between these two portions (shaded red and blue) are reflected by a bulge with two branches (the red and the blue parts of the bulge). **e** An arrangement with a firmly found rehashed k-mer TAG. **f** The de Bruijn diagram of the succession in **e**. The firmly found rehashed k-mer (TAG) shapes a circle in the chart

with expel swells in the de Bruijn diagrams. In the following subsection, it will be discussed that Sibelia does not have to expressly expel circles, but rather concentrates just on bulges. The purpose for this is only expanding the estimation of k can dispense with little circles in the de Bruijn diagram. Figure 2c demonstrates a circle or in other words a firmly found 3-mer rehash (TAG). The de Bruijn graph diagram having greater vertex measure k = 4 does not produce a loop.

Sequence Modification algorithm Give C a chance to be a bulge with aggregate edge number smaller than C, framed by substrings P_1 and P_2 of S. To evacuate the bulge, the calculation changes S by substituting all events of P_1 in S by P_2. Figure 3 shows the de Bruijn graph for S = TAGCCAATTGT...TAGCGAGTTGT, with two repeats with smaller difference. Minor differences between these two recurrent occasions frame a bulge having two branches (red and blue in the figure). By changing S into S_0 = TAGCCAATTGT...TAGCGAGTTGT (i.e. selecting the red branch in lieu of the blue branch), the bulge is additionally simplified. Note that S_0 currently has a correct rehash of multiplicity 2.

Algorithm In Sibelia [7], Algorithm 1 is used to find synteny block. At first, the calculation builds the de Bruijn chart from a generally little estimation of $k = k_0$ and performs diagram simplification with a little cycle length limit (C_0). The calculation works on the de Bruijn diagram $G_0(S_0, k_0)$ and simplifies all bulges utilizing the grouping modification approach depicted previously. Accordingly, we get a simplified de Bruijn chart G1 and the comparing modified genome S_1. S_1 is a contorted adaptation of S with the end goal that its de Bruijn chart $DB(S_1, k_0)$ does not include short bulges of length tinier than C_0. We take note of that chart simplification ought

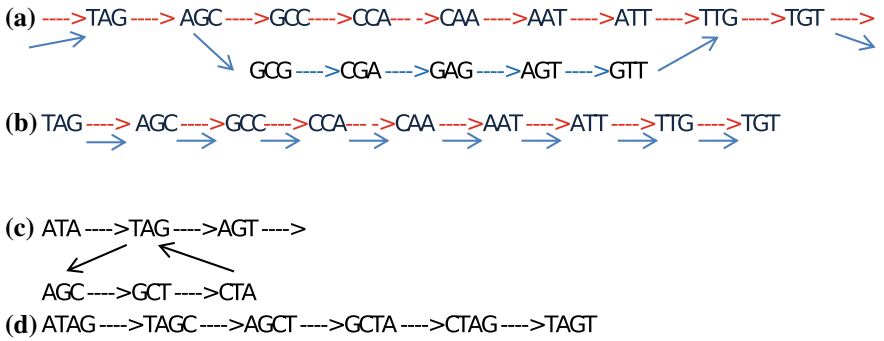

Fig. 3 **a** De Bruijn diagram of a grouping with two estimated rehashes S = TAGCCAATTGT...TAGCGAGTTGT. The minor differences in the inaccurate rehashes shape a bulge having two branches: The red branch (AGC) → (GCC) → (CCA) → (CAA) → (AAT) → (ATT) → (TTG) and the blue branch (AGC) → (GCG) → (CGA) → (GAG) → (AGT) → (GTT) → (TTG). **b** We improve the diagram by changing the grouping from TAGCCAATTGT to TAGCGAGTTGT, therefore shaping a correct rehash. The modified arrangement compares to a non-fanning way on the de Bruijn diagram. **c** A firmly found rehashed k-mer (ATC) frames a circle in the diagram. **d** Enhancing the estimation of k can resolve the circle in **c**

to be connected utilizing the succession modification calculation; generally, S_1, the twisted adaptation of S_0, is not accessible for the development of the diagram utilizing a bigger estimation of $k = k_1$. The objective of the first cycle is to crumple swells caused by single point changes or tiny indels. Subsequently, we can build the estimation of k and develop another de Bruijn diagram $G_1 = DB(S_1, k_1)$, where $k_1 > k_0$. The procedure proceeds until the point that we achieve an estimation of k that is sufficiently extensive to uncover huge scale synteny blocks (Algorithm 1). As a rule, the iterative procedure should proceed until the point when the genome is displayed as a solitary synteny block. This contention may seem irrational at first locate, as our objective was to break down genomes into synteny blocks. In any case, anyone can see here two perceptions shown in the above section that helps us to follow our means to find past synteny blocks.

Result: $S_t, Graph_t$
Input: *Genome Sequences G*
procedure
$ITERATIVE\ DE\ BRUIJN\ (G, ((k_0, C_0), ..., (k_t, C_t)))$
$S_0 \leftarrow Concatenate(G)$
$Assert(k_0 < k_1 < ... < k_t)$
$Assert(C_0 < C_1 < ... < C_t)$
$i \leftarrow 0$
while $i < t$ **do**
  $Graph_i \leftarrow ConstructDeBruijnGraph(k_i, S_i)$
  $S_i \leftarrow SimplifyBulgesSmallerThanC(Graph_i, C_i)$
  $i \leftarrow i + 1$
end

Algorithm 1: Iterative de Bruijn Graph (Existing algorithm)

3 Methods

3.1 SSAHA Improvement by Dynamic Programming

An improved dynamic programming approach for lower k-mer is proposed here. For this, we have to construct a two-dimensional array of size $QuerySequenceLength * SubjectSequenceLength$.

Improved Algorithm Recurrence relation of the algorithm is given below:
Array[i][j]=Array[i-1][j-1]+1 ; row[i]==col[j]
Array[i][j]=Array[i-1][j-1] ; row[i]!=col[j]

Result: $Array[QuerySequenceLength][SubjectSequenceLength]$
Input: $Maximum\ Value\ from\ Last\ Row\ or\ Column$
procedure
$HASH\ TABLE\ DP(QuerySequence\ ,\ SubjectSequence)$
for $i \leftarrow 0\ to\ SubjectSequenceLength - 1$ **do**
  $Array[0][i] \leftarrow 0$
end
for $j \leftarrow 0\ to\ QuerySequenceLength - 1$ **do**
  $Array[j][0] \leftarrow 0$
end
for $i \leftarrow 0\ to\ SubjectSequenceLength - 1$ **do**
  for $j \leftarrow 0\ to\ QuerySequenceLength - 1$ **do**
    if $QuerySequence[i] == SubjectSequence[j]$ **then**
      $Array[i][j] = Array[i-1][j-1] + 1$
    else
      $Array[i][j] = Array[i-1][j-1]$
    end
  end
end

Algorithm 2: Hash Table Construction by Dynamic Programming

DP table construction Suppose we have three sequences in a database D. They are ACCGTTGTAGCT, ACCTTGTTCTATGACC, and CGTTGTCGTCAAATGG and a query sequence is AAATGG. DP table for them are shown in the following three tables (Tables 6, 7 and 8).

DP table analysis We have to get the highest value of last row and last column of each table. Highest Values from Table 1 is 2, from Table 2 is 3, and from Table 3 is 6. Now we have to get the highest value from the highest value of each table. Here we can see the value is 6 that means we have an alignment of Query sequence of length 6 to the 3rd subject sequence.

Index finding from DP table Suppose, Query Length is m (column wise) and Subject Length is n (row wise). If the highest value is from ith index of last row then alignment value for that index will be **(i-m)**. If highest value is from ith index of last column

Table 6 DP Table 01

		A	**C**	**C**	**G**	**T**	**T**	**G**	**T**	**A**	**G**	**C**	**T**
	0	0	0	0	0	0	0	0	0	0	0	0	0
A	0	1	0	0	0	0	0	0	0	1	0	0	0
A	0	1	1	0	0	0	0	0	0	1	1	0	0
A	0	1	1	1	0	0	0	0	0	1	1	1	0
T	0	0	1	1	1	1	1	0	1	0	1	1	**2**
G	0	0	0	1	2	1	1	2	0	1	1	1	1
G	0	0	0	0	**2**	**2**	1	**2**	**2**	0	**2**	1	1

Table 7 DP Table 02

		A	**C**	**C**	**T**	**T**	**G**	**T**	**T**	**C**	**T**	**A**	**T**	**G**	**A**	**A**	**C**
	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
A	0	1	0	0	0	0	0	0	0	0	0	1	0	0	1	0	0
A	0	1	1	0	0	0	0	0	0	0	0	1	1	0	1	1	0
A	0	1	1	1	0	0	0	0	0	0	0	1	1	1	1	1	1
T	0	0	1	1	2	1	0	1	1	0	1	0	2	1	1	1	1
G	0	0	0	1	1	2	2	0	1	1	0	1	0	3	1	1	1
G	0	0	0	0	1	1	**3**	2	0	1	1	0	1	1	**3**	1	1

Table 8 DP Table 03

		C	**G**	**T**	**T**	**G**	**T**	**C**	**G**	**T**	**C**	**A**	**A**	**A**	**T**	**G**	**G**
	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
A	0	0	0	0	0	0	0	0	0	0	0	1	1	1	0	0	0
A	0	0	0	0	0	0	0	0	0	0	0	1	2	2	1	0	0
A	0	0	0	0	0	0	0	0	0	0	0	1	2	3	2	1	0
T	0	0	0	1	1	0	1	0	0	1	0	0	1	2	4	2	1
G	0	0	1	0	1	2	0	1	1	0	1	0	0	1	2	5	3
G	0	0	1	1	0	2	2	0	2	1	0	1	0	0	1	3	**6**

then alignment value for that index will be **(n-i)**. So from the above three DP tables the alignment can be found from the following table (Table 9).

So from the subject sequences in the database, we get that the if we align the Query sequence to the 10th index of the third subject sequence we will get the maximum match of 6 characters.

Table 9 Result analysis

Database			
Parameters	Table 1	Table 2	Table 3
Highest value	2	3	6
ith index from column	4		6
ith index from row	4, 5, 7, 8, 10	6, 14	16
Alignments from column **(n-i)**	8		**10**
Alignments from row **(i-m)**	−2,−1, 1, 2, 4	0, 8	**10**

3.2 Improved MUMs Finding Using BWT Matrix and FM Index

Existing algorithm of MUMs finding uses suffix tree of one genome to thread the suffices of other genomes to that tree. But using suffix tree an unique match cannot be always identified. For example, Fig. 4 shows the suffix tree of genome A and genome B which are **caatacc** and **gtaccgtacc** consecutively. The suffix tree shows that the string "tacc" is one of the MUMs as it is a maximal match between the genomes. But we can see that "tacc" is not unique in this set of genomes. Genome A contains the substring "tacc" twice, "g**tacc**g**tacc**". So obviously "tacc" cannot be an MUM, as it is not unique (Table 10).

In our approach, Burrows wheeler transform (BWT) matrix is used instead of that threading procedure. In case of BWT, the first and last column is needed to find whether the sequence exists in the other one or not. So without constructing suffix tree, BWT of one genome and can check the suffixes of other genome using FM

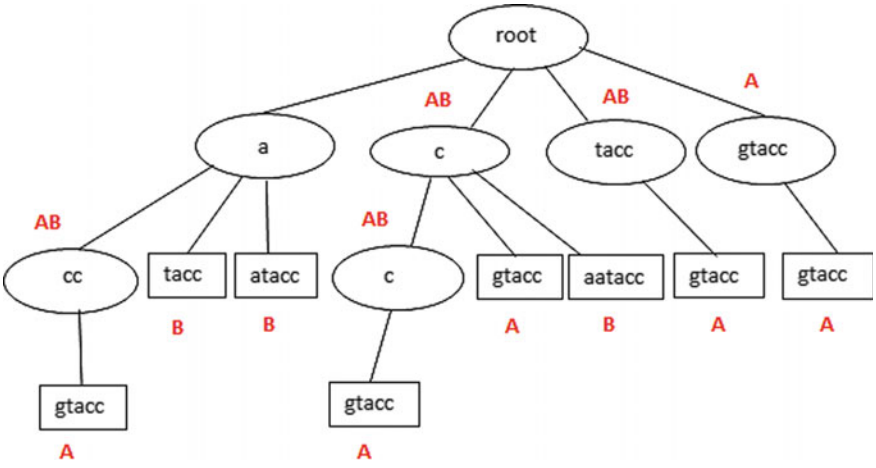

Fig. 4 Suffix tree on genome A and B

Table 10 Suffices of genome A and genome B

Genome A	Genome B
caatacc	gtaccgtacc
aatacc	taccgtacc
atacc	accgtacc
tacc	ccgtacc
acc	cgtacc
cc	gtacc
c	tacc
	acc
	cc
	c

index to find maximal unique matches. Generally, MUMs of very short length must not be reported. So along with the genomes, a threshold value to set the minimum base pairs in the MUM alignment is provided as input parameter. As the MUMs can overlap, so the algorithm checks and compares the genomes by deleting only one character from last position each time. The algorithm is given below.

Result: *mumStorage*
Input: *Genome A*, *Genome B*, *Threshold*
procedure
MUMfinding(*GenomeA*, *GenomeB*, *threshold*)
S ← *Genome B*
L ← *BWT of Genome A*
mumStorage ← ∅
Give each character in L a number, equal to the number of occurrence of that character consecutively starting from zero
F ← *Sorted BWT of Genome A*
while *length*(*S*) >= *threshold* **do**
  match ← ∅
  match ← *all matched substrings of S found by modified FM index with L and F*
  if *length*(*match*) == 1 *and length*(*matchedElement*) >= *threshold* **then**
    if ***mumStorage*** *does not contain* ***match*** **then**
      mumStorage ← *match*
    else
      delete **match** *from* **mumStorage**
    end
  end
  delete the last character of string S
end

Algorithm 3: MUM finding by BWT and FM index

3.3 Improved Synteny Block Detection Method

Synteny Block loop deletion of length ≤ bulge length In the existing method, it is considered that all the bulges are smaller than the threshold loop. But in the real-life genome sequence there can be many variations. There can be loop sizes that are smaller than the bulge or equal to the size of the bulge as the algorithm only concerns about the bulges smaller than the loop so there is a scope of improvement. Examples of genomes are given in Fig. 5 that contains the variation of the size of loop and bulge.

Improved Algorithm In the improved algorithm, it is ensured that a bulge will be handled after the deletion of all the loops smaller or equal to the size of that particular bulge. The algorithm is given below.

Result: $S_t, Graph_t$
Input: *Genome Sequences* G
procedure
$ITERATIVE\ DE\ BRUIJN\ (G, ((k_0, C_0), ..., (k_t, C_t)))$
$S_0 \leftarrow Concatenate(G)$
$Assert(k_0 < k_1 < ... < k_t)$
$Assert(C_0 < C_1 < ... < C_t)$
$i \leftarrow 0$
while $i < t$ **do**
  $Graph_i \leftarrow ConstructDeBruijnGraph(k_i, S_i)$
  $LoopLength \leftarrow FindLoopLengthUsingDFS(Graph_i)$
  $BulgeLength \leftarrow FindBulgeLength(Graph_i)$
  while $LoopLength \leq BulgeLength$ **do**
    $i \leftarrow i + 1$
    $Graph_i \leftarrow ConstructDeBruijnGraph(k_i, S_i)$
    $LoopLength \leftarrow FindLoopLengthUsingDFS(Graph_i)$
    $BulgeLength \leftarrow FindBulgeLength(Graph_i)$
  end
  $S_i \leftarrow SimplifyBulgesSmallerThanC(Graph_i, C_i)$
  $i \leftarrow i + 1$
end

Algorithm 4: Iterative de Bruijn Graph (Improved Algorithm)

4 Results

4.1 Statistical Analysis of Hash Table Verus DP

For the k-mers upto length 2, the dynamic programming approach triumphs over the hash table method. But from k-mer of length 3 and above the hash table method

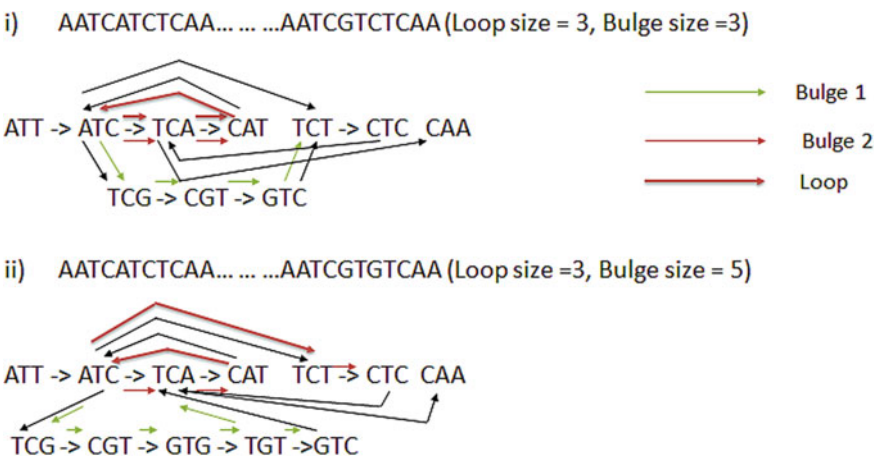

Fig. 5 Loop created in the de Bruijn graph is (i) smaller and (ii) equal to bulge size

Table 11 Statistical analysis of dynamic programming

k-mer length	Query length	DP
1	1000	93.18% faster
2	1000	39.28% faster
3	1000	75.21% slower
1	500	92.80% faster
2	500	34.37% faster
3	500	77.3% slower

is more improved in terms of speed though runtime complexity of both algorithms is the same. For a database of 1000 sequences each having length 1000, the comparison is given below (Table 11).

4.2 MUM Finding: Suffix Tree Versus BWT and FM Index

Existing approach of MUM finding fails to detect whether the MUMmer is unique or not in any of the two genomes that are to be aligned. This proposed approach will maintain the uniqueness of the MUMmers. If any repetitive maximal match is found greater than or equal the threshold value then the match is discarded. Statistically for smaller length genome, the proposed approach gives comparatively faster result. But with the increase of genome length, the speed slows down. Though the time complexity for both the methods is same (Table 12).

Table 12 Statistical analysis of MUM finding by BWT and FM index

Length of genome A	Length of genome B	Improvement
500	500	16.21% faster
1000	1000	14.02% faster
1500	1500	11.47% faster
2000	2000	10.63% faster

Table 13 Tabular analysis of synteny block generation with genome sequence

Sequence	Loop and bulge size	Sibelia	Proposed method
i	Loop = 0 Bulge = 5	Can handle	Can handle
ii	Loop = 3 Bulge = 3	Can't handle	Can handle
iii	Loop = 3 Bulge = 5	Can't handle	Can handle

4.3 Synteny Block: Detection Versus Avoidance of Loop ≤ Bulge

There is a potential scope of improvement in that case if the bulge larger or equal in accordance to the loop. In that cases loop has to be detected than the determination of the length of the loop has to be done and the length of the bulge and the size comparison of the loop and bulge will decide the computation that whether the size of the K-mer has to be increased or the Sequence Modification Algorithm has to be used.We have proposed to detect the loop using Depth-First Search as it has linear time complexity. We find the loops in lexicographical order. If discovered cases are not resolved than there can be branching paths in the synteny block which will not lead to the proper comparison of microbial genomes. Analysis over three genomes
(i) $ATCGGTTAACT...ATCGATCAACT$,
(ii) $AATCATCTCAA...AATCGTCTCAA$ and
(iii) $AATCATCTCAA...AATCGTGTCAA$
to generate synteny block is shown below in Table 13.

5 Time and Space Complexity Analysis

5.1 Dynamic Programming Approach for SSAHA

In the dynamic programming approach, the main task is to construct the table which takes O(mn) where m and n is the length of query sequence and subject sequence, respectively. Moreover, finding out the index of alignment can be done in linear time by finding the highest value from last row and column. So overall time complexity

will be O(mn). To calculate a specific index of the array we just need the upper diagonal value of that index. So the space complexity will be minimum of O(m) and O(n).

5.2 MUM Finding by BWT and FM Index

Construction of Suffix tree needs linear time and space complexity by using pointers efficiently [2, 6, 12, 13]. On the other hand building BWT and FM index and working with those requires the same linear time and space complexity.

5.3 Synteny Block : Detection of Loop and Bulge

In the actual version of Sibelia, the time complexity is mainly dependent on the construction of de Bruijn graph and finding the length of bulges. Linear time algorithms are available to construct de Bruijn graph [4, 10]. The complexity to find bulges smaller than that of the threshold loop can be also done in linear time. In this improved algorithm it is suggested that the Depth-First Search (DFS) algorithm to find out the size of the loop which can be done in linear time with the complexity of O(V+E), where V is the number of vertex and E is the number of edges. So overall time complexity will be linear.

6 Discussion

In the diversified field of string comparison, three different issues are described and better approaches are shown here. At first, a dynamic approach is proposed against hash table algorithm of DNA database which shows tremendous improvement of runtime (though same in complexity) for smaller k-mer. Then an improved way of ensuring uniqueness of MUMs is described using BWT matrix and FM indexing instead of existing suffix tree method where repetitive MUMs are found unexpectedly. Last of all, a major discrepancy of avoiding the loops not greater than bulge size to find synteny block by Sibelia is determined and improved algorithm is proposed. Thus, the aim of improving some existing string algorithms in the field of computational biology is expected to be served with flying colors.

References

1. Burrows M, Wheeler DJ (1994) A block sorting lossless data compression algorithm
2. Chen MT, Seiferas J (1985) Combinatorial algorithms on words. Springer, New York, pp 97–107

3. Delcher AL, Kasif S, Fleischmann RD, Peterson J, White O, Salzberg SL (1999) Alignment of whole genome
4. Dinh H, Kundeti KV, Rajasekaran S, Thapar V, Vaughn M Efficient parallel and out of core algorithms for constructing large bi-directed de Bruijn graphs. Department of computer science & engineering, University of Connecticut, 371 Fairfield way, U-2155, Storrs, CT, 06269, USA
5. Ferragina P, Manzini G (2000) Opportunistic data structures with applications. Proc IEEE FOCS 390–398
6. McCreight EM (1976) J ACM 23:262–272
7. Minkin I, Patel A, Kolmogorov M, Vyahhi N, Pham S (2013) Sibelia: a scalable and comprehensive synteny block generation tool for closely related microbial genomes. Department of computer science and engineering, UCSD, La Jolla, CA, USA. St. Petersburg Academic University, St. Petersburg, Russia
8. Needleman SB, Wunsch CD (1970) J Mol Biol 48:443–453
9. Ning Z, Cox AJ, Mullikin JC (2001) SSAHA: a fast search method for large DNA databases. Informatics division, The Sanger centre, Wellcome trust genome campus, Hinxton, Cambridge CB10 1SA, UK
10. Rahman MS, Rahman MM, Sharker R (2017) HaVec: an efficient de Bruijn graph construction algorithm for genome assembly. Depatment of CSE BUET, ECE Building, West Palashi, Dhaka-1205, Bangladesh
11. Smith TF, Waterman MS (1981) J Mol Biol 147:195197
12. Ukkonen E (1995) Algorithmica 14:249–260
13. Weiner P (1973) Linear pattern matching algorithms. In: Proceedings of the 14th IEEE symposium on switching and automata theory, p 111