Mining Maximal Adjacent Frequent Patterns from DNA Sequences using Location Information

Submitted by:

Md. Tayeb Hasan Shuvo	Student No. 094436
Asif Ahmed Sarja	Student No. 094440

Supervisor
Tareque Mohmud Chowdhury
Assistant Professor,
CSE Department, IUT

Co-Supervisor
Moin Mahmud Tanvee
Lecturer, CSE Department, IUT

Computer Science and Engineering
Islamic University of Technology
May 21, 2013.

Table of Contents

Chapter Title	Page
Title Page	**1**
Table of Contents	**2**
1. Introduction	
1.1 Background	**4**
1.2 Statement of problem	**5**
1.3 Motivation	**5**
1.4 Methodology	**6**
1.5 Scopes	**6**
2. Literature Review	**7**
2.1 Maximal contiguous frequent pattern mining	**7**
2.2 Frequent pattern mining existing techniques	**8**
2.2.1 Kang's approach	**8**
2.2.2 Position based approach	**9**
2.2.3 PrefixSpan	**10**
2.2.4 MacosFSpan	**11**
2.3 Improvement of existing methods	**13**
2.3.1 Projected database	**14**
2.3.2 Fspan Tree with position information	**15**
2.3.3 Sorting method	**17**
2.3.4 Joining method	**18**
2.3.5 Final result with improved approach	**19**
2.3.6 Proposed algorithm	**20**
2.3.7 Flow chart	**21**
2.3.8 Experimental & Performance Analysis	**22**

3. Progress Report **27**

3.1 Improved areas **27**

3.2 Future work **27**

4. Conclusion **28**

5. Simulation code **29**

6. References **44**

Chapter 1

Introduction

1.1 Background

Nowadays, frequent pattern mining is drawing a considerable attention to many researchers for its remarkable contribution in many research areas .Among these, bioinformatics is a very rapidly growth area that has resulted in datasets with new characteristics. Basically, bioinformatics requires information technologies for quick and effective analysis of huge biological information. Thus, researches in the analysis and prediction of genetic function by bioinformatics can be helpful to control the excessive expenses required by the existing biological techniques and to reduce the experimental verification time. Genetic function analysis includes predictions of specific protein function region and the comparison between DNA as well as amino acid sequence homology.

The growth of bioinformatics has resulted in datasets with new characteristics. The DNA sequences typically contain a large number of items. From them biologists assemble a whole genome of species based on frequent concatenate sequences. These frequent concatenate sequences ordinarily have hundreds of items. How to efficiently discover long frequent concatenate sequences poses a great challenge for existing sequential pattern discovery algorithms.

Many studies have contributed to the efficient mining of sequential patterns, e.g., [1,2,9,7,8,3,6,5,4]. Almost all of the previously proposed methods for mining sequential patterns are Apriori-like, i.e., based on the Apriori property proposed in association mining [1], which states the fact that any super-pattern of a non-frequent pattern cannot be frequent. A typical Apriori-like method such as GSP [9] adopts a multiple-pass, candidate-generation-and-test approach in sequential pattern mining. However, their running time increases exponentially with increasing average sequence length, and thus such high-dimensional data renders most current algorithms impractical.

There are several DNA pattern mining techniques. Like Kang's approach, Position based approach, PrefixSpan approach, MacosFSpan approach. Frequent pattern mining in DNA sequence is very important for obtaining vital clues regarding new genetic discovery and functional analysis, predicting the biological function of a gene, finding the evolution distance, helping genome assembly, finding repeats within a genome.

1.2 Statement of problem

The following are the problems associated with frequent DNA pattern mining

- **Space Requirement**: DNA pattern requires a large amount of disk space to be stored. If the proposed method may be applied then the space requirement may be reduced.

- **Time Complexity**: The time complexity for DNA pattern data sets are likely to be large since a vast amount of processing needs to be done for a conclusion to be reached.

- **Redundant and Irrelevant Data**: As the volume of data is very large, DNA pattern data set usually incorporates some data which are redundant or irrelevant.

1.3 Motivation

The ultimate goal of this study is to find the best way of frequent DNA pattern mining that produces the ideal classification. This will pave the way for better machine percept and hence better data intelligibility.

The objectives of frequent DNA pattern mining are manifold, the most important ones being:

- Obtain vital clues regarding new genetic discovery and functional analysis

- The DNA sequences typically contain a large number of items from which biologists assemble a whole genome of species based on frequent contiguous sequences. These frequent sequences ordinarily consist of hundreds of items.

- Capability of reducing time and space complexity

- Avoids generalization and over fitting problems

1.4 Methodology

We will follow the following steps in our study and the process of implementation

- Study existing implementations of the different variants of maximum contiguous frequent pattern mining
- Comparative performance analysis of the various methods
- Identify key areas where improvement is possible.
- Prepare model representing the new goal
- Reproduce the model through proper simulation

1.5 Scopes

We want to identify a better model that will reduce the time and space complexity of the current approaches. By examining several existing approaches we find some scopes for developing.

- There is possibility of improvement
- Existing techniques may be modified to give better accuracy values
- Maximum frequent pattern mining will help to improve other related fields
- Time complexity can be reduced by modification
- Space complexity can be reduced by modification

Chapter 2

Literature Review

2.1 Maximal contiguous frequent pattern mining

A frequent pattern is a frequent pattern that occurs frequently in multiple DNA sequence without any gap in it's sequence. A contiguous frequent pattern is a frequent pattern that occurs frequently in multiple DNA sequence without any gap in it's sequence.

The DNA sequence data is one of the basic and important data among biological data. The DNA sequence pattern mining has got wide attention and rapid development. Traditional algorithms for the sequential pattern mining may generate lots of redundant patterns when dealing with the DNA sequence. The maximal frequent pattern is preferable to express the function and structure of the DNA sequence. Base on the characteristics of the DNA sequence, the author develops the joined maximal pattern segments algorithm-JMPS, for the maximal frequent patterns mining of the DNA sequence. First, the maximal frequent pattern segments base on adjacent generated. Then, longer maximal frequent pattern can be obtained by combining the above segments, at the same time deleting the nonmaximal patterns. The algorithm can deal with the DNA sequence data efficiently.

Nowadays, frequent pattern mining is drawing a considerable attention to many researchers for its remarkable contribution in many research areas Among these, bioinformatics is a very rapidly growth area that has resulted in datasets with new characteristics. Basically, bioinformatics requires information technologies for quick and effective analysis of huge biological information Thus, researches in the analysis and prediction of genetic function by bioinformatics can be helpful to control the excessive expenses required by the existing biological techniques and to reduce the experimental verification time. Genetic function analysis includes predictions of specific protein function region and the comparison between DNA as well as amino acid sequence homology. Finally, the outcomes of this analysis can contribute to obtain vital clues regarding new genetic discovery and functional analysis. To analyze the huge biological data, the traditional bioinformatics technologies include comparative and analytic methods of the homology of biological data sequences that are of two kinds: namely DNA and amino acid sequence.

2.2 Frequent pattern mining existing techniques

Consequently, a lot of researches have been done in the area of fast and efficient sequential pattern mining. Frequent contiguous sequential pattern mining is among them that basically finds the frequent contiguous patterns in the database sequences of more than two. The problem of finding the frequent contiguous patterns seems to be very important and widely applied in bioinformatics. In this regard, a recent method was proposed by Kang *et al*. using fixed-length spanning tree to show its superiority over the MacosVSpan approach to mine frequent contiguous sequential patterns in large biological data sequences. In the recent approach once fixed-length spanning tree is constructed, all the candidates of all lengths are generated including frequent and non-frequent patterns. Finally, each candidate is scanned through the database to check its support, which causes huge time and memory consumption indeed. In general, most of the previous approaches for sequential pattern mining are based on *Apriori* algorithm. Besides, there is another technique called PrefixSpan that finds out the frequent sequential patterns based on the projected databases. The available algorithms seem to be impractical as the running time increases exponentially with increasing average sequence length.

2.2.1 Kang's approach:

- Fixed length contiguous sequences
- Projected database not required
- Construct fixed length spanning tree
- Satisfy a specified minimum support threshold

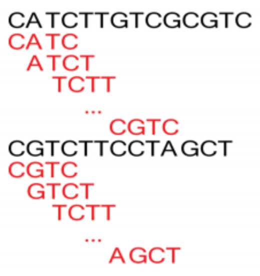


Figure 1: Fixed-length scanning method over the database.

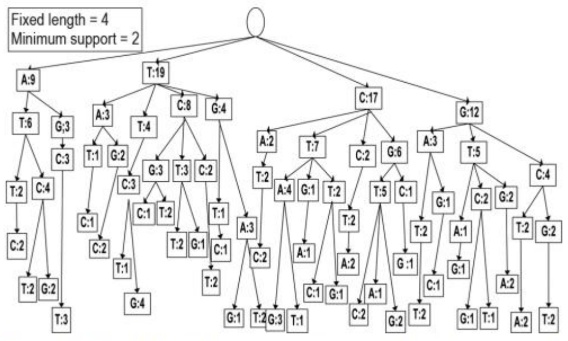


Figure 2: Fixed-length spanning tree without sequence ID and position information.

2.2.2 Position based approach:

- Improvement of Kang's approach
- Position information record
- Hash table
- Apriori algorithm : All the Items in the first pattern excluding the first item should be exactly the same as all the items of the second pattern excluding the last item

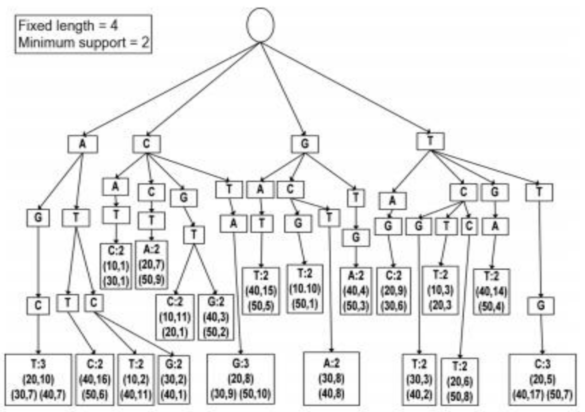


Figure 4: Proposed fixed-length spanning tree with sequence ID and position information.

Table 4: Length-4 frequent subsequences

Length-4	(ID, Start position)
AGCT	(20,10), (30,7), (40,7)
ATCG	(30,2), (40,1)
ATCT	(10,2), (40,11)
ATTC	(40,16), (50, 6)
CATC	(10,1), (30,1)
CCTA	(20,7), (50,9)
CGTC	(10,11) (20,1)
CGTG	(40,3),(50,2)
CTAG	(20,8), (30,9), (50,10)
GATT	(40,15), (50,5)
GCGT	(10,10), (50,1)
GCTA	(30,8), (40,8)
GTGA	(40,4), (50,3)
TAGC	(20,9), (30,6)
TCCT	(20,6), (50,8)
TCGT	(30,3), (40,2)
TCTT	(10,3), (20,3)
TGAT	(40,14), (50, 4)
TTCC	(20,5), (40,17), (50,7)

Table 5: Length-4 frequent subsequences joining based on the proposed algorithm

Length-4	Length-4	(ID, Start position)	Length-5 candidate support
AGCT	GCTA	{(20,10), (30,7), (40,7) } ^ {(30,8), (40,8)}	2
ATCG	TCGT	{(30,2), (40,1)} ^ {(30,3), (40,2)}	2
ATCT	TCTT	{(10,2), (40,11)} ^ {(10,3), (20,3)}	1
ATTC	TTCC	{(40,16), (50, 6)} ^ {(20,5), (40,17), (50,7)}	2
CATC	ATCT	{(10,1), (30,1)} ^ {(10,2), (40,11)}	1
CATC	ATCG	{(10,1), (30,1)} ^ {(30,2), (40,1)}	1

Table 6: Length-5 frequent subsequences

Length-5	(ID, start position)
AGCTA	(30,7), (40,6)
ATCGT	(30,2), (40,1)
ATTCC	(40,16), (50,6)
CCTAG	(20,7), (50,9)
CGTGA	(40,2), (50,2)
GATTC	(40,15), (50,5)
TAGCT	(20,9), (30,6)
TCCTA	(20,6), (50,8)
TGATT	(40,14), (50,4)
TTCCT	(20,5), (50,7)

Table 7: Length-6 frequent subsequences

Length-6	(ID, start position)
GATTCC	(40,14), (50,5)
TCCTAG	(20,6), (50,8)
TGATTC	(40,14), (50,4)
TTCCTA	(20,5), (50,7)

Hash table

Item	Start_index	End_index
A	1	4
C	5	10
G	11	12
T	13	18

Table 8: Length-7 frequent subsequences

Length-7	(ID, start position)
TGATTCT	(40,14), (50,4)
TTCCTAG	(20,5), (50,7)

2.2.3 PrefixSpan :

- Creates projected database
- Examine the prefix subsequences
- Project only their corresponding postfix subsequences
- In each projected database,
- Sequential patterns are grown by exploring local length-1 frequent patterns
- This process is performed recursively on projected databases

Example: ATCGTGAGCTATCTGATTCC is a sample DNA sequence

T projected database:

CGTGAGCTATCTGATTCC

GAGCTATCTGATTCC

ATCTGATTCC

CTGATTCC

GATTCC

TCC

CC

Prefix: T
Postfix: Rest of the subsequence excluding T

Result: ATCGTGAGCTATCTGATTCC

Now considering TC and TG as prefix and generating the projected database for them

TC projected database:

GTGAGCTATCTGATTCC

TGATTCC

C

TG projected database:

AGCTATCTGATTCC

ATTCC

Projected databases for A, C and G are created similarly

2.2.4 MacosFSpan :

- Improvement of PrefixSpan
- Less number of projected database is created
- Sequential patterns are grown by exploring fixed length-w frequent pattern in each projected database.
- Creates Fspan tree

Example: A sample DNA database

ID	**SEQUENCE**
10	CATCTTGTCGCGTC
20	CGTCTTCCTAGCT
30	CATCGTAGCTAG
40	ATCGTGAGCTATCTGATTCC
50	GCGTGATTCCTAG

Prefix: T
Postfix: Rest of the subsequence excluding T
W=3

T projected Database:

```
10    CTTGTCGCGTC
       TGTCGCGTC
        GTCGCGTC
         CGCGTC
              C    ×
30    CGTAGCTAG
        AGCTAG
           AG      ×
50    GATTCCTAG
        TCCTAG
           AG      x

20    CTTCCTAGCT
       TCCTAGCT
        CCTAGCT
          AGCT

40    CGTGAGCTATCTGATTCC
        GAGCTATCTGATTCC
           ATCTGATTCC
            CTGATTCC
             GATTCC
               TCC
                CC  x
```

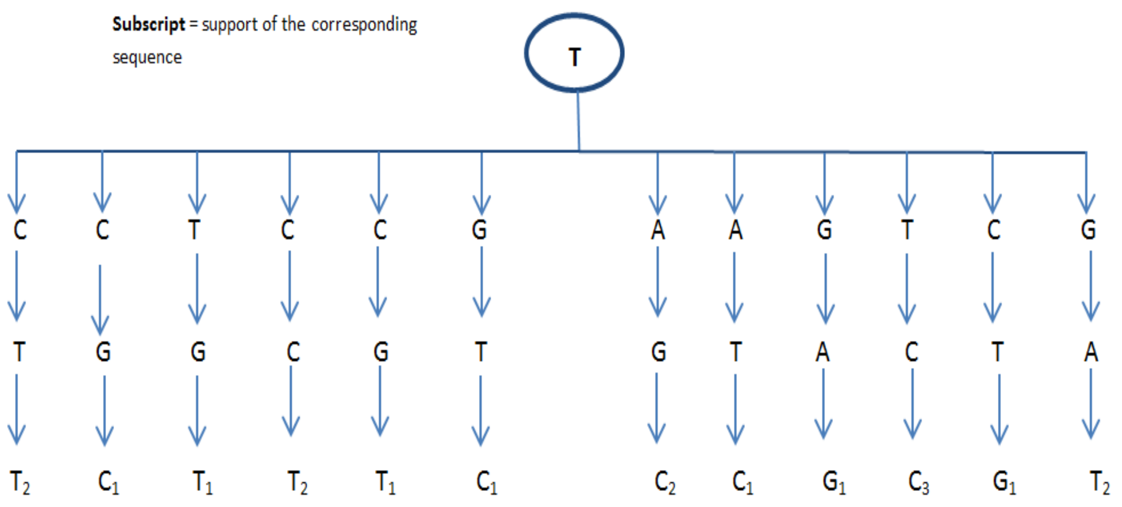


Fig: FSpan tree

With minimum support, 6 candidates has confirmed as 4-length frequent pattern with prefix T.

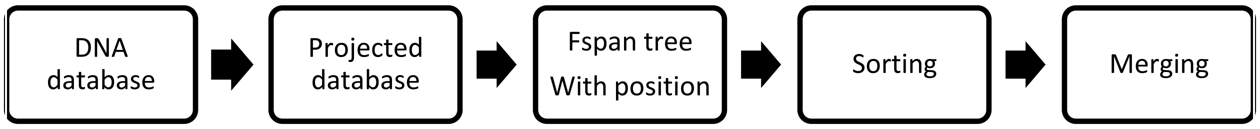
TAGC
TCCT
TCGT
TCTT
TGAT
TTCC

For prefix A, C, G candidate 4-length frequent pattern are

AGCT	CATC	GATT
ATCG	CCTA	GCGT
ATCT	CGTC	GCTA
ATTC	CGTG	GTGA
	CTAG	

Considering all of the frequent pattern as prefix, it generates the projected databases and also create Fspan tree recursively.

2.3 Improvement of existing methods



First in our approach we are going to create projected database according to MacosFspan approach[2].After creating projected database from the main DNA database we will make the FSpan tree. But here the main difference is that we are keeping the position information of each DNA sequence in the main database according the position based method [1].Then we are going to sort the frequent subsequences according the last occurring position of them in the database and we are also keeping a hash table for storing the indexes of the prefixes. Then we start joining the subsequent pattern according to the Apriori joining rule .Through this approach we get the maximum length frequent pattern at once. While creating the projected database we are going to run 4 parallel processor to process all projected database simultaneously that saves a lot of time considering to the previous processes.

- Creating projected database according to MacosFSpan approach[2]
- FSpan tree
- Recording the position of sequences in database according to position based method[1]
- **Sorting the sequences from last occurring position**
- **Hash table for prefixes**
- **Merging fixed length frequent pattern to get the maximal length according to their sequence position.**

2.3.1 Projected Database:

Example: A sample DNA database

ID	SEQUENCE
10	CATCTTGTCGCGTC
20	CGTCTTCCTAGCT
30	CATCGTAGCTAG
40	ATCGTGAGCTATCTGATTCC
50	GCGTGATTCCTAG

Prefix: T
Postfix: Rest of the subsequence excluding T
W=3

T projected Database:

10 CTTGTCGCGTC
TGTCGCGTC
GTCGCGTC
CGCGTC
C ×

20 CTTCCTAGCT
TCCTAGCT
CCTAGCT
AGCT

30 CGTAGCTAG
AGCTAG
AG ×

40 CGTGAGCTATCTGATTCC
GAGCTATCTGATTCC
ATCTGATTCC
CTGATTCC
GATTCC
TCC
CC x

50 GATTCCTAG
TCCTAG
AG x

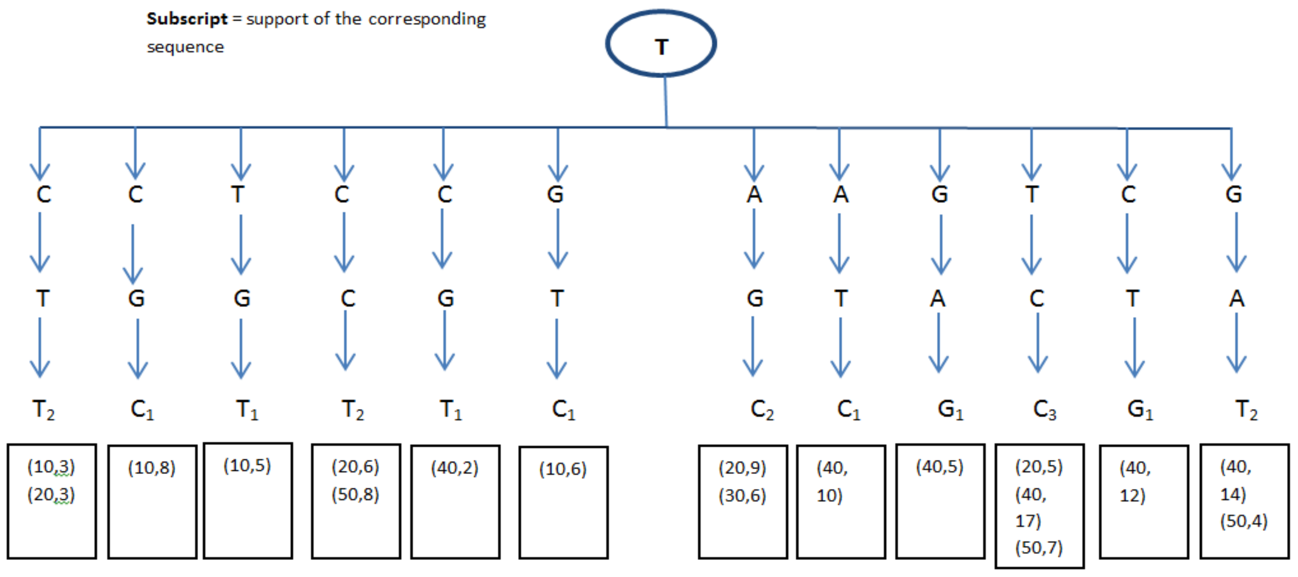
2.3.2 Fspan Tree with position information

In previous methods they used FSpan tree for generating frequent pattern from DNA sequence. In our proposed method we are also using FSpan tree but we are keeping the record of position information of each DNA sequence in database. According to the position information we are sorting the subsequences according to the last occurring position and then joining the subsequences with Apriori approach.

For each base in DNA sequence we create projected database and their corresponding FSpan tree. For example we have shown here the FSpan tree for T projected database from the following sample DNA database.

ID	SEQUENCE
10	CATCTTGTCGCGTC
20	CGTCTTCCTAGCT
30	CATCGTAGCTAG
40	ATCGTGAGCTATCTGATTCC
50	GCGTGATTCCTAG

Minimum support=2


Subscript = support of the corresponding sequence
T
C → T → T_2 (10,3) (20,3)
C → G → C_1 (10,8)
T → G → T_1 (10,5)
C → C → T_2 (20,6) (50,8)
C → G → T_1 (40,2)
G → T → C_1 (10,6)
A → G → C_2 (20,9) (30,6)
A → T → C_1 (40, 10)
G → A → G_1 (40,5)
T → C → C_3 (20,5) (40, 17) (50,7)
C → T → G_1 (40, 12)
G → A → T_2 (40, 14) (50,4)

With minimum support, 6 candidates have confirmed as 4-length frequent pattern with prefix T.

TAGC (20, 9)(30,6)

TCCT (20, 6) (50,8)

TCGT (30, 3) (40, 2)

TCTT (10, 3) (20,3)

TGAT (40, 14) (50, 4)

TTCC (20, 5) (40, 17) (50, 7)

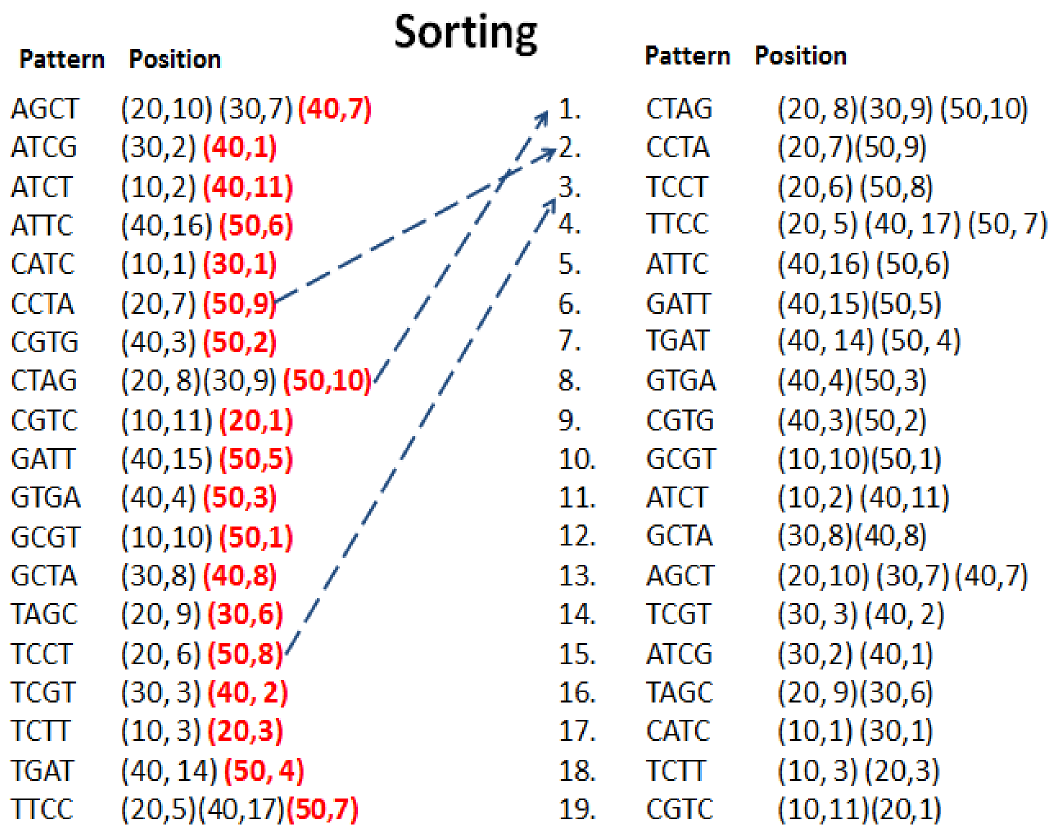
And for prefix A, C, G candidate 4-length frequent pattern are

AGCT(20,10) (30,7) (40,7)	CATC(10,1) (30,1)	GATT(40,15)(50,5)
ATCG(30,2) (40,1)	CCTA (20,7)(50,9)	GCGT(10,10)(50,1)
ATCT(10,2) (40,11)	CGTC (10,11)(20,1)	GCTA(30,8)(40,8)
ATTC (40,16) (50,6)	CGTG (40,3)(50,2)	GTGA(40,4)(50,3)
	CTAG (20, 8)(30,9) (50,10)	

After generating the projected databases and Fspan tree we get 19 four length frequent pattern. Now we will sort the 4-length frequent pattern according to their last occurring position and then join them for generating maximum length frequent pattern.

2.3.3 Sorting method

After generating the spanning tree we will sort the frequent subsequences in a different way. We will sort the frequent subsequences according to their last occurring position. By this sorting when we join them for maximum length frequent pattern we get the desired result at once. And we also generate other length frequent pattern simultaneously. And also we are making the joining process one sided as we start from the last occurring position. Joining is possible only to the left side as we start from the last occurring position. But in previous proposed approaches the joining process goes on both sides that is why those methods require more time than our approach.



Pattern	Position
AGCT	(20,10) (30,7) **(40,7)**
ATCG	(30,2) **(40,1)**
ATCT	(10,2) **(40,11)**
ATTC	(40,16) **(50,6)**
CATC	(10,1) **(30,1)**
CCTA	(20,7) **(50,9)**
CGTG	(40,3) **(50,2)**
CTAG	(20, 8)(30,9) **(50,10)**
CGTC	(10,11) **(20,1)**
GATT	(40,15) **(50,5)**
GTGA	(40,4) **(50,3)**
GCGT	(10,10) **(50,1)**
GCTA	(30,8) **(40,8)**
TAGC	(20, 9) **(30,6)**
TCCT	(20, 6) **(50,8)**
TCGT	(30, 3) **(40, 2)**
TCTT	(10, 3) **(20,3)**
TGAT	(40, 14) **(50, 4)**
TTCC	(20,5)(40,17)**(50,7)**

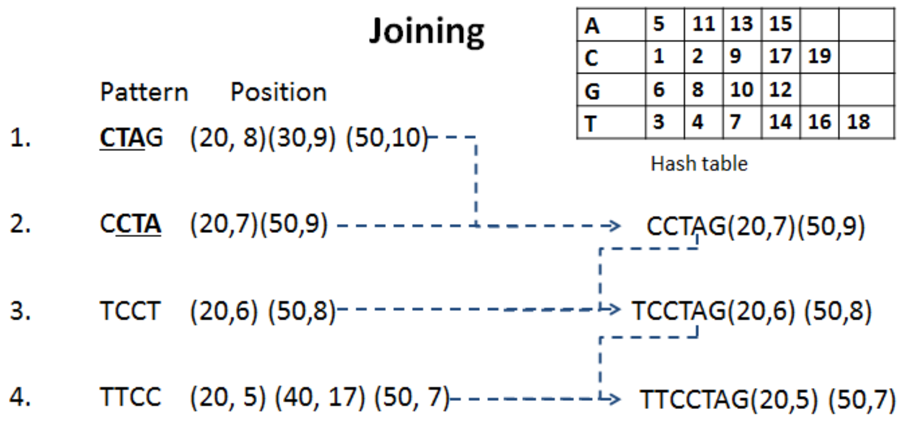
	Pattern	Position
1.	CTAG	(20, 8)(30,9) (50,10)
2.	CCTA	(20,7)(50,9)
3.	TCCT	(20,6) (50,8)
4.	TTCC	(20, 5) (40, 17) (50, 7)
5.	ATTC	(40,16) (50,6)
6.	GATT	(40,15)(50,5)
7.	TGAT	(40, 14) (50, 4)
8.	GTGA	(40,4)(50,3)
9.	CGTG	(40,3)(50,2)
10.	GCGT	(10,10)(50,1)
11.	ATCT	(10,2) (40,11)
12.	GCTA	(30,8)(40,8)
13.	AGCT	(20,10) (30,7) (40,7)
14.	TCGT	(30, 3) (40, 2)
15.	ATCG	(30,2) (40,1)
16.	TAGC	(20, 9)(30,6)
17.	CATC	(10,1) (30,1)
18.	TCTT	(10, 3) (20,3)
19.	CGTC	(10,11)(20,1)

2.3.4 Joining method

After sorting we need to join the subsequent patterns for generating the maximum length and other length frequent patterns. In our proposed approach in the joining method we consider our sorted first pattern element as suffix and the second one as prefix. Then we match the position of those pattern if they are adjacent then we joined those pattern to generate updated length frequent pattern. Except the prefix we consider all other elements as suffix and match the position whether they are adjacent or not. If adjacent then we joined those patterns and update the result. Sequentially we go through the whole sorted list and generate higher length frequent pattern. As we consider the first element as suffix and sequentially go through the sorted list. At the end all the elements of our list except the first one are considered as suffix and as prefix once. So according to our algorithm we have to consider the first element of our list as prefix also. So, after generating all the updated results by considering all the elements as suffix and prefix except first one we again come back to the first element to consider it as prefix. And then we match the position of the first element with all other updated result for joining.

According to our algorithm the first element of our sorted list occurred in the last position of our DNA database. So when we join those elements for generating higher length frequent pattern the joining occur only on one side. And it makes our joining unidirectional. But the first element can also appear in other positions in the sequences beside last position that gives the possibility for joining on both sides to generate higher length frequent pattern. So when we consider the first element as prefix and compare it with other updated results for joining, this remove the problem of skipping several higher length pattern for their possibility of joining on both sides.

And then we again follow the same steps for the updated result for generating maximum length frequent pattern by considering them as suffix and prefix sequentially. We use Apriori approach joining rule for this method. All the Items in the first pattern excluding the first item should be exactly the same as all the items of the second pattern excluding the last item. To join the same length frequent patterns, we adopt the Apriori joining rule. Thus, to join the frequent patterns of length-4 to generate the length-5 frequent patterns, all the items in the first pattern excluding the first item should be exactly same as all the items of second pattern excluding the last item. To increase the frequency counter of the generated next length pattern, both the patterns should be present in the same sequence and the second pattern's starting position should be the right next position to the first pattern in the same sequence. If the frequency counter satisfies the minimum support threshold, we consider the generated new frequent pattern in one of the next length frequent candidates. For example the joining process is given below



We will use updated result for further joining. If the joining does not match with the updated result then we will join it with it’s corresponding fixed length frequent pattern.

2.3.5 Final result with improved approach

Pattern & position	Updated result & position
1.CTAG(20, 8)(30,9) (50,10)	
2.CCTA(20,7)(50,9)	CCTAG(20,7)(50,9)
3.TCCT(20,6) (50,8)	TCCTAG(20,6) (50,8)
4.TTCC(20, 5) (40, 17) (50, 7)	TTCCTAG(20,5) (50,7)
5.ATTC(40,16) (50,6)	ATTCC(40,16) (50,6)
6.GATT(40,15)(50,5)	GATTCC(40,15) (50,5)
7.TGAT(40, 14) (50, 4)	TGATTCC(40,14) (50,4)
8.GTGA(40,4)(50,3)	X
9.CGTG(40,3)(50,2)	CGTGA(40,3) (50,2)
10.GCGT(10,10)(50,1)	X
11.ATCT(10,2) (40,11)	X
12.GCTA(30,8)(40,8)	X
13.AGCT(20,10) (30,7) (40,7)	AGCTA(30,7) (40,7)
14.TCGT(30, 3) (40, 2)	X
15.ATCG(30,2) (40,1)	ATCGT(30,2) (40,1)
16.TAGC(20, 9)(30,6)	TAGCT(20,9) (30,6)
17.CATC(10,1) (30,1)	X
18.TCTT(10, 3) (20,3)	X
19.CGTC(10,11)(20,1)	X

2.3.6 Proposed algorithm

Input: DNA database contains N DNA sequence (S1, S2,S3….Sn), Minimum Support Threshold Min_Sup

Parameters: 'N' is the number of sequences, fixed pattern length W

Output: maximum length frequent concatenate DNA subsequence

[1] Extract fixed length DNA subsequence by scanning the DNA database and construct fixed length pattern table

[2] For (i=0; i<N, i++)

//extract fixed length subsequences with position information using position based approach [10]

//store all the fixed length subsequence in the fixed length pattern table

[3] Sort the subsequences of the fixed length pattern table

//sort the sequence by observing its last occurring position

//create a hash table that contains that contains the positions of the subsequence starting with A, T, C and G respectively.

[4] Joining the subsequence to get the maximum length frequent concatenate DNA subsequence

//start joining from the second element of the pattern table

// take second subsequence as prefix pattern and search suffix pattern from the hash table.

// if the suffix pattern starts right next to the prefix pattern and their sequence id is adjacent then the frequency of the newly generated pattern will be incremented by one

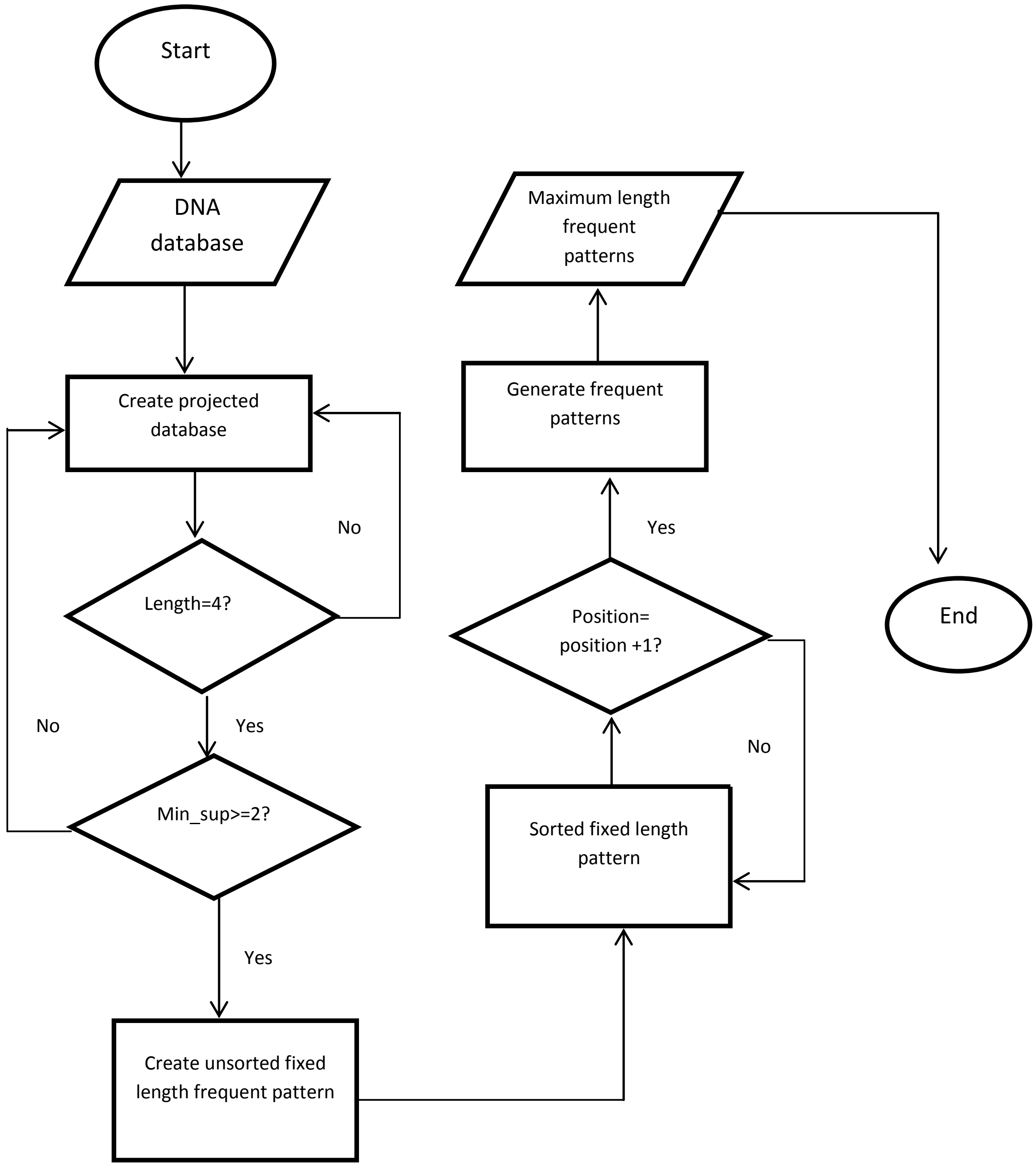
//if the frequency of the new subsequence is more than or equal to the Min_Sup, then store it along with its fixed prefix pattern

//First pattern is not joined because we started from second pattern considering it as prefix. Now loop will be started from top element of the pattern table to consider the first pattern as suffix and other patterns as prefix.

// Next time if its fixed prefix pattern is needed to be used as suffix pattern then use the newly generated pattern and generate the higher length pattern and check its support count like previous.

// by this way generate the upper length pattern until reaching the maximum length possible.

2.3.7 Flow chart of our proposed algorithm :


Start
DNA database
Create projected database
Length=4?
No
Yes
Min_sup>=2?
No
Yes
Create unsorted fixed length frequent pattern
Sorted fixed length pattern
Position= position +1?
No
Yes
Generate frequent patterns
Maximum length frequent patterns
End

2.3.8 Experimental & Performance Analysis

The proposed algorithm was applied on two datasets named HMR195 Dataset and BursetGuigo96 Dataset. HMR195 Dataset contains total 195 DNA sequences of human, mouse and rats. The mean length is 7096 bp. In BursetGuigo96 Dataset there are total of 570 sequences of vertebrate animals. Both of the DNA datasets were extracted from GenBank.

The algorithm was implemented in visual C++ 10 and run it on windows 7 platform with core2 duo 2.13 GHz CPU with a 4 GB of main memory.

Table 4 and Table 5 show the comparison of the run-time performance of different approaches to find out the maximum length pattern with various values of minimum support. The proposed algorithm was compared with 3 latest existing approaches. They are position based approach [10], position based approach with indexing, MacosVspan. From the table it can be seen that for minimum support threshold of 10%, the proposed algorithm took only 18.325 seconds where the latest Position based approach with indexing took more than 43.672 seconds. Other algorithms such as MacosVspan and Position based approach took 150.432 and 61.205 seconds to get the maximum length pattern respectively. The comparative analysis of the algorithms for BursetGuigo96 Dataset is shown in table 5. From both of the tables it can be seen that the proposed algorithm outperforms other algorithms.

Besides, the latest existing approaches like position based approach generally sort all the fixed patterns alphabetically. This sorting is very useful for the joining step from which higher length pattern was obtained. But sorting alphabetically is time consuming. The main contribution of our approach is, after getting the entire fixed length pattern; the patterns were sorted based on the last occurring position of that pattern. Since, the position information was stored initially, sorting is very easy and it saves time compared to the existing position based approach.

To make the process of comparison easier a graph based comparison is done. The comparison among different methods is graphically represented in Fig. 4 and, Fig 5.

Dataset descriptions:

No.	Name	Type	Number of sequence	Average sequence length	Maximum length
1.	HMR195	Human	30	28000	35000
2.	HM16R	Human	7	3001	3001
3.	HM01R	Human	18	2001	2001
4.	HM20R	Human	35	2001	2001

Support threshold (%)	MacosVspan (seconds)	Position based approach (seconds)	Positioned based approach with indexing (seconds)	Proposed approach (seconds)
10	150.432	61.205	43.672	**18.344**
20	102.694	46.672	31.792	**10.893**
30	78.548	33.456	23.162	**8.227**
40	56.542	24.673	18.567	**5.784**
50	39.746	19.898	12.480	**3.754**

Table 4: Performance comparison in terms of run time among proposed and other approaches for different support thresholds (for HM16R Dataset)

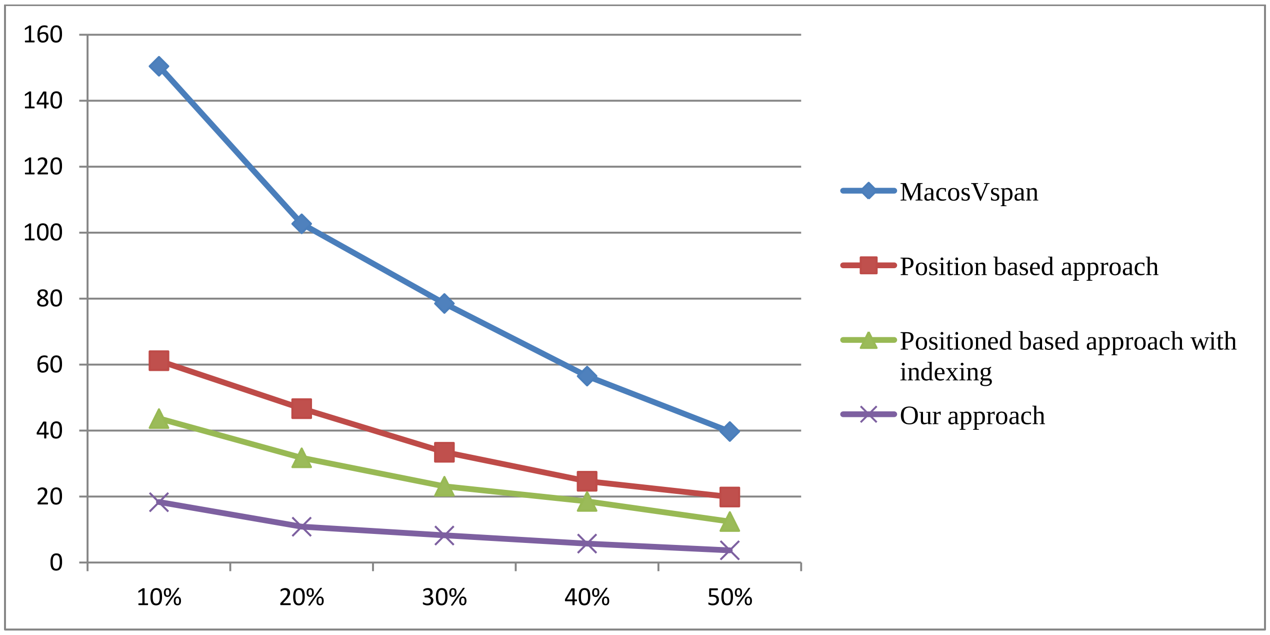


Fig 4: Performance comparison graph in terms of time among proposed approach and other approaches for different support thresholds (for HMR195 Dataset)

Support threshold (%)	**MacosVspan (seconds)**	**Position based approach (seconds)**	**Positioned based approach with indexing (seconds)**	**Proposed approach (seconds)**
10	713.315	634.895	533.859	**410.442**
20	570.129	450.483	392.474	**312.655**
30	325.314	312.540	297.582	**253.643**
40	173.243	161.469	157.361	**136.77**
50	67.214	56.983	49.543	**46.998**

Table 5: Performance comparison in terms of run time among proposed and other approaches for different support thresholds (for HMR195 Dataset)

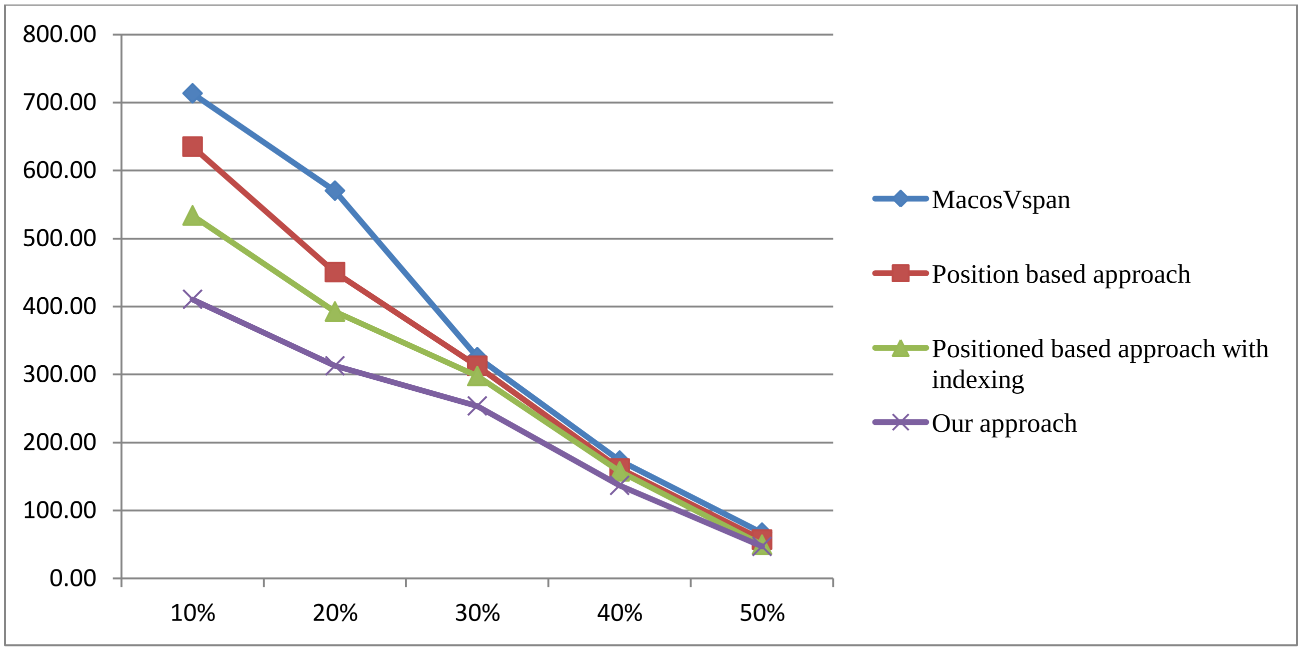


Fig 5: Performance comparison graph in terms of run time among proposed and other approaches for different support thresholds (HMR195 Dataset)

Support threshold(%)	**MacosVspan(seconds)**	**Position based approach(seconds)**	**Positioned based approach with indexing(seconds)**	**Proposed approach(seconds)**
10	176.234	88.211	56.987	**31.763**
20	97.376	62.115	35.22	**22.122**
30	63.187	39.189	23.227	**15.232**
40	41.765	22.755	15.221	**10.225**
50	29.292	16.198	11.477	**6.187**

Table 6: Performance comparison in terms of run time among proposed and other approaches for different support thresholds (for HM01R Dataset)

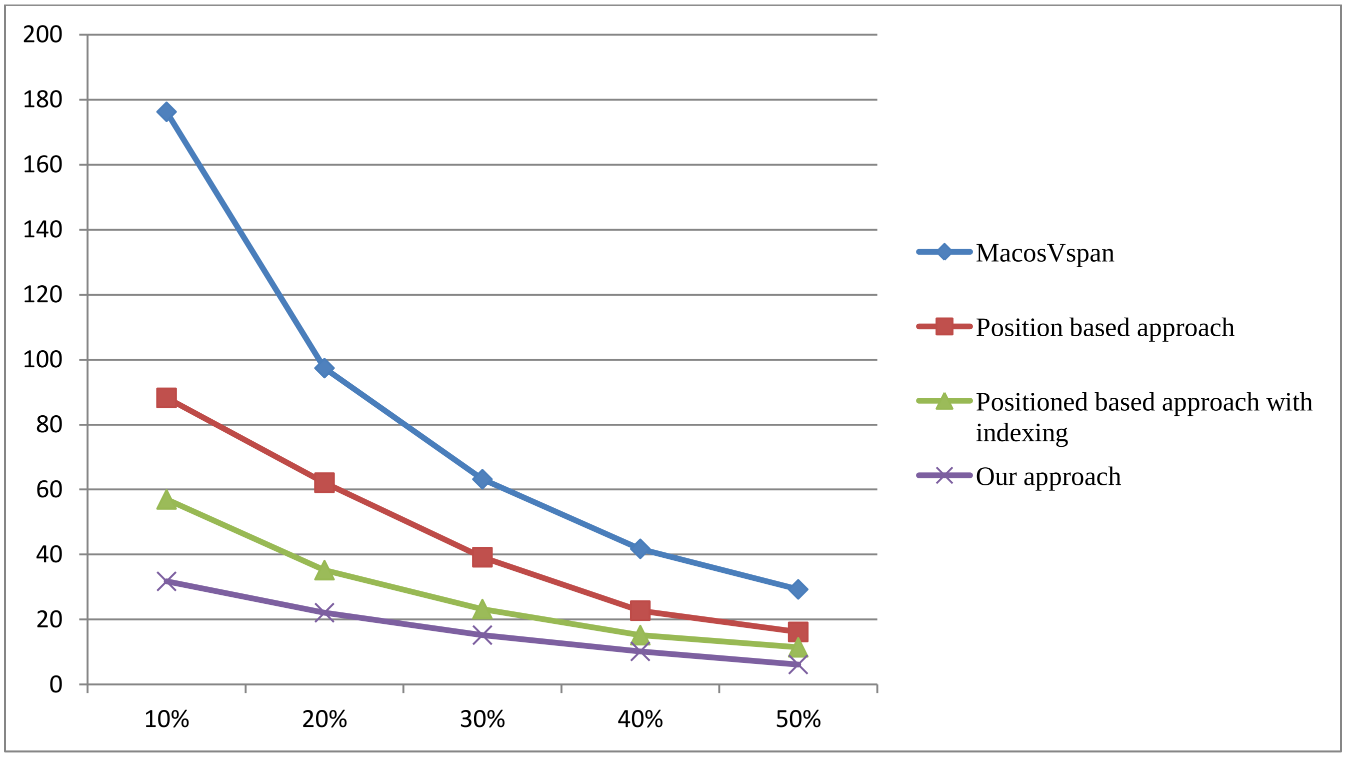


Fig 6: Performance comparison graph in terms of run time among proposed and other approaches for different support thresholds (HM01R Dataset)

Support threshold(%)	**MacosVspan(seconds)**	**Position based approach(seconds)**	**Positioned based approach with indexing(seconds)**	**Proposed approach(seconds)**
10	150.432	61.205	43.672	**18.344**
20	102.694	46.672	31.792	**10.893**
30	78.548	33.456	23.162	**8.227**
40	56.542	24.673	18.567	**5.784**
50	39.746	19.898	12.480	**3.754**

Table 7: Performance comparison in terms of run time among proposed and other approaches for different support thresholds (for HM16R Dataset)

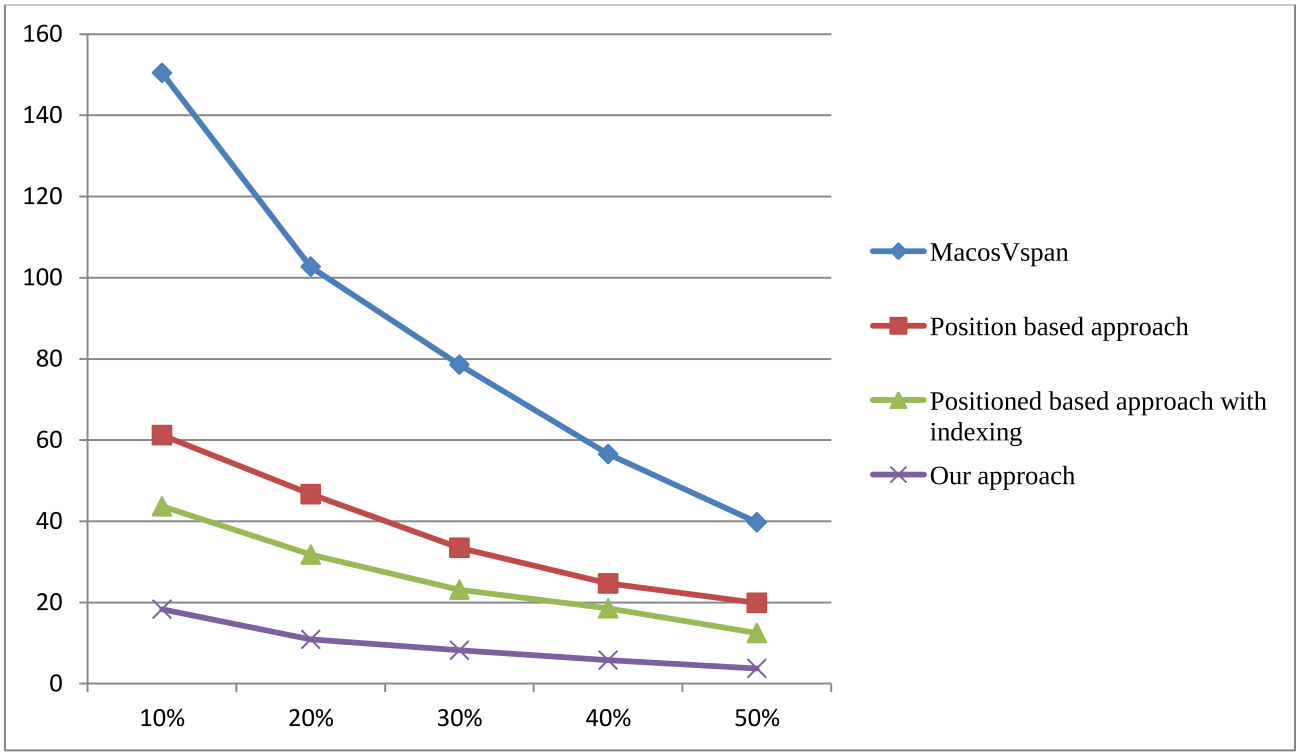


Fig 6: Performance comparison graph in terms of run time among proposed and other approaches for different support thresholds (HM16R Dataset)

Chapter 3

Progress Report

In this section we will try to provide concise overview of what we have implemented so far, the algorithms and other working procedures and how we have integrated different portions of the total work.

3.1 Improved areas

So far we have improved the existing frequent pattern mining techniques in several ways. Existing methods require a lot of memory space and require a lot of time for processing the DNA sequences. We implement a sorting and merging method that decrease the memory size and also reduce processing time.

The key areas we have improved are given below:

- Non frequent candidates are avoided
- Scan the main database only once
- Find the maximum length and other closed frequent pattern sequence simultaneously in a single scan
- Previous approaches generates 35 patterns for finding maximum length frequent we generate 29 patterns at one scan

3.2 Future work

In future the proposed approach can be optimized by adjusting different parameters and applying the proposed algorithm in other biological sequence datasets

Our future works will be these

- Optimizing our proposed algorithm by adjusting different parameters
- Applying our algorithm in other biological sequence datasets

Chapter 4

Conclusion

Mining Maximal Adjacent Frequent Patterns from DNA Sequences using Location Information is an efficient approach to mine the maximum length frequent pattern in DNA database is proposed where location information is used. A new sorting technique is introduced to mine maximum length frequent patterns within a single calculation. Compared to other existing techniques, the proposed approach generates less number of patterns and also takes less time to mine the maximum length frequent pattern.

Chapter 5

Simulation Code

```
#include <string.h>

#include <stdio.h>

#include <iostream>

#include <fstream>

#include <string>

#include <sstream>

#include <vector>

#include <utility>

#include <algorithm>


using namespace std;


void charfinder(char c);

void unsorted(char c);

void checkdatabase(std::string str);

void sorted();

void createhashtable();

void final1();


int main ()

{


   charfinder('A');
```

```
  charfinder('T');
  charfinder('C');
  charfinder('G');
  unsorted('A');
      sorted();
      createhashtable();
      final1();


  return 0;
}
void charfinder(char c){
  char buffer[5000];
  std::ifstream file("example.txt",ios::in);
  std::ofstream out;

  if(c=='A')
  {
    out.open("outputA.txt",ios::app);
  }
  else if(c=='T')
  {
    out.open("outputT.txt",ios::app);
  }

  else if(c=='C')
```

```
{
    out.open("outputC.txt",ios::app);
}
else
{
    out.open("outputG.txt",ios::app);
}

int count=0;

while(!file.eof())
{

    count++;
    std::string temp;
                std::string str2;
    std::getline(file, temp);
    std::size_t length = temp.copy(buffer,temp.length(),0);
    buffer[length]='\0';



    for(int i=0;i<=length;i++)
    {
        if(buffer[i]==c)
        {
                str2 = temp.substr (i+1,length-i-1);

                out<<str2<<endl;

            }

        }


    }

    out.close();


}


void unsorted(char c)

{

            std::ifstream file("outputA.txt",ios::in);

            vector<string> geneList;


            while(!file.eof())

    {


              std::string temp;

              std::string str;

              std::getline(file, temp);

              //cout<<temp;

              int flag=0;


              if(temp!=""&&temp.length()>=3)
        {
                str=c+temp.substr(0,3);
                for(int k=0;k<geneList.size();k++)
                {
                        if(str.compare(geneList.at(k))==0)
                        {
                                flag=1;
                                break;
                        }


                }
                if(flag==0)
                {
    checkdatabase(str);
                        geneList.push_back(str);
                }


        }


    }


}


void checkdatabase(std::string str)
{
        std::ifstream file("example.txt",ios::in);

        std::ofstream out;

        out.open("unsorted.txt",ios::app);

        vector<std::string> geneList;


        int count=0;

  int matchcount=0;

        std::string temp2=str;

        int flag;


        std::string temp;


        while(!file.eof())

        {

                count++;

                stringstream ss;


    ss << count;

                int j=0;

    flag=0;


    std::getline(file, temp);

                int size=temp.length();


                for(int i=0;i<(size-3);i++)
        {


    stringstream sss;
            if(temp.substr(j,4).compare(str)==0)
            {
                sss << j;
                matchcount++;
                temp2=temp2+" ("+ss.str()+","+sss.str()+") ";



            }
            j++;
        }




    }
    if(matchcount>1)
        {
            out<<temp2<<endl;
        }




    out.close();
```

```
}


void sorted()

{

   std::ifstream file("unsorted.txt",ios::in);

   std::ofstream out;

   out.open("sorted.txt",ios::app);

   std::string temp;

   vector< pair<int,int> > a;

   vector<std::string> b;




   while(!file.eof())

        {

                  std::getline(file, temp);

                  b.push_back(temp);


                  int index1=temp.find_last_of("(");

                  int index2=temp.find_last_of(")");

                  int index3=temp.find_last_of(",");


                  int num1=atoi(temp.substr(index1+1,(index3-index1-1)).c_str());

                  int num2=atoi(temp.substr(index3+1,(index2-index3-1)).c_str());


                  a.push_back(make_pair(num1,num2));
}

    std::sort(a.begin(),a.end());
                std::string temp2;
                for(int k=a.size()-1;k>=0;k--)
                {
                        stringstream ss;
        stringstream sss;
                        std::pair <int,int> test = a.at(k);
                        ss << test.first;
                        sss << test.second;
                        temp2="("+ss.str()+","+sss.str()+")";
                        for(int l=0;l<b.size();l++)
                        {
                                if((strstr(b.at(l).c_str(),temp2.c_str())!=NULL))
                                {
                                        out<<b.at(l)<<endl;
                                        //b.erase(b.begin()+l);
                                        break;
                                }

                        }
                }
                out.close();
}


void createhashtable()
{
        std::ifstream file("sorted.txt",ios::in);
        std::string temp;
        vector<int> A;
        vector<int> T;
        vector<int> C;
        vector<int> G;
   int count=0;


        while(!file.eof())
        {
                count++;
                std::getline(file, temp);
                if(temp.substr(0,1).compare("A")==0)
                {


                        A.push_back(count);
                }
                if(temp.substr(0,1).compare("T")==0)
                {
                        T.push_back(count);
```

```
		}
		if(temp.substr(0,1).compare("C")==0)
		{
			C.push_back(count);
		}
		else
		{
			G.push_back(count);
		}

	}



	std::pair <char,vector<int> > pA;
	std::pair <char,vector<int> > pT;
	std::pair <char,vector<int> > pC;
	std::pair <char,vector<int> > pG;

	pA=std::make_pair('A',A);
	pT=std::make_pair('T',T);
	pC=std::make_pair('C',C);
	pG=std::make_pair('G',G);

	//to check the output
	/*for(int i=0;i<pA.second.size();i++)
```

```
        {
                cout<<pA.first<<" "<<pA.second.at(i)<<endl;
        }*/
}
void final1(){
        freopen("sorted.txt","r+",stdin);
        freopen("final.txt","w+",stdout);
        int i,n=0,n3=0,t,k,x,y,z,j,first1,first2,second1,second2,cnt,flag;
        string temp,seq1,seq2,maxseq;
        char a[10000],num1[200],num2[200],newseq[300];
        vector <string> v[10000],ss;
        vector <int> vi[10000],v1,v2,maxv;
        while(gets(a))
        {
                stringstream s(a);
           while( s>>temp )                    v[n].push_back(temp);
                n++;
        }
        for(i=0;i<n;i++){
                int n1=v[i].size(); //cout<<v[i][0]<<" ";
                for(x=1;x<n1;x++){
                        k=0; temp=v[i][x];
                        for(y=1;temp[y]!=',';y++)
                                num1[k++]=temp[y];
                        num1[k]='\0';
                        first1=atoi(num1); k=0;
                        for(z=y+1;temp[z]!=')';z++)
                                num2[k++]=temp[z];
                        num2[k]='\0'; second1=atoi(num2);
                        vi[i].push_back(first1);  vi[i].push_back(second1);
                        //printf("%d,%d ",first1,second1);
                }
                //printf("\n");
        }
        printf("**********\n");
        for(i=0;i<n;i++){
                int n1=vi[i].size();
                seq1=v[i][0];
                //cout<<seq1<<" ";
                for(j=0;j<n;j++){
                        int n2=vi[j].size();
                        seq2=v[j][0];
                        if(i==j) continue;
                        cnt=0;
                        vector <int> last;
                                for(x=0;x<n1;x+=2){
                                        first1=vi[i][x];
                                        second1=vi[i][x+1];
                                        flag=0;
                                        for(y=0;y<n2;y+=2){
```

```
                                first2=vi[j][y];
                                second2=vi[j][y+1];
                                if(first1==first2 && second1+1==second2){
                                        last.push_back(first1);
                                        last.push_back(second1);
                                        flag=1; break;
                                }
                        }
                        if(flag) cnt++;
                }

        if(last.size()>=4){
                k=0;  newseq[k++]=seq1[0];
                for(z=0;z<seq2.length();z++)
                        newseq[k++]=seq2[z];
                newseq[k]='\0'; temp=newseq;
                if(temp.length()>n3){
                        maxv.clear();
                        maxseq=newseq;
                        n3=temp.length();
                        for(x=0;x<last.size();x+=2){
                                maxv.push_back(last[x]);
                                maxv.push_back(last[x+1]);
                        }
                }
```

```
                        v[i][0]=newseq;
                        cout<< v[i][0]<<"  ";
                        for(x=0;x<last.size();x+=2){
                                cout<<last[x]<<","<<last[x+1]<<"  ";
                        }
                        /*v[n].push_back(temp);
                        cout<<v[n][0]<<" ";
                        for(x=0;x<n1;x+=2){
                                vi[n].push_back(vi[i][x]);
                                vi[n].push_back(vi[i][x+1]);
                                cout<<vi[n][x]<<","<<vi[n][x+1]<<"  ";
                        }*/
                     printf("\n"); break;
                    }
                }
        }
        printf("*********\n");
        cout<< maxseq<<"  ";
        for(x=0;x<maxv.size();x+=2){
                    cout<<maxv[x]<<","<<maxv[x+1]<<"  ";
        }
        printf("\n");
}
```

References

[1] Shuang Bai, Si-Xue Bai, “The Maximal Frequent Pattern Mining of DNA Sequence”, GrC, pp 23-26, 2009.

[2] T.H Kang, J.S Yoo and H, Y Kim, “Mining frequent contiguous sequence patterns in biological sequences”, in proceeding of the 7th IEEE International Conference on Bioinformatics and Bioengineering, pp 723-8, 2007.

[3] R. Wanger and M. Fischer “The string-to-string Correction Problem” J. of the ACM (JACM), Vol 21, No 1, pp.168-173, 1974.

[4] D. Hirschberg “Algorithms for the longest common subsequence problem” J. of the ACM (JACM). Vol 24, No 4, pp.664-675, 1977.

[5] M. Garofalakis, R. Rastogi, and K. Shim, “Spirit: Sequential pattern mining with regular expression constraints.” In Proc. 1999 Int. Conf. Very Large Data Bases (VLDB’99), pages 223–234, Edinburgh, UK, Sept. 1999.

[6] R. Srikant and R. Agrawal, “Mining sequential patterns: Generalizations and performance improvements.” In Proc. 5th Int. Conf. Extending Database Technology (EDBT’96), pages 3–17, Avignon, France, Mar. 1996.

[7] R. Agrawal and R. Srikant, “Fast algorithms for mining association rules.” In Proc. 1994 Int. Conf. Very Large Databases (VLDB’94), pages 487–499, Santiago, Chile, Sept. 1994.

[8] J. Pei, J. Han, B. Mortazavi-Asl, Q. Chen, U. Dayal, and M.C. Hsu, “PrefixSpan: Mining sequential patterns efficiently by prefix-projected pattern growth.” InICDE’01, Germany, April 2001.

[9] J. Pan, p. Wang, W. Wang, B. Shi and G. Yang, “Efficient algorithms for mining maximal frequent concatenate sequences in biological datasets”, in proceeding of the fifth International Conference on Computer and Information Technology (CIT), , pp 98-104, 2005.

[10] . Zerin SF, Ahmed CF, Tanbeer SK, Jeong BS, “A fast in-dexed-based contiguous sequential pattern mining tech-nique in biological data sequences.” In: Proceeding of 2nd International Conference on Emerging Databases (EBD’10), 2010 Aug 30-31, Jeju.

[11] Rashid MM, Karim MR, Hossain MA, Jeong BS, “An ef-ficient approach for mining significant contiguous fre-quent patterns in biological sequences.” In: Proceeding of 3rd International Conference on Emerging Databases (EBD’11), 2011 Aug 25-27, Incheon

[12] Rashid MM, Karim MR, Hossain MA, Jeong BS and -Jin Choi, “Efficient Mining of Interesting Patterns in Large Biological Sequences.” Genomics & Informatics Vol. 10(1) 44-50, March 2012

[13] Jiawei Han & Micheline Kamber , “Data Mining Concepts and Techniques”, Elsevier, 2006.

[14] Pan-Ning Tan, Vipin Kumar, Michael Steinbach , “ Introduction to Data Mining”, Pearson Education Inc, 2006

[15] Ye-In Chang, Chen-Chang Wu, Jiun-Rung Chen and Yin-Han Jeng, “Mining Sequence Motifs from Motif Databasesbased on a Bit Pattern Approach”, International Journal of Innovative Computing, pp. 647-657,2012 .

[16] J. Yang, W.Wang, P.S Yu and J.Han, “Mining long sequential patterns in a noisy environment”,SIGMOD, 2002