ISSN 1870-4069

Sequential Frequent Patterns for a DNA Sequence Using Mapping and Mining Techniques

Luis Heriberto García-Islas, Anilu Franco-Arcega, Víctor Ignacio Sobrevilla-Solís, Esteban Rueda-Soriano, Kristell Daniella Franco-Sánchez

Universidad Autónoma del Estado de Hidalgo,
Instituto de Ciencias Básicas e Ingeniería,
Área Académica de Computación y Electrónica,
Mexico

{luishg, afranco, so314246, estebanrs, kristell_franco}@uaeh.edu.mx

Abstract. Biological sequences contain a significant amount of genetic information from living organisms. The analysis of these sequences can provide information that might help biologists better understand them. The discovery of frequent patterns from a specific DNA sequence has become one of the greatest challenges in the application of data mining techniques. This is especially true for those sequences whose length is extensive and/or the number of frequent patterns generated exceeds the time needed to fully reveal themselves. There is a considerable time and effort for obtaining sequential frequent patterns when the methods utilized are based on Apriori algorithms such as GSP or Key-segment. However, these methods can be enhanced and improved. In this paper, we propose a sequence mapping-based algorithm designed to improve the search for contiguous frequent patterns in a single DNA sequence. Our experiment was applied over 11,230 DNA real different sequences with lengths between 118 and 52,255 nucleotides, obtained from a real biological database. This experiment demonstrated a faster algorithm for frequent pattern mining on DNA sequences compared with other related algorithms.

Keywords: Data mining, sequential pattern mining, frequent contiguous patterns, DNA sequences, bioinformatics.

1 Introduction

Large DNA sequences are often composed of several short frequent subsequences that play important functional or structural roles in a living organism [17]. One way to study this is through Frequent Pattern matching [1], which has become a relevant area of research for bioinformatics [4-8, 12, 13, 19-22]. Even though in recent years, several approaches to identify frequent patterns on sequences have been proposed [2, 14], some of them must scan the sequence multiple times to obtain even only the subsequences frequency, which increases the computational effort to read the sequences.

To reduce this complexity, other algorithms perform non-exhaustive searches because they are only able to find fixed-length frequent patterns [9, 15]. However, there is no guarantee that major additional relevant frequent patterns can be found.

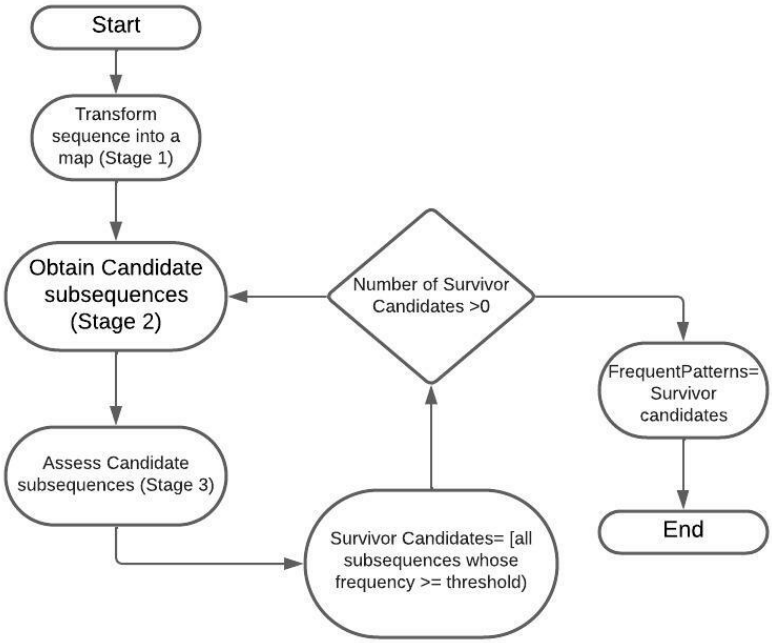

Fig. 1. General approach for proposed frequent pattern algorithm.

Finally, some algorithms can only validate whether a subsequence is a frequent pattern or not [10].

For the algorithms that perform exhaustive searches, there are the classic algorithms such as PrefixSpan, Spade, SPAM, and GSP [2] which perform searches using algorithms such as Naive, KMP or Boyer Moore [11]. New approaches have appeared and demonstrate better performance than classic algorithms, such as Key-segment introduced by Mao [14], which allows the use of a compact data structure to be retained in memory resulting from sequences scanning.

The Mao's algorithm is based on GSP [3] and is used to mine key segments from long DNA sequences. This method uses an exhaustive search to identify frequent patterns, thus the results are more effective than more classic techniques. However, this operation still consumes considerable time to perform its analysis since the sequence must be scanned every time in accordance with the number of occurrences in order to obtain each frequent subsequence.

Despite several researches design to determine frequent patterns, using enhanced algorithms to determine variable length frequent DNA sequence patterns, a significant challenge remains. In this paper, we present an algorithm where the frequent subsequent nucleotide sequences can be identified within a single DNA sequence using a novel mapping technique.

Obtaining these kind of patterns in a single DNA sequence is important because it can establish the basis for discovering more complex behaviors, such as motifs. The most frequent subsequence patterns with variable length will result as output for this

Algorithm 1 Map generation

INPUT: sequence - sequence to be mapped
OUTPUT: map - the obtained map

```
1: function GENERATEMAP(sequence)
2:     for each nucleotide in sequence do
3:         map[char][lastItem] ← [Pos, nextChar]
4:     end for
       return map
5: end function
```

proposed method. The emphasis on finding repeated nucleotide subsequences with different sizes promises to have a significant and positive impact in many fields of genetics and bioinformatics.

The remainder of the paper is organized as follows. Section 2 introduces the proposed algorithm and describes in detail each stage. The experimental results and a comparison with other related algorithms are discussed in section 3. Finally, in section 4 conclusions are presented.

2 Proposed Algorithm

To identify frequent patterns in a single DNA sequence, the proposed algorithm is formed by three stages: sequence mapping, candidate subsequences generation and the assessment for those candidate subsequences. Figure 1 presents these stages. The first stage creates a map from the sequence that can be utilized to perform a fast search to obtain frequencies of subsequences.

Then, the stages that follow will iterate the frequent patterns. In these iterations, the generation of candidates is performed (stage 2), and the process follows by obtaining their number of occurrences (stage 3). During these iterations, all those candidate subsequences whose occurrence numbers are greater or equal to an established threshold will be used as a source to create new candidate subsequences.

Stages 2 and 3 will be iterated until the number of candidates, that fulfill the threshold condition, becomes zero.

2.1 Sequence Mapping

As part of the proposed process, the first step consists of transforming the DNA sequence into a map represented as a table. In this tabular abstraction, the rows represent each different element in the sequence, i.e. a row for each nucleotide (A, C, G, T).

Every row stores a set of pairs formed as following: (Pos, nextChar), where Pos represents the position of the nucleotide within the sequence and nextChar represents the next element. The Algorithm 1 shows the process of creating the map.

The resulting map will be used to obtain the frequency for every possible candidate that will be generated in next stage.

Algorithm 2 Candidate generation

INPUT: CandidateSubsequences - the set of candidates that will be used as base to generate new ones
OUTPUT: NewCandidates - the obtained new candidates

```
1: function GENERATECANDIDATES(CandidateSubsequences)
2:     alphabet ← [A, C, G, T]
3:     for each CandidateSubsequence do
4:         for each letter in alphabet do
5:             NewCandidate ← CandidateSubsequence + letter
6:             if NewCandidate[2:length(NewCandidate)] exists in CandidateSubsequences
   then
7:                 NewCandidates ⇒ append(NewCandidate)
8:             end if
9:         end for
10:    end for
       return NewCandidates
11: end function
```

2.2 Candidate Subsequence Generation

This stage is performed in two steps: the initial stage and the followed by iterative candidate generation. The first occurs right after stage 1 is concluded and requires the initial candidate generation, which consists of creating 2^4 2-length candidates by using the nucleotides alphabet.

The second will be an iterative process. For each iteration i, the generation process uses the n-length survivor candidates of iteration *i-1*, which are obtained in stage 3, in order to create new ones.

For each survivor candidate, this step will generate (i+1)-length possible new candidates by adding all of the chars from the $alphabet$, to each one, i.e. if "AA" is a survivor candidate, four new candidates will be created as {"AAA","AAC","AAG","AAT"}. Then, every possible new candidate will be evaluated by using anti-monotone property of support [18] which is applied to avoid generating unnecessary candidates.

The process of generating subsequence candidates can be observed in Algorithm 2.

2.3 Candidate Subsequence Assessment

Once all new candidates have been created, the next stage will consist of the candidate subsequences assessment. To perform this, a novel approach for obtaining their frequency is proposed. This can be observed in Algorithm 3 where a process is employed to locate all pairs $(pos, nextChar)$ in the map, traveling the row corresponding with every char from candidate subsequence to calculate how many times they appear in the sequence.

This number will represent the number of occurrences, i.e. the frequency for each candidate subsequence. Then, the next step is to create the set of survivor candidates to be used in the next iteration. A candidate survives if $f_{support}(candidate\_sequence) \geq Threshold$. When this set is empty, the algorithm has completed its cycle.

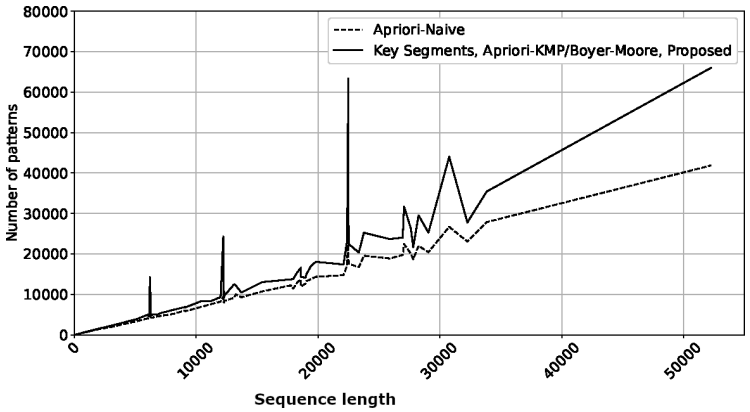

Fig. 2. Sequence length and number of obtained contiguous patterns.

3 Experiments and Performance Evaluation

Some experiments are shown to validate the performance of the proposed algorithm. The algorithm was tested over 11,230 sequences of different lengths between 118 and 52,255 nucleotides from 550 organisms, for example Chikungunya virus, Cactus, Xenopus laevis, among others. These sequences were downloaded from the NCBI repository [16].

We compared our proposal with algorithms based on Apriori using different search methods (Naive, KMP and Boyer-Moore), and with key-segment algorithm. The reason to use these algorithms is because they can be used to obtain frequent patterns contained in only one sequence, unlike other algorithms, such as fp-tree based algorithms that requires a set of sequences to obtain frequent patterns. All of the algorithms were programmed with Python 2.7 and for these experiments it was considered the threshold with a value of 2, because it obtains the whole set of frequent patterns. If the threshold increases his value then the length of the set of frequent patterns will be decreased.

The experimental results show that the proposed algorithm obtained the same amount of patterns than Apriori-KMP, Apriori-Boyer Moore and Key-segments for the 100% of the tested sequences as it can be seen on Figure 2. In particular, Apriori-Naive obtains less frequent patterns than the other algorithms. The reason of this is because it doesn't consider when patterns with same nucleotides appears on contiguous elements, i.e. pattern "AA" on sequence "AAATC", for Naive algorithm, has a frequency of one instead of two.

The efficacy of the proposed algorithm resides in the identification and utilization of a novel method to obtain frequent patterns via a structured mapping search, which consumes less processing time than related algorithms.

The number of patterns for all cases is the same, except for Apriori-Naive but, the major difference is in the processing time needed to obtain them. Table 1 indicates the processing time required by the tested algorithms, our proposed algorithm evidenced the fastest processing times. In the longest sequence tested, there were significant

Algorithm 3 Frequency obtaining

INPUT:

- pattern: whose frequency will be obtained.
- StartPosition: allows the method to find the path of search on a defined position of the pattern.
- positionsArray: enables an positions array whose sequence[pos] are part of the pattern

OUTPUT: length(pos) is the obtained frequency of the candidate to be assessed and pos is the array containing the positions of the substrings corresponding with the assessed pattern

```
1: function GETFREQUENCY(pattern, startPosition, positionsArray)
2:     frequency ← 0
3:     posAnt ← empty        ▷ Represents the array of positions of the previous iteration whose sequence[Pos] are part of the pattern
4:     pos ← empty           ▷ Represents the array of positions on the current iteration whose sequence[pos] are part of the pattern
5:     if length(pos) > 0 then        ▷ this initial iteration allows identify which positions in the selected row fits with the input pattern
6:         currentChar ← pattern[0]
7:         nextChar ← pattern[1]
8:         row ← row of currentChar in map
9:         pos ← row[x][1]∀x[2] = pattern[2]        ▷ Gets all the pairs[pos,nextChar] whose nextChar=pattern[1]
10:        indexPattern ← 2
11:    else        ▷ There is a sub pattern and it will be used as an start to complement
12:        pos ← positionsArray
13:        indexPattern = startPosition − 1
14:    end if
15:    while length(pos) > 0 do
16:        newPos ← empty        ▷ Represents the positions identified whose sequence[pos] are part of the pattern
17:        for each position in pos do
18:            if [position+1,pattern[indexPattern] exists in map[row] then
19:                newPos → append(position)
20:            end if
21:        end for
22:        pos ← newPos
23:        indexPattern ← indexPattern + 1
24:    end while
       return length(pos), pos
25: end function
```

improvements in the processing of 42.7, 46.9, 22.7 and 53.3 times faster compared with Apriori-Naive, Apriori-KMP, Apriori-Boyer Moore and Key Segments, respectively.

The execution time for all the algorithms with the complete set of sequences is exhibited in Figure 3. The average process improvement times of our algorithm were 46.16, 43.75, 24.95 and 39.16 over Apriori-Naive, Apriori-KMP, Apriori-Boyer Moore and Key-segments, respectively.

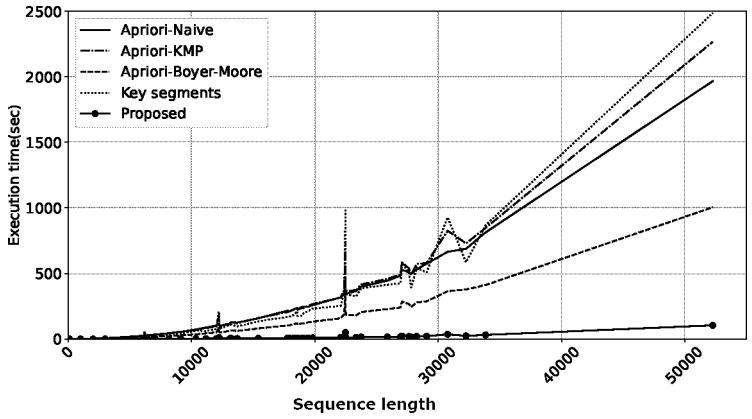

Fig. 3. Processing time to obtain frequent patterns for different length sequences.

As we can see in the previous figure, the maximum improvement for each algorithm was of 59.59, 72.24, 41.34 and 98.53 (on the same order of algorithms). This indicates a significant enhancement when the proposed algorithm is applied. Furthermore, the behavior for the other algorithms presents an irregular increment in execution time for several sequences.

This behavior is related to the number of patterns that a sequence contains combined with its length, as indicated in Table 1. This table only shows a summary of the whole set of experiments performed, there are 14 of 11,230 executions. The summary of Table 1 demonstrates that while larger is the length of the DNA sequence more execution time requires to complete the discovery of patterns.

The proposed algorithm presents a minimum variation of the execution time even when these cases are processed. This means that execution time for our algorithm is not related to the number of patterns but to the sequence length, while the time for the comparison algorithms is associated with both, the length and number of patterns contained in the sequence.

4 Conclusions

In this paper, a novel algorithm to improve frequent pattern generation for a single DNA sequence is proposed. Greater speed and efficacy are obtained through transformation of the sequence by way of a map, and then applying the map in combination with the well known anti-monotone property of support to obtain these frequent patterns.

In addition, a new way to search the number of occurrences for a subsequence using the created map is proposed. Such map is obtained by scanning the entire DNA sequence only once with the application of the algorithm.

Then, it allows the identification of the number of occurrences of a subsequence into a DNA sequence using the improved proposed search, avoiding the need to scan the entire sequence and even the entire subsequence every time it is evaluated. In addition, the use of the anti-monotone property reduces the number of possible frequent subsequence candidates and hence, the number of iterations.

Table 1. Summary of execution time obtained through experimental results with different lengths DNA sequences, in seconds.

Sequence ID	length	Apriori-Naive	Apriori KMP	Apriori-BoyerMoore	Key Segment	Proposed
HL714398	118	0.0159	0.0150	0.0160	0.0160	**0.0016**
NM_007169	1008	1.0160	0.8910	0.5779	0.6099	**0.0460**
NM_053477	2020	4.0900	3.5320	2.2260	2.6570	**0.1719**
NM_001322998	3000	9.2190	8.1139	4.8289	5.7829	**0.3280**
NM_022009	4071	11.4390	0.5160	16.7109	14.4429	**8.6890**
XM_011544263	5103	25.2250	22.9159	13.6380	17.679	**0.6879**
NM_001310478	6037	38.0969	33.4040	18.5870	27.5810	**1.5469**
NM_001102653	7068	48.6870	43.6400	25.8659	32.9810	**1.1870**
NM_002763	8178	65.7319	58.5820	34.5220	45.8090	**1.4220**
NR_133925	9096	80.4949	72.8770	41.7019	54.0269	**1.7500**
NG_029868	10355	101.7990	93.2639	53.5950	75.2279	**2.1099**
NG_013266	22093	488.1619	440.3600	246.3190	365.9060	**9.2349**
NG_011731	30767	941.9620	1100.0530	532.6480	1451.1100	**19.2990**
NG_008111	52255	2735.4070	3007.2460	1456.9200	3413.6280	**64.4760**

Execution time is reduced as well. Experimental results confirm enhanced performance when our algorithm is applied in contrast to others that process only one DNA sequence. A major finding of this study is that the performance of comparative algorithms relies upon both, length and number of patterns obtained from a sequence, while our algorithm does not, regardless of the number of patterns. According to the experiments performance, it can be established that the proposed algorithm is faster than compared algorithms. Furthermore, frequent pattern studies can be enhanced by using this proposed method.

References

1. Aggarwal, C.C.: Data mining: The Textbook. Springer International Publishing, vol. 1, pp. 1–734 (2015). DOI: 10.1007/978-3-319-14142-8.
2. Aggarwal, C.C., Han, J.: Frequent Pattern Mining, Springer Cham (2014). DOI: 10.1007/978-3-319-07821-2_1.
3. Agrawal, R., Srikant, R.: Fast Algorithms for Mining Association Rules in Large Databases. In: Proceedings of the 20th International Conference on Very Large Data Bases, vol. 1215, pp. 487–499 (1994)
4. Azmi, A.M., Al-Ssulami, A.M.: Discovering Common Recurrent Patterns in Multiple Stringsover Large Alphabets. Pattern Recognition Letters, vol. 54, pp. 75–81 (2015). DOI: 10.1016/j.patrec.2014.12.009.
5. Beernaerts, J., Debever, E., Lenoir, M., Baets, B.D., de-Weghe, N.V.: A Method Based on the Levenshtein Distance Metric for the Comparison of Multiple Movement Patterns Described by Matrix Sequences of Different Length. Expert Systems with Applications, vol. 115, pp. 373–385 (2019). DOI: 10.1016/j.eswa.2018.07.076.
6. Bustio-Martínez, L., Muñoz-Briseño, A., Cumplido, R., Hernández-León, R., Feregrino-Uribe, C.: A Novel Multi-Core Algorithm for Frequent Itemsets Mining in Data Streams. Pattern Recognition Letters, vol. 125, pp. 241–248 (2019). DOI: 10.1016/j.patrec.2019.05.003.
7. Chanda, A.K., Ahmed, C.F., Samiullah, M., Leung, C.K.: A New Framework for Mining Weighted Periodic Patterns in Time Series Databases. Expert Systems with Applications, vol. 79, pp. 207–224 (2017). DOI: 10.1016/j.eswa.2017.02.028.

8. Danger, R., Pla, F., Molina, A., Rosso, P.: Towards a Protein–Protein Interaction Information Extraction System: Recognizing Named Entities. Knowledge-Based Systems, vol. 57, pp. 104–118 (2014). DOI: 10.1016/j.knosys.2013.12.010.
9. Devikarubi, R., Rubi, R.D., Arockiam, L.: IndexedFCP - An Index Based Approach to Identify Frequent Contiguous Patterns (FCP) in Big Data. In: International Conference on Intelligent Computing Applications, pp. 27–31 (2014). DOI: 10.1109/ICICA.2014.15.
10. Gureja, V., Sharma, N., Sharma, A.: An Optimized Tabular Structure Based Pattern Search Over DNA String. In: International Conference on Soft Computing Techniques and Implementations, pp. 72–76 (2015). DOI: 10.1109/ICSCTI.2015.7489540.
11. Gusfield, D.: Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology. Cambridge University Press, (1997). DOI: 10.1017/CBO9780511574931.
12. Kieu, T., Vo, B., Le, T., Deng, Z.H., Le, B.: Mining Top-k Co-Occurrence Items with Sequential Pattern. Expert Systems with Applications, vol. 85, pp. 123–133 (2017). DOI: 10.1016/j.eswa.2017.05.021.
13. Maestre-Vidal, J., Sotelo-Monge, M.A., García-Villalba, L.J.: A Novel Pattern Recognition System for Detecting Android Malware by Analyzing Suspicious Boot Sequences. Knowledge-Based Systems, vol. 150, pp. 198–217 (2018). DOI: 10.1016/j.knosys.2018.03.018.
14. Mao, G.: An Efficient Mining Algorithm for Key Segment from DNA Sequences. In: 28th Canadian Conference on Electrical and Computer Engineering, pp. 396–399 (2015). DOI: 10.1109/CCECE.2015.7129310.
15. Mutakabbir, K.M., Mahin, S.S., Hasan, M.A.: Mining Frequent Pattern Within a Genetic Sequence Using Unique Pattern Indexing and Mapping Techniques. In: International Conference on Informatics, Electronics and Vision, pp. 1–5 (2014). DOI: 10.1109/ICIEV.2014.6850729.
16. National Center of Biotechnology Information: Home / nucleotide / NCBI (2018)
17. Snustad, D., Simmons, M.: Principles of Genetics. Wiley (2015)
18. Tan, P.N., Steinbach, M., Kumar, V.: Introduction to Data Mining. Pearson Addison-Wesley (2005)
19. Tastan-Bishop, O.: Bioinformatics and Data Analysis in Microbiology. Caister Academic Press (2014)
20. Tozammel-Hossain, K.S., Patnaik, D., Laxman, S., Jain, P., Bailey-Kellogg, C., Ramakrishnan, N.: Improved Multiple Sequence Alignments Using Coupled Pattern Mining. IEEE/ACM Transactions on Computational Biology and Bioinformatics, vol. 10, no. 5, pp. 1098–1112 (2013). DOI: 10.1109/TCBB.2013.36.
21. Wang, Q., Davis, D.N., Ren, J.: Mining Frequent Biological Sequences Based on Bitmap Without Candidate Sequence Generation. Computers in Biology and Medicine, vol. 69, pp. 152–157 (2016). DOI: 10.1016/j.compbiomed.2015.12.016.
22. Zhang, J., Wang, Y., Zhang, C., Shi, Y.: Mining Contiguous Sequential Generators in Biological Sequences. IEEE/ACM Transactions on Computational Biology and Bioinformatics, vol. 13, no. 5, pp. 855–867 (2016). DOI: 10.1109/TCBB.2015.2495132.