A Novel Suffix Based Pattern Mining On Sequential Datasets

Naga Laxmi Pramoda.G[*], Kumar Vasantha [#],Prof. C.Mohan rao [$]
[*] *M.Tech Scholar,* [#] *HOD,* [$] *Professor and Principal*
[*#$]*Dept of CSE, Avanthi Institute of Engineering and Technology*

Abstract: Pattern mining is the hard task in searching and matching process. It plays key role in the search engines. User wants their search results have to accurate and moderate. So that the search or matching operation must have to find all patterns with respect to the given pattern by the user. So we implemented a new method for pattern mining or matching. It generates all possible patterns of the given pattern. By this user definitely gets efficient pattern matching results. This method reduce more processing time and the computational complexity.

I.INTRODUCTION

Sequence mining is a topic of data mining concerned with finding statistically relevant patterns between data examples where the values are delivered in a sequence.[1] It is usually presumed that the values are discrete, and thus time series mining is closely related, but usually considered a different activity. Sequence mining is a special case of structured data mining. There are several key traditional computational problems addressed within this field. These include building efficient databases and indexes for sequence information, extracting the frequently occurring patterns, comparing sequences for similarity, and recovering missing sequence members. In general, sequence mining problems can be classified as string mining which is typically based on string processing algorithms and itemset mining which is typically based on association rule learning[1][2].

In a number of sequential data mining applications, the goal is to discover frequently occurring patterns. The challenge in discovering such patterns is to allow for some noise in the matching process. At the heart of such a method is the definition of a pattern, and the definition of similarity between two patterns.[3][4][5] This definition of similarity can vary from one application to another. This approximate subsequence mining problem is of particular importance in computational biology, where the challenge is to detect short sequences, usually of length 6- 15, that occur frequently in a given set of DNA or protein sequences. These short sequences can provide clues regarding the locations of so called "regulatory regions," which are important repeated patterns along the biological sequence. The repeated occurrences of these short sequences are not always identical, and some copies of these sequences may differ from others in a few positions [7][8].

Database mining is motivated by the decision support problem faced by most large retail organizations. Progress in bar-code technology has made it possible for retail organizations to collect and store massive amounts of sales data, referred to as the basket data. A record in such data typically consists of the transaction date and the items bought in the transaction. Very often, data records also contain customer-id, particularly when the purchase has been made using a credit card or a frequent-buyer card. Catalog companies also collect such data using the orders they receive. The problem of finding frequently occurring (non-contiguous) sub-sequences in large sequence databases has been extensively studied in previous works. Traditionally, B is called a subsequence of A, if B can be constructed by projecting out some of the elements of sequence A. For instance, if A is the sequence "a, b, a, c, b, a, c," the sequence "a, b, b, c" is a subsequence constructed by choosing the first, second, fifth, and seventh elements from the original sequence and omitting the rest. While mining for frequent non-contiguous sub-sequences has many uses, it is not appropriate for many applications such as DNA and protein motif mining. A subsequence constructed by gluing together distant parts of the original sequence is not meaningful in many applications. In mining for motifs, we are interested in contiguous sub-sequences. Furthermore, previous work on non-contiguous subsequence models cannot easily incorporate noise tolerance in the way that contiguous motif models can. In short, subsequence mining and motif mining are different data mining operations, and there are distinct applications of each of these. This paper focuses on the contiguous subsequence (motif) mining problem.[9][10].

II.RELATED WORK

A) Periodic Patterns

This model of pattern is quite rigid and it fails to find patterns whose occurrences are asynchronous. Periodicity detection on time series database is an important data mining task and has broad applications. For example, "The gold price increases every weekend" is a periodic pattern. As mentioned above, this model is often too restrictive since we may fail to detect some interesting pattern if some of its occurrences are misaligned due to inserted noise events[11]. A pattern can be partially filled to enable a more flexible model. For instance, pattern length three (I_1,*,*) is a partial pattern showing that the first symbol must be I_1. The system behaviour may change over time and some patterns may not be present all the time. Two parameters, namely min-rep and max-dist, are used to

specify the minimum number of occurrences that is required within each sub-sequences and the maximum disturbance between any two successive sub-sequences. The rationale behind this is that a pattern needs to repeat itself at least a certain number of times to demonstrate its significance and periodicity. Couple of algorithms in this area focused on patterns for some pre-specified period length and several models can discover all periodic patterns regardless of the period length. Notice that period usually is a part of what we would like to mine from data. Let us look at an example to explain some definitions.

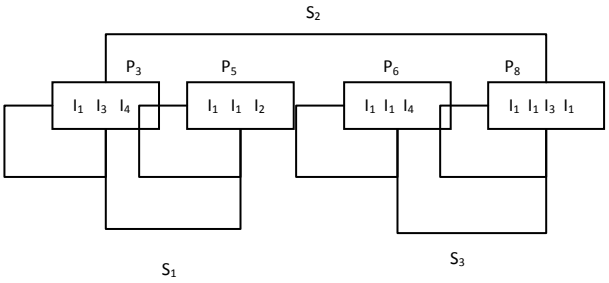

Figure 1 shows matches of partial pattern (I_1,*,*) that is a 1-pattern of period 3. In this Figure, P_1,P_2,…, and P_{10} are ten matches of pattern (I_1,*,*) and S_1,S_2, and S_3 are three segments which each one forms a list of N consecutive matches of (I_1,*,*). For example, segment S_2 is consist of 5 successive matches of this pattern. S_1 and S3 are disjoint segments and S_1 and S_2 are overlapped segments. In many applications, overlapped segments often are not considered.

If the value of min-rep is set to 3, then S_1 and S_2 qualify as valid segments and S_3 is not a valid segment. Two or more than two contiguous and disjoint segments can construct a valid subsequence provided that distance between any two successive valid segments does not exceed the parameter max-dist. For example, if the value of min-rep and max-dist are set to 2 and 3 respectively, both S_1 and S_3 recognized as valid subsequence whose overall number of repetition is 5. Given a sequence S, the parameters min-rep and max-dist[13][14].

and the maximum period length Lmax, in three phases we can discover the valid sub-sequences that have the most repetitions for each valid pattern whose period length does not exceed Lmax. When parameters are not set properly, noise may be qualified as a pattern. Though, parameter max-dist is employed to recognize the noises between segments of perfect repetition of a pattern. The following three phases outline algorithm for mining periodic patterns in brief. The first phase: For each symbol I, the distance between any two occurrences of I are examined and then for each period l, the set of symbols whose number of times are at least min-rep are sent to the next phase. Since there are a huge number of candidates, a pruning method is needed to reduce it. The second phase: In this phase, the single patterns (1- pattern) are generated. For each period l and each symbol I a candidate pattern (I,*,*,…,*) is formed that number of symbol * is (l-1). The third phase: After discovering the single patterns in previous phase, i-patterns are generated from the set of valid (i-1)-patterns and then these patterns are validated. In this phase, we can apply some heuristics. For example, it is obvious that if a pattern is valid, then all of its generalizations are valid. Pattern (I_1,I_2,*) is a generalization of pattern (I_1,I_2,I_3).[15]

B) Mining sequential patterns:

1. Sort Phase: The database (D) is sorted, with customer-id as the major key and transaction-time as the minor key. This step implicitly converts the original transaction database into a database of customer sequences.

2. 'l' itemset Phase: In this phase we find the set of all 'l' itemsets L. We are also simultaneously finding the set of all large 1-sequences, since this set is just {(l) | l € L}. The problem of finding large itemsets in a given set of customer transactions, albeit with a slightly different definition of support. In these papers, the support for an itemset has been defined as the fraction of transactions in which an itemset is present, whereas in the sequential pattern finding problem, the support is the fraction of customers who bought the itemset in any one of their possibly many transactions. The main difference is that the support count should be incremented only once per customer even if the customer buys the same set of items in two different transactions.

Cust Id	Transaction Time	Items Bought
1	Jan 25' 12	30
1	Jan 30' 12	90
2	Jan 10' 12	10,20
2	Jan 15' 12	30
2	Jan 20' 12	40,60,70
3	Jan 25' 12	30,50,70
4	Jan 23' 12	30
4	Jan 20' 12	40,70
4	Jan 25' 12	90
5	Jan 12' 12	90

3. Transformation Phase: We need to repeatedly determine which of a given set of large sequences are contained in a customer sequence. To make this test fast, we transform each customer sequence into an alternative representation. In a transformed customer sequence, each transaction is replaced by the set of all 'l' itemsets contained in that transaction. If a transaction does not contain any 'l' itemset, it is not retained in the transformed sequence. If a customer sequence does not contain any 'l' itemset, this sequence is dropped from the transformed database. However, it still contributes to the count of total number of customers. A customer se-

quence is now represented by a list of sets of 'l' itemsets. Each set of 'l' itemsets is represented by (l_1, l_2,. . . l_n), where l_i is a 'l' itemset.

This transformed database is called DT . Depending on the disk availability, we can physically create
this transformed database, or this transformation can be done on-the-y, as we read each customer sequence
during a pass. (In our experiments, we physically created the transformed database.)

For example, during the transformation of the customer sequence with Id $_2$, the trans-action (10 20) is dropped because it does not contain any 'l' itemset and the transaction (40 60 70) is replaced by the set of 'l' itemsets {(40), (70), (40 70)}.

4.The Sequence phase: The general structure of the algorithms for the sequence phase is that they make multiple passes over the data. In each pass, we start with a seed set of large sequences. We use the seed set for generating
new potentially large sequences, called candidate sequences. We find the support for these candidate sequences during the pass over the data. At the end of the pass, we determine which of the candidate sequences are actually large. These large candidates become the seed for the next pass. In the first pass, all 1-sequences with minimum support, obtained in the 'l' itemset phase, form the seed set.

.**Apriori Candidate Generation**

The apriori-generate function takes as argument L_{k-1}, the set of all large (k-1)-sequences. The function works as follows.
First, join L_{k-1} with L_{k-1}:
insert into C_k
select $p.litemset_1$, ..., $p.litemset_{k-1}$, $q.litemset_{k-1}$
from L_{k-1} p, L_{k-1} q
Next, delete all sequences c_2 belongs to C_k such that some (k-1)-subsequence of c is not in L_{k-1}.
If this is given as input to the apriori-generate function, we will get the sequences shown in the second column after the join. After pruning out sequences whose sub-sequences are not in L3, the sequences shown in the third column will be left.

Both the count-some algorithms have a forward phase, in which we find all large sequences of certain lengths, followed by a backward phase, where we find all remaining large sequences. The essential difference is in the procedure they use for generating candidate sequences during the forward phase. As we will see momentarily, Apriori-Some generates candidates for a pass using only the large sequences found in the previous pass and then makes a pass over the data to find their support. Dynamic-Some generates candidates on- the-y using the large sequences found in the previous passes and the customer sequences read from the L_1 = {large 1-sequences};
// Result of litemset phase
for (k = 2; $L_{k-1} \neq \phi$;; k++) do
begin
C_k = New candidates generated from L_{k-1}
foreach customer-sequence c in the database do
Increment the count of all candidates in C_k that are contained in c.
L_k = Candidates in C_k with minimum support.
end
Answer = Maximal Sequences in S_k L_k;
Notation In all the algorithms, L_k denotes the set of all large k-sequences, and Ck the set of candidate k-sequences.

III.PROPOSED WORK

We call our motif model the (L,M, s, k) model after the four parameters that determine it. L is the length of the motif, M is a distance matrix that is used to compute the similarity between two strings, s is the maximum distance threshold within which two strings are considered similar, and finally, k is the minimum support required for a pattern to qualify as a motif. Given, L, d, and k, a naive algorithm is to consider all
possible strings of length L over the alphabet (the space of all models), and compute the support for each of them by scanning the data set. This algorithm is exponential and becomes infeasible with large L and d values. One might be tempted to improve this method by considering only those strings of length L that actually occur in the data set.

However, this approach might miss motifs as the model string might not actually occur in the data set even once. To illustrate this point, suppose that the string ABCDEF is the true motif. Assume that we are looking for a (6, 2, 3) pattern, and that the instances of this pattern in the data set are FFCDEF, ABFFEF, and ABCDAA. Each instance is at a distance of 2 from the model ABCDEF, but the distance between any two instances is 4. If we consider only instances from the data set (which need not contain ABCDEF), then we will not find the motif. First we construct suffix tree means it counts number of nodes, and that is not restricted to a particular string is called model suffix tree.

Then we can find the supports of the nodes in the data suffix tree. By this we can easily find that every node contains number of leaves and some nodes have same leaves. For example ABCDE and ABCDF , it contains same prefixes . Our algorithm does not constructs the suffix tree but if it needs that it will. To understand our strategy of pruning the model suffix tree, consider the following example: Assume that the data set consists of sequences over the alphabet {A,B,C,D,E}. The data set and the values of L, d, and k are specified as input. All the strings of length L starting with the symbol A form a subset of the model space. We call this the A partition. This partition corresponds to all the nodes in the model suffix tree under the sub tree corresponding to node A. This partition is further divided into sub partitions with prefix AA, AB, AC, AD, and AE. These partitions continue on for L levels, and at the last level, we have only one model string for each partition. Suppose that we start by considering the models

in partition A. Assuming no mismatches are allowed, if the support for A is less than k, then, clearly any model that starts with A cannot qualify as a valid motif since there will be fewer than k instances of it, and it will not have the minimum support. Consequently, we can safely toss away the entire space of models starting with the symbol A. This step essentially prunes away the subtree corresponding to A in the model suffix tree. After pruning A, we proceed to consider the B partition. An important step here is to compute the support for models starting with A. This value is simply the number of times A occurs in the data set, and this value can be quickly looked up from the data suffix tree.

FSBP (modelTree, dataTree, l, d, k)
model = model Tree.FirstNode()

While (model ≠ model Tree.lastmodel())

Evaluate_Support (model,data Tree)

If (is valid (model) print "Found model:" , model

Else If(model.support() < k)

Model Tree.prune At(model)

Model = Next Node (model,model Tree)

End While

End

Sub Evaluate_Support (model, data Tree)

new symbol = last symbol of model.String

old matches = model.Parent().Matches()

new matches = EmptyMatches()

If (model.Parent() == root)

new matches = Expand_Matches (root,new symbol,data Tree)

Else

ForEach match x in old matches

New matches = new matches U

Expand_Matches(x,newsymbol,data Tree)

End ForEach

Model.Set Matches (new Matches)

Return

Sub Expand_Matches (x,newsymbol,data Tree)

Let Y = Set of all single character expansions of x.String in data Tree

ForEach element b in Y

If b's last symbol ≠ new symbol

b.mismatches++

If b.mismatches > max_mismatches

Remove b from Y

End ForEach

Return Y

The list of Matches for the Model A

Node	Number of mismatches	Count
A	0	100
B	1	50
C	1	45
D	1	120
E	1	15
Support	-	330

When mismatches are allowed, computing the support of a (partial) model string is more complicated. Suppose that d = 1. When considering matches for models starting with A, we cannot rule out strings that start with B (or any other symbol), since a string starting with B could match a model starting with A by only differing in the first position. Now assume that the data suffix tree nodes at depth 1 labeled A, B, C, D, and E have counts of 100, 50, 45, 120,

and 15, respectively. The possible number of strings starting with B that could match a model starting with A is simply the count of node B, namely 50. In a similar fashion, the count value from other nodes at most d mismatches away is read, and a list of potential matches for A is constructed as shown in Table 2. The list contains the node in the data suffix tree, the number of mismatches corresponding to this node, and the count from that node. For instance, node A in the data suffix tree has a count of 100 and perfectly matches the model string (A)—we store this information in the list as (A, 0, 100). The total support for the partial model is now computed by summing up the individual counts. In the example for Table 2, this sum is 330. Those nodes where the number of mismatches with the model being considered is greater than d are pruned away and not included in the list of matches. The algorithm then proceeds to consider the next partial model—AA. Observe that the list of matches for any partial model can be constructed incrementally using the list of matches for that model's longest prefix. For instance, the list of matches for AC can be constructed using the list for A (Table 2). We take each string from the list, and extend it by one symbol. The first string A, for instance, can be extended by one symbol to AA, AB, . . . , AE. The string AC has 0 mismatches to itself, the remaining strings have 1 mismatch each. The support for each of these string can be quickly looked up in the data suffix tree. We locate the model suffix tree node corresponding to A (stored in the list of matches). This node points to its children: AA, AB, . . . , AE. The support for each of them is read from the suffix tree, and a new list of matches is constructed for AC to compute its support. Similarly, when B is extended to length 2, all strings except BC have more than one mismatch with the model string AC. Therefore, only BC is included in the match list. The remaining nodes (C, D, and E) are expanded similarly.

Sub Expand_Matches_lMsk (x,newsymbol, data tree)

Let Y=Set of all single charcter expansions of x.string in data tree

For Each element b in Y

b.distance += Distance_Matrix(b.lastsymbol,newsymbol)

If b.distance > max_distance

Remove b from Y

End forEach

We take advantage of this method for incrementally computing the support by traversing the model suffix tree in the depth-first order. If L = 3, the partitions will be considered in the order A, AA, AAA, AAB, AAC, etc. At each node, the match list and the support for the parent node has already been computed, and can be used to compute the support of the current node. Observe that if we want to distinguish between multiple matches within a single sequence or matches within different sequences, we can simply replace the count in each node of the data suffix tree with the count of sequence separator node in its sub-tree. That is, while building the suffix tree, we simply store the number of distinct sequences the patterns occur in instead of the total count. This allows FLAME to easily support both models.

The algorithm simply puts together the ideas described above. It starts by traversing the nodes of the model space in depth first order. At each node in the model suffix tree, the subroutine Evaluate_Support is called to compute the list of matches and the new support. This routine uses the match list from the parent node to speed up the computation. The routine Expand_Matches ensures that the number of mismatches to the model string does not exceed d. At any node, if FLAME discovers that the support is lower than k, it prunes away that subtree in the model suffix tree, and continues its traversal. If it finds a model of length L with the required support, it simply outputs the result. Instead of merely keeping track of the number of mismatches, they keep track of the substitution distance score. That is, for each node, the match list stores

$$\sum_{i=1}^{n} M(x_i, y_i)$$

where x_i is the symbol from the prefix of the i partition, and y_i is the symbol it is being matched to in the data set. If this distance score exceeds the preset threshold (s), we prune the model suffix tree at that point, and continue the depth-first traversal just as in the case of the simpler (L,d, k) model.

CONCLUSION

In paper we introduced a suffix based pattern mining algorithm, it works on different motifs and it also works on synthetic datasets. It find more accurate patterns when the user search. It constructs data tree based on the patterns available in the given data. So that finding a pattern is easier in small amount of time and accurate. It extract more matches till the mismatch occurs in the given data.

REFERENCES

[1] M.O. Dayhoff, R.M. Schwartz, and B. Orcutt, "A odel forEvolutionary Changes in Proteins," Atlas of Protein equence and Structure, vol. 5, pp. 345-352, Nat'l Biomedical Research Foundation,1978.

[2] S. Henikoff and J. Henikoff, "Amino Acid Substitution Matrices from Protein Blocks," Proc. Nat'l Academy of Sciences USA, vol. 89, no. 22, pp. 10915-10919, 1992.

[3] R. Agrawal and R. Srikant, "Fast Algorithms for Mining Association Rules," Proc. Int'l Conf. Very Large Data Bases (VLDB), pp. 487-499, 1994.

[4] R. Agrawal and R. Srikant, "Mining Sequential Patterns," Proc. 11th IEEE Int'l Conf. Data Eng. (ICDE), pp. 3-14, 1995.
[5] M.J. Zaki, "SPADE: An Efficient Algorithm for Mining Frequent Sequences," Machine Learning, vol. 42, nos. 1/2, pp. 31-60, 2001.
[6] J. Wang and J. Han, "BIDE: Efficient Mining of Frequent Closed Sequences," Proc. 20th IEEE Int'l Conf. Data Eng. (ICDE), pp. 79-90, 2004.
[7] X. Yan, J. Han, and R. Afshar, "CloSpan: Mining Closed Sequential Patterns in Large Datasets," Proc. SIAM Int'l Conf. Data Mining (SDM), 2003.
[8] J. Yang, W. Wang, P.S. Yu, and J. Han, "Mining Long Sequential Patterns in a Noisy Environment," Proc. ACM SIGMOD, pp. 406- 417, 2002.
[9] S. Sinha and M. Tompa, "YMF: A Program for Discovery of Novel Transcription Factor Binding Sites by Statistical Overrepresentation," Nucleic Acids Research, vol. 31, no. 13, pp. 3586-3588, 2003.
[10] G. Pavesi, P. Mereghetti, G. Mauri, and G. Pesole, "Weeder Web: Discovery of Transcription Factor Binding Sites in a Set of Sequences From Co-Regulated Genes," Nucleic Acids Research, vol. 32, pp. W199-W203, 2004.
[11] E. Eskin and P.A. Pevzner, "Finding Composite Regulatory Patterns in DNA Sequences," Proc. 10th Int'l Conf. Intelligent Systems for Molecular Biology (ISMB), pp. S354-S363, 2002.
[12] J. Buhler and M. Tompa, "Finding Motifs Using Random Projections," J. Computational Biology, vol. 9, no. 2, pp. 225-242, 2002.
[13] G. Das, K.-I. Lin, H. Mannila, G. Renganathan, and P. Smyth, "Rule Discovery from Time Series," Proc. Int'l Conf. Knowledge Discovery and Data Mining (KDD), pp. 16-22, 1998.
[14] S. Hoppner, "Discovery of Temporal Patterns—Learning Rules about the Qualitative Behaviour of Time Series," Proc. Fifth European Conf. Principles and Practice of Knowledge Discovery in Databases, pp. 192-203, 2001.
[15] P. Patel, E. Keogh, J. Lin, and S. Lonardi, "Mining Motifs in Massive Time Series Databases," Proc. IEEE Int'l Conf. Data Mining (ICDM), pp. 370-377, 2002.

BIOGRAPHIES

Author 1:

Naga Laxmi Pramoda .G pursuing M.Tech 2 nd year Computer Science and Engineering interested in Networking and DBMS

Author 2:

Mr. kumar vasantha He has obtained M.Tech in Computer Science and Technology from jawaharlal nehru technological university Kakinada, He has published 10 papers in National and international journals his interested areas are RDBMS,Web Technologies

Author 3:

Dr. C. Mohan Rao, He has obtained M.Tech in Computer Science and Technology from Andhra University College of Engineering and awarded Ph.D by Andhra University during 2000. He has 18 years of teaching and research experience and guided number of M.Tech students for their projects. He has published 23 papers in National and international journals. He is guiding 2 research scholars for Ph.D. He received Best Teacher award from JNTU, Kakinada during 2009.