A Fast Algorithm to Find All the Maximal Frequent Sequences in a Text

René A. García-Hernández, José Fco. Martínez-Trinidad, and Jesús Ariel Carrasco-Ochoa

National Institute of Astrophysics, Optics and Electronics (INAOE)
Puebla, México
{renearnulfo,fmartine,ariel}@inaoep.mx

Abstract. One of the sequential pattern mining problems is to find the maximal frequent sequences in a database with a β support. In this paper, we propose a new algorithm to find all the maximal frequent sequences in a text instead of a database. Our algorithm in comparison with the typical sequential pattern mining algorithms avoids the joining, pruning and text scanning steps. Some experiments have shown that it is possible to get all the maximal frequent sequences in a few seconds for medium texts.

1 Introduction

The *Knowledge Discovery in Databases* (KDD) is defined by Fayyad [1] as "the nontrivial process of identifying valid, novel, potentially useful and ultimately understandable patterns in data". The key step in the knowledge discovery process is the data mining step, which following Fayyad: "consisting on applying data analysis and discovery algorithms that, under acceptable computational efficiency limitations, produce a particular enumeration of patterns over the data". This definition has been extended to *Text Mining* (TM) like: "consisting on applying text analysis and discovery algorithms that, under acceptable computational efficiency limitations, produce a particular enumeration of patterns over the text". So TM is the process that deals with the extraction of patterns from textual data. This definition is used by Feldman [2] to define *Knowledge Discovery in Texts* (KDT). In both KDD and KDT tasks, especial attention is required in the performance of the algorithms because they are applied on a large amount of information. In especial the KDT process needs to define simple structures that can be extracted from texts automatically and in a reasonable time. These structures must be rich enough to allow interesting KD operations [2].

The frequent sequences are of interest in some areas, such as data compression, human genome analysis and in the KDD process. But some of these areas are more interested in the maximal frequent sequences (MFS) because these areas search the longest pattern that could match or that could be extracted from the database. The *sequential pattern mining problem* is defined by Agrawal [3] as the problem of finding MFS in a database; this is a data mining problem. Therefore, we are interested in finding all the MFS in a text, in order to do text mining for the KDT process.

MFS have received special attention in TM because this kind of patterns can be extracted from text independently of the language. Also they are human readable patterns or descriptors of the text. MFS can be used to predict or to determine the causality of an event. For information retrieval systems, MFS can be used to find keywords; in this case, the MFS are key phrases. MFS allow constructing links between docu-

A. Sanfeliu et al. (Eds.): CIARP 2004, LNCS 3287, pp. 478–486, 2004.

ments in an automatic way. MFS could help in the definition of stop-phrases instead of stop-words. Also, they can be used for text summarization.

In this paper, we propose a fast algorithm which gets all the MFS from a simple structure. This structure is relatively easy to extract from the text.

The paper organization is as follows: In section 2, the problem definition is given. In section 3 the related works are presented. Section 4 describes our algorithm and in section 5 a complexity analysis of it is presented. Section 6 presents some experiments. Section 7 gives a discussion about our algorithm. Finally, in section 8 the conclusions and future work are given.

2 Problem Definition

A s*equence S* is an ordered list of at least two elements called *items*. The ith element or item in the sequence is represented as s_i. A sequence is *frequent* if it appears in the text twice or more. And *S* is *maximal* if *S* is not a subsequence of any other frequent sequence. The number of elements in a sequence *S* is called the *length* of the sequence and is denoted by $|S|$. A sequence with length *k* is denoted as *k-sequence*.

Let $P=p_1p_2\ldots p_n$ and $S=s_1s_2\ldots s_m$ be sequences, *P* is contained or is a subsequence of *S,* denoted $P\subseteq S$, if there exists an integer $1\leq i$ such that $p_1=s_i$, $p_2=s_{i+1,}$ $p_3=s_{i+2,}\ldots,p_n=s_{i+(n-1)}$. *P* is a *proper subsequence* of *S*, $P\subset S$, if *P* is a subsequence of *S*, but $P\neq S$.

Let $X\subseteq S$ and $Y\subseteq S$ then *X* and *Y* are *mutually excluded* if *X* and *Y* do not share items i.e., if $(x_n=s_i$ and $y_1=s_j)$ or $(y_n=s_i$ and $x_1=s_j)$ then $i<j$.

Given a text *T* expressed as a sequence and a user-specified threshold β. A sequence *S* is *β-frequent* in *T*, if it is contained at least β times in *T* in a mutually excluded way.

In this paper, we are interested in the problem of finding all the maximal *β-frequent* sequences in a text.

3 Related Work

Most of the algorithms that get all MFS [3,4,5,6] have been developed to work in a database of sequences. In [7], an algorithm was developed for a text collection, which is different from finding all the MFS into a single text. The algorithms for getting all MFS can be classified as Apriori-based (typical) and Pattern-growth methods [8].

Into the typical methods the first approach was the AprioriAll algorithm [3] and some approaches, like GSP [9], were after developed. These typical algorithms operate in a bottom-up breadth-first way. These methods use the joining, pruning and text scanning steps, for the candidate generation phase. In the fig.1a the general algorithm for the Apriori methods is shown. The fig. 1b shows an example of the different *k-sequences* that are generated for this kind of algorithms. The subindex in the sequences of fig. 1b indicates the frequency of each sequence.

Unlike the typical methods, the pattern-growth methods avoid the joining, pruning and scanning steps. These pattern-growth methods try to find the MFS more directly, expanding the growth of the *k-sequences*, beginning with *2-sequences*. For this reason, these methods are faster than the typical methods, when there are long sequences. Examples of this category are the PrefixSpan [5] and SPADE [6] algorithms

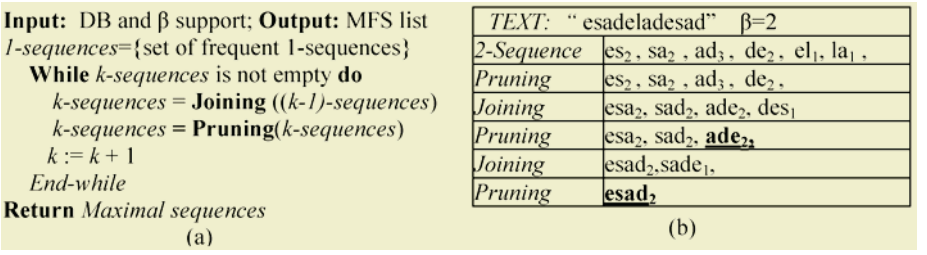

Fig. 1. a) General algorithm for the typical methods; b) Example of how to get the MFS with the joining and pruning steps for the text "esadeladesad"

4 Proposed Algorithm

Our algorithm belongs to pattern-growth methods class; because it uses a bottom-up strategy without candidate generation. The main idea consists in to generate only all the distinct pairs of items from the text, i.e. the *2-sequences*, and do not lose the relation between them, in order to allow the growth of the sequences. The input data of the algorithm are a text (*T*) and a β threshold. The proposed algorithm has three phases. These phases are as follows:

Phase 1. - Get the alphabet from the text. The algorithm gets an *id* for each different item (chars or words) from the text.

Phase 2: Algorithm to construct the array structure
Input: A text *T* **Output:** The *array* structure
For *all the pairs* $[t_i,t_{i+1}] \in T$ **do**
PositionNode.Pos ←*index* ← *array* $[t_i,t_{i+1}]$; // *if* $[t_i,t_{i+1}]$ *it is not in array, add it*
array[index].Positions ←**New** *PositionNode* // *new node*
array[index].Freq ← *array[index].Freq*+ 1 // *increase the frequency*
array[LastIndex].Positions.NextIndex←*index;* // *keep the index of the pair*
array[LastIndex].Positions.NextPos←*PositionNode;*//*link the new node*
LastIndex ← *index*; //*keep the last index*
End-for

Fig. 2. Algorithm to construct the *array* structure (phase 2)

Phase 2. - Construct an array structure for text representation (fig. 2). The algorithm constructs an *array* structure from the text *T*. Each element of the *array* contains two *id*'s corresponding to a distinct pair (t_i,t_{i+1}), the *frequency* of this pair and a list of the *positions* where this pair appears in the text (see fig. 3a). Each *position* node of the *positions* list contains the position where the pair appears, together with the *next-index* of the following pair in each position (see fig. 3b). The phase 2 works as follows: for each item t_i get the *index* of the pair (t_i,t_{i+1}) in the *array* and in this position add a *Position* node at the end of the list of *positions*. Increase the frequency (*Freq*) of this pair and link this *position* node with the previously added *position* node in order to build the *NextPos* list, which stores the text representation. In fig. 3a an example of this array is shown.

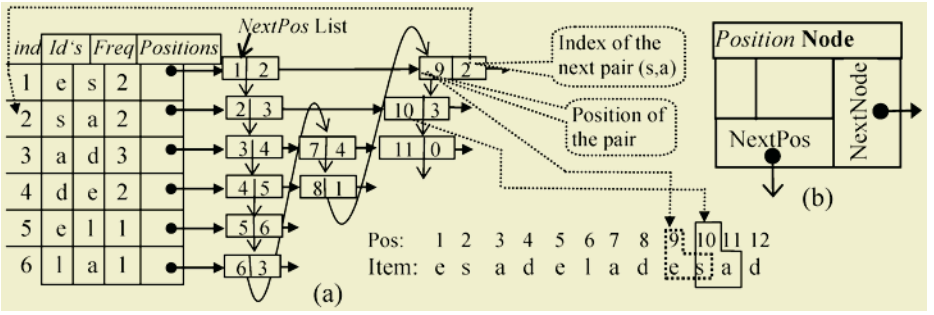

Fig. 3. a) *Array* for the text: "esadeladesad". b) Node for *Positions* list

Phase 3: Algorithm to find all MFS
Input: *Array* from phase 2, β support **Output:** MFS list
Actual←1 *//index of the array where it is the first element of NextPos List*
while *Actual* ≠ 0 **do**
if *Array[Actual].Frequency* ≥ β, **then** *//if the pair has frequency ≥ β*
temporal ← Copy_list (*array[Actual].Positions) //create a similar list*
PMS ← *Array[Actual].Id*$_1$ + *Array [Actual].Id*$_2$ *//initial elements of the PMS*
Pos ← *Array [Actual].NextIndex* *//The first time Pos←1*
while *Pos* ≠ 0 **do** *//expand the pair*
temporal ← **Get common nodes((***temporal.Pos+1) , (Array[Pos].Positions.Pos))*
if | *temporal* | ≥ β, **then** *//if temporal has a number of nodes ≥ than β*
if *Pos* = *Array*, **then** there is a cycle,
PMS ← **Cycle(** β, *temporal*, *array*, *Actual*, *Pos* **)** *//call to Cycle function*
if the *PMS* cannot grow **then** exit from the while
else *PMS* ← *PMS* + *Array[Pos].Id*$_2$ *// expand the PMS*
Pos ← *Array[Actual].NextIndex*
end-while
delete all the MFS ⊆ *PMS*
if (*PMS* ⊄ MFS) **then** MFS←Add(*PMS*)
Actual ← *Array[Actual].NextIndex*
End-while

Fig. 4. Algorithm to find all MFS (phase 3)

Phase 3. – Find all MFS (fig. 4). For each element *i* of the *NextPos list,* check if it has a frequency ≥ β, in order to determine if this pair can become a possible maximal sequence PMS. If frequency ≥ β then grow forward all the elements in the *NextPos* list w.r.t *i*. If after this growth there is (in the *NextPost* list) a number of elements ≥ β, then the PMS can grow. When the PMS cannot grow it is added to the MFS list if only if the PMS is not a subsequence of any previously stored MFS, and all the MFS that are subsequence of the PMS are deleted from the MFS list. The table 1 shows how the algorithm finds the first four MFS for the example of the fig. 3a.

In the example of figure 3a, the generated PMS does not contain cycles since all its pairs are different. For our algorithm each PMS can be classified as a PMS with and without cycles. A *cycle* is detected when the first pair is repeated if it happens, the *cycle* function is used to get the PMS. The cycle function guarantees the *mutually excluded* property. If the PMS obtained from c*ycle* function can grow, then it is treated as a PMS without cycles, because it could grow.

Table 1. Example of how to get a PMS for the text "esadeladesad" with the structure presented in figure 2.a, using the algorithm of figure 4 with β=2. Since the first pair "es" has two ocurrences in the text (frequency ≥ β) in the positions 1 and 9 (*temporal* list), this pair becomes the new PMS. Then, the next item is "a" for both ocurrences. Therefore, the PMS can grow in one (PMS="esa"), it is, the positions are increased in one (*temporal={2,10}*). The same happends w.r.t. the "d" item and the PMS="esad". Since the next item has frequency < β the PMS cannot grow and it is store. Then, this process is repeated to get more PMS

Actual	Pos	temporal	PMS_Freq	Action	PMS
1	**1**	1,9	2	PMS_Freq ≥ β, PMS=Pair=T_{pos} + T_{pos+1}	es
2	2	2,10	2	PMS_Freq ≥ β, Grow PMS= PMS + T_{pos+1}	esa
3	3	3,11	2	PMS_Freq ≥ β, Grow PMS= PMS + T_{pos+1}	**esad**
4	4	4	1	PMS_Freq is not ≥ β, Store the PMS	
2	**2**	2,10	2	PMS_Freq ≥ β, PMS=Pair=T_{pos} + T_{pos+1}	sa
3	3	3,11	2	PMS_Freq ≥ β, Grow PMS= PMS + T_{pos+1}	**sad**
4	4	4	1	PMS_Freq ≥ β, Store the PMS	

Cycle function: Algorithm to find the PMS with cycles
Input: β support, *temporal*, *array*, *Actual*, *aux*; **Output:** A PMS
CycleSize ← *Array[aux].Pos-Array[Actual].Pos*; //size of the cycle
Intervals ← From *temporal* find the intervals of groups of cycles
ActualGrpSize ← Size of the interval where [*Array[actual].Pos-Array[aux].Pos*] ∈ *Intervals*
While *ActualGrpSize* ≥ 2 **do**
  For each *Interval* get the *frequency* w.r.t. *ActualGrpSize*
    if ∑ *frequencies* ≥ β **then** PMS←T[*Array[actual].Pos*] +…+T[*Array[actual].Pos+ActualGrpSize*]
      if *ActualGrpSize* was not decremented **then** the PMS can grow
        temporal ← Rebuild *temporal* with (*end of Intervals* -1) in which the frequency = 1
      return (PMS, *temporal*);
  end-for
end-while
The *frequency* is calculated as follows:
    Used-Periods ← Ceiling (*ActualGrpSize* / *CycleSize*);
    Period ← *GrpSize analyzed* / *CycleSize*
  if *Used-Periods.*$_{remainder}$ = *Period.*$_{remainder}$ and *Period.*$_{remainder}$ >0 **then**
    Period←*Period* +1
Frequency = *Period* / *Used-Periods*

Fig. 5. Algorithm for finding a PMS with cycles

Cycle function (fig. 5). Using the *size of the cycle* (number of elements between the first and the repeated pair) find all the groups of occurrences of the cycle in order to build a list of *intervals* with the beginning and end of such positions. Using this list of intervals it is possible to find the longest PMS. Given the *size of the interval,* this function tests in decreasing way (because we search the longest PMS) how many PMS are contained in each interval, therefore the sum of this local frequency becomes the total frequency that must be ≥ β. In such case, the PMS has as size the *size of the interval* that can appear β times into the text. If the *size of the interval* was not decremented then it is a PMS that can grow. The table 2 shows an example of how to find a PMS with cycles.

Table 2. Example of the PMS obtained from the *cycle* function for the text "abcabcabMabc GabcabcabMabc" with β =2. When the algorithm of phase 3 detects that the initial pair "ab" is repeated in the PMS="abcab", then the *cycle* function is activated. In step 1, all the groups of occurrences of cycles are found. In the Step 2, the *CycleSize* and *ActualGrpSize* are caculated. In the Step 3, the frequency of apparition is computing for each group using *CycleSize* and *ActualGrpSize*. In the Step 4, if the total frequency ≥ β, then the length of the PMS is *ActualGrpSize* and the PMS starts in the originial inital position, but if total frecuency < β, then decrement *ActualGrpSize* and return to step 3. In this example, the PMS obtained from *Cycle function* is "abcabcab". But, if *ActualGrpSize* is not decremented, then the PMS obtained is treated as a PMS without clycles and the phase 3 gets the PMS="abcabcabMabc"

Text:	a	b	c	a	b	c	a	b	M	a	b	c	G	a	b	c	a	b	c	a	b	M	a	b	c
Position	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25

Given the above text, the following steps are generated (similar process of the Table 1)

Actual	*Pos*	*Temporal*	*PMS-Freq*	*Action*	*PMS*
1	**1**	1,4,7,10,14,17,20,23	8	PMS-Freq ≥ β, PMS=Pair=T_{pos} + T_{pos+1}	ab
2	2	2,5,11,15,18,24	6	PMS-Freq ≥ β, Grow PMS= PMS + T_{pos+1}	abc
3	3	3,6,16,19	4	PMS-Freq ≥ β, Grow PMS= PMS + T_{pos+1}	abca
1	**4**	4,7,17,20	4	PMS-Freq ≥ β, Grow PMS= PMS + T_{pos+1}	**abcab**

Step 1: The *cycle* function is activated since the first value of *Actual* is repeated:

From *temporal* list it is possible to get the *intervals* of positions of the groups of occurrences of the cycle:{4 , 7 , 17 , 20 } =*temporal*

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
a	b	c	a	b	c	a	b	M	a	b	c	G	a	b	c	a	b	c	a	b	M	a	b	c

[1 , 8] ⟵ Groups ⟶ [14 , 21]

Step 2: The *CycleSize* and *ActualGrpSize* values are needed to calculate the frequency of each interval:

The *CycleSize*=3:(size of the repeated string)

1	2	3	4	5	6	7	…
a	b	c	a	b	c	a	…

ActualGrpSize=8 (size of the interval in which is contained the actual positions that are analized, that is [1,4] ∈ [1,8], therefore *ActualGrpSize*=(8-1)+1=8)

Step 3: Using the algorithm of fig. 5 the frequency for each interval is calculated:
For [1-8] the frequency = 1; and for [14-21] the frequency = 1

Step 4: Since (∑ frequencies) ≥ β , then the |PMS|= *ActualGrpSize* and the PMS=$T_{(m-1)+i}$ where $1 \le i \le$ *ActualGrpSize* and *m* is the first value of *Pos*, in this case the PMS="abcabcab". Since *ActualGrpSize* was not decremented, the PMS can growth and must be treated as a PMS without cycles. The process continue so on, finally the PMS obtained is "abcabcabMabc"

5 Complexity Analysis for the Proposed Algorithm

Let T a Text and $n=|T|$, then the required space in the worst case is $O(n)$ because no additional space is needed. Only the space for the array structure is used. As example, if n =100,000 chars (approx. 50 pages), the required space is approx. 3 MB. The time complexity to get the MFS is $O(kn)$ where k is the length of longest the PMS, in the worst case $k=n/2$ in such case the complexity is $O(n^2)$, but in the practice the PMS's are not very long, for example in [7] the longest PMS has a length of 22.

6 Experiments

From the collection given in [10] we chose the text "Autobiography" by Thomas Jefferson corresponding to: 243,115 chars, 31,517 words (approx. 100 pages), 5,499 different words and 18,739 different pairs. Also, we chose the text "LETTERS" by Thomas Jefferson with around 1,812,428 chars and 241,735 words (approx. 800 pages). In both texts the stop words were not removed and only the numbers and punctuation symbols were omitted. In order to show the behavior of the processing time against the number of words in the text, we compute the MFS using our algorithm for the minimum threshold value, β =2. Each chart in fig. 6 corresponds to one text, processing different quantities of words. The figure 6a starts with 5,000 words and uses an increment of 5,000, in order to see how the processing time grows when the number of processed words is augmented in the same text. In the fig. 6b, an increment of 40,000 words was used in order to see how the processing time grows for a big text. Also, in both charts the time for phases (1 and 2) is shown.

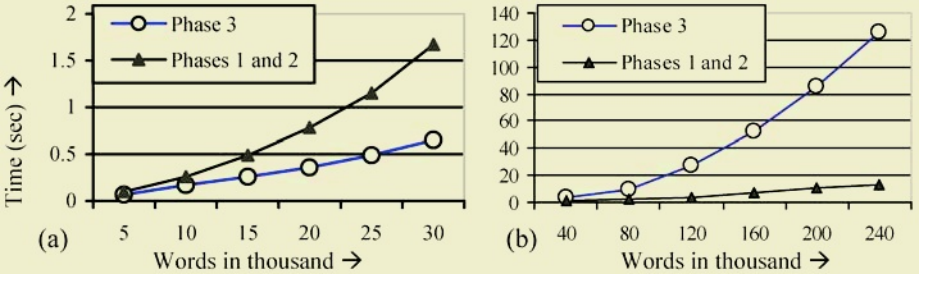

Fig. 6. Processing time for: a) "Autobiography" and, b) "LETTERS"

For the same documents the whole text was processed to find all the MFS, in order to appreciate (fig. 7) how the performance of our algorithm is affected for different values of the β threshold. Fig. 7a shows the time in seconds for "Autobiography" and fig. 7b for "LETTERS".

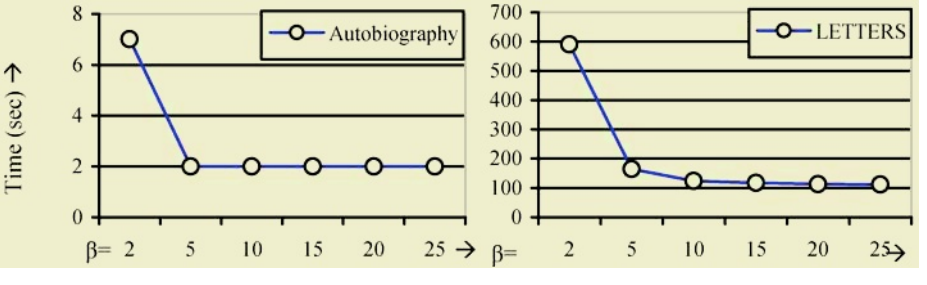

Fig. 7. Time performance for different values of β for: a) "Autobiography" and b) "LETTERS"

Furthermore, we have included in fig. 8 an analysis of the growth of the amount of MFS obtained from the same texts for different values for β.

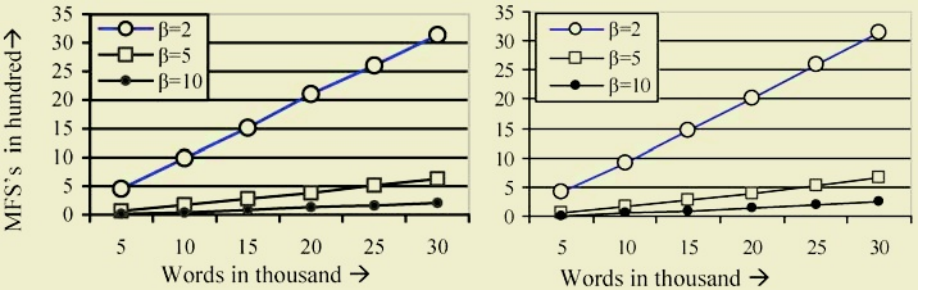

Fig. 8. Amount of MFS generated for different values of β for: a) "Autobiography" and, b) "LETTERS"

Additionally to these experiments, we processed the biggest text from the collection [10] "An Inquiry into the nature …" by Adam Smith with 2,266,784 chars corresponding to 306,156 words (approx. 1000 pages) with β=2, all MFS were obtained in 353 sec. approx. 5.88 min.

7 Discussion

The sequential pattern mining algorithms have been designed for working on a database. The comparison of our algorithm against this kind of algorithm is difficult since they have different assumptions and they were not designed to work on one text. First, with respect to the space the typical methods have a reasonably performance for 2-sequences, but the performance drastically decrease when any of the MFS becomes longer because if a l-sequence is a MFS, it implies the presence of 2^l-2 candidates (as example if l=100 then 2^{100}-1≈10^{30}), and each candidate must be explicitly examined [4,5]. The pattern-growth methods, for example PrefixSpan [5], need to get the frequent prefixes in order to keep all the projected databases, but it is very expensive. In contrast, our algorithm does not need to generate any extra auxiliary set of sequences or projected databases. Consequently, the required space for our algorithm is much smaller than pattern-growth methods. Moreover, our algorithm avoids the text scan to see if a sequence is frequent or it exists in the source. Our algorithm goes directly to find the PMS. Finally in comparison with others algorithms, our algorithm can run different values for β using the same structure; getting better times.

8 Conclusions

This paper proposes a fast algorithm to find all the MFS in one text, requiring only the β parameter. The algorithm uses a simple structure and does not need to use hash tables or any other auxiliary structure. In texts with cycles the algorithm uses the *cycle* function to become faster. As future work we are going to adapt this algorithm to work over a text collection, in order to allow a fair comparison against the other methods. Since our algorithm constructs an alphabet at phase 1, and the following phases work over this alphabet, it can be applied on any data represented as a sequence, for example DNA sequences in human genome analysis.

Acknowledgement

This work was financially supported by CONACyT (Mexico) through project J38707-A.

References

1. Fayyad, U., Piatetsky-Shapiro G. "Advances in Knowledge Discovery and Data mining". AAAI Press, 1996.
2. Feldman, R and Dagan, I. "Knowledge Discovery in Textual Databases (KDT)", *In Proceedings of the 1st International Conference on Knowledge Discovery (KDD-95)* 1995.
3. Agrawal, R and Srikant, R. "Mining Sequential Patterns" in *Proceedings of the International Conference on Data Engineering*, 1995.
4. Lin, Dao-I. Fast Algorithms for Discovering the Maximum Frequent Set, Ph. Thesis, New York University, 1998.
5. Pei, J, Han, et all: "PrefixSpan: Mining Sequential Patterns Efficiently by Prefix-Projected Pattern Growth" in *Proc International Conference on Data Engineering* (ICDE 01), 2001.
6. Mohammend j. Zaki, SPADE: An Efficient Algorithm for Mining Frequent Sequences, Machine Learning, Kluwer Academic Publishers, 2000.
7. Ahonen, H. "Finding All Maximal Frequent Sequences in Text". *ICML-99 Workshop: Machine Learning in Text Data*, 1999.
8. Antunes, C., Oliveira A. Generalization of Pattern-growth Methods for Sequential Pattern Mining with Gap Constraints. *Third IAPR Workshop on Machine Learning and Data Mining MLDM´2003*, 2003.
9. Srikant, R., and Agrawal, R. Mining sequential patterns: Generalizations and performance improvements. *In 5th Intl. Conf. Extending Database Discovery and Data Mining*, 1996.
10. Public domain documents from American and English literature as well as Western philosophy. http://www.infomotions.com/alex/