Protein Sequence Pattern Mining with Constraints

Pedro Gabriel Ferreira* Paulo J. Azevedo**

University of Minho,
Department of Informatics
Campus of Gualtar, 4710-057 Braga, Portugal
{pedrogabriel,pja}@di.uminho.pt

Abstract. Considering the characteristics of biological sequence databases, which typically have a small alphabet, a very long length and a relative small size (several hundreds of sequences), we propose a new sequence mining algorithm (*gIL*). *gIL* was developed for linear sequence pattern mining and results from the combination of some of the most efficient techniques used in sequence and itemset mining. The algorithm exhibits a high adaptability, yielding a smooth and direct introduction of various types of features into the mining process, namely the extraction of rigid and arbitrary gap patterns. Both breadth or a depth first traversal are possible. The experimental evaluation, in synthetic and real life protein databases, has shown that our algorithm has superior performance to state-of-the art algorithms. The use of constraints has also proved to be a very useful tool to specify user interesting patterns.

1 Introduction

In the development of sequence pattern mining algorithms, two communities can be considered: the *Data Mining* and the *Bioinformatics* community. The algorithms from the Data Mining community inherited some characteristics from the association rule mining algorithms. They are best suited for data with many (from hundred of thousands to millions) sequences with a relative small length (from 10 to 20), and an alphabet of thousands of events, e.g. [9, 7, 11, 1]. In the bioinformatics community, algorithms are developed in order to be very efficient when mining a small number of sequences (in the order of hundreds) with large lengths (few hundreds). The alphabet size is typically very small (ex: 4 for DNA and 20 for protein sequences). We emphasize the algorithm Teiresias [6] as a standard.

The major problem with Sequence pattern mining is that it usually generates too many patterns. When databases attain considerable size or when the average

* Supported by a PhD Scholarship (SFRH/BD/13462/2003) from Fundação Ciência e Tecnologia

** Supported by Fundação Ciência e Tecnologia - Programa de Financiamento Pluri-anual de Unidades de I & D, Centro de Ciências e Tecnologias da Computação - Universidade do Minho length of the sequences is very long, the mining process becomes computationally expensive or simply infeasible. This is often the case when we are mining biological data like proteins or DNA. Additionally, the user interpretation of the results turns out to be a very hard task since the interesting patterns are blurred into the huge amount of outputted patterns. The solution to this problem can be achieved through the definition of alternative interesting measures besides support, or with user imposed restrictions to the search space. When properly integrated in the mining process these restrictions reduce the computation demands in terms of time and memory, allowing to deal with datasets that are otherwise potentially untractable. These restrictions are expressed through what is typically called as *Constraints*. The use of Constraints enhances the database queries. The runtime reduction grants the user with the opportunity to interactively refine the query specification. This can be done until an expected answer is found.

2 Preliminaries

We consider the special case of linear sequences databases. A database D is as a collection of linear sequences. A *linear sequence* is a sequence composed by successive atomic elements, generically called events. Examples of this type of databases are protein or DNA sequences or website navigation paths. The term *linear* is used to make the distinction from the transactional sequences, that consist in sequences of *EventSets*(usually called as *ItemSets*). Given a sequence S, S' is subsequence of S if S' can be obtained by deleting some of the events in S. A sequence pattern is called a *frequent sequence pattern* if it is found to be subsequence of a number of sequences in the dataset greater or equal to a specified threshold value. This value is called *minimum support*, σ, and is defined as an user parameter. The *cover* represents the list of sequence identifiers where the pattern occurs. The cardinality of this list corresponds to the support of that pattern.

Considering patterns in the form $A_1 - x(p_1, q_1) - A_2 - x(p_2, q_2) - \ldots\ A_n$, a sequence pattern is an *arbitrary gap sequence pattern* when a variable (zero or more) number of gaps exist between adjacent events in the pattern, i.e. $p_i \leq q_i, \forall i$. Typically a variable gap with n minimum and m maximum number of gaps is described as $-x(n,m)-$. In the sequences $< 1\ 5\ 3\ 4\ 5 >$ and $< 1\ 2\ 2\ 3 >$ exists an arbitrary gap pattern $1{-}x(1,2){-}3$. A *rigid gap pattern* is a pattern where gaps contain a fixed size for all the database occurrences of the sequence pattern, i.e. $p_i = q_i, \forall i$. To denote a rigid gap the $-r(n)-$ notation is used, where n is the size of the gap. The $1{-}r(2){-}3$ is a pattern of length 4, in the sequences $< 1\ 2\ 5\ 3\ 4\ 5 >$ and $< 1\ 1\ 6\ 3 >$. Each gap position is denoted by the "." (wildcard) symbol, meaning that it matches any symbol of the alphabet. A pattern belongs to one of three classes: *maximal*, *closed* or *all*. A sequence pattern is *maximal* if it is not contained in any other pattern, and *closed* when all its extensions have an inferior support than itself. The *all* refers to when all the patterns are enumerated. When extending a sequence pattern $S =< s_1\ s_2 \ldots s_n >$, with a new event s_{n+1}, then

S is called a *base sequence* and $S' =< s_1\ s_2\ \ldots s_n\ s_{n+1} >$ the *extended sequence.* If an event b occurs after a in a certain sequence, we denoted it as: $a \rightarrow b$, and a is called the *predecessor*, $pred(a \rightarrow b) = a$, and b the *successor*, $succ(a \rightarrow b) = b$. The pair is frequent if it occurs in at least σ sequences of the database.

Constraints represent an efficient way to prune the search space [9, 10]. Considering the user's point of view, it also enables to focus the search on more interesting sequence patterns. The most common and generic types of constraints are:

– *Item Constraints*: restricts the set of the events (*excludedEventsSet*) that may appear in the sequence patterns,
– *Gap Constraints*: defines the (*minGap*) minimum distance or the maximum distance (*maxGap*) that may occur between two adjacent events in the sequence patterns,
– *Duration or Window Constraints*: defines the maximum distance (*window*) between the first and the last event of the sequence patterns.
– *Start Events Constraints*: determines that the extracted patterns start with the specified events (*startEvents*).

Another useful feature in sequence mining, in particular to protein pattern mining, is the use of *Equivalent/Substitution Sets.* When used during the mining process an event can be substituted by another event belonging to the same set. A *"is-a"* hierarchy of relations can be represented through substitution sets.

Depending on the target application of the frequent sequence patterns other measures of interest and scoring can be applied as posterior step of the mining process. Since the closed and the maximal patterns are not necessarily the most interesting we designed our algorithm in order to find all the frequent patterns. From the biological point of view, rigid patterns allow to find more well conserved regions, while arbitrary patterns permit the cover of a large number of sequences in the database.

The problem we address in this paper can be formulated as follow: given a database D of linear sequences, a minimum support, σ, and the optional parameters *minGap*, *maxGap*, *window*, *excludedEventsSet*, *startEventsSets* and *substitutionSets*, find *all* the *arbitrary* or *rigid gap* frequent sequence patterns that respect the defined constraints.

3 Algorithm

The proposed algorithm uses a Bottom-Up search space enumeration and a combination of frequent pairs of events to extend and find all the frequent sequence patterns. The algorithm is divided in two phases: *scanning phase* and *sequence extension phase.* Since the frequent sequences are obtained from the set of frequent pairs, the first phase of the algorithm consists in traversing all the sequences in the database and building two auxiliary data structures. The first structure contains the set of all pairs of events found in the database. Each pair representation points to the sequences where they appear (through a sequence identifier bitmap). The second data structure consists of a vertical representation of the database. It contains the positions or offsets of the events in the sequences where they occur. This information is required to ensure that the order of the events along the data sequence is respected. Both data structures are thought for quick information retrieval. At the end of the scanning phase we obtain a map of all the pairs of events present in the database and a vertical format representation of the original database. In the second phase the pairs of events are successively combined to find all the sequence patterns. These operations are fundamentally based on two properties:

Property 1 (Anti-Monotonic) *All supersequences of an infrequent sequence are infrequent.*

Property 2 (Sequence Transitive Extension) *Let* $S =< s_1 \dots s_n >$, C_S *is its cover list and* O_S *the list of the offset values of* S *for all the sequences in* C_S. *Let* $P = (s_j \rightarrow s_m)$, C_P *is it cover list and* O_P *the offset list of* $succ(P)$ *for all sequences in* C_P. *If* $succ(S) = pred(P)$, *i.e.,* $s_n = s_j$, *then the extended sequence* $E =< s_1 \dots s_n s_m >$ *will occur in* C_E, *where* $C_E = \{X : \forall X \text{ in } C_S \cap C_P, O_P(X) > O_S(X)\}$.

Hence, the basic idea is to successively extend a frequent pair of events with another frequent pair, as long as the predecessor of one pair is equal to the successor of the other. This joining step is sound provided that the above mentioned properties (1 and 2) are respected. The joining of pairs combined with a breadth first or a depth first traversal yields all the frequent sequences patterns in the database.

3.1 Scanning Phase

The first phase of the algorithm consists in the following procedure: For each sequence in D, obtain all ordered pairs of events, without repetitions. Consider the sequence 5 in the example database of table 1(a). The obtained pairs are: $1 \rightarrow 2$, $1 \rightarrow 3$, $1 \rightarrow 4$, $2 \rightarrow 2$, $2 \rightarrow 3$, $2 \rightarrow 4$ and $3 \rightarrow 4$. During the determination of the pairs of events the first auxiliary data structure, that consists of an N-bidimensional matrix, is built and updated. N corresponds to the size of the alphabet. The N^2 cells in the matrix correspond to the N^2 possible combinations of pairs. We call this structure the *Bitmap Matrix*. Each $Cell(i, j)$ contains the information relative to the pair $i \rightarrow j$. This information consists of a bitmap that indicates the presence (1) or the absence (0) in the respective sequence (i-th bit corresponds to the sequence i in D) and an integer that contains the support count. This last value allows a fast support checking. For each pair $i \rightarrow j$ we update the respective $Cell(i, j)$ in the Bitmap Matrix, by activating the bit corresponding to the sequence where the pair occurs and incrementing the support counter. As an example, for the pair $1 \rightarrow 3$, the $Cell(1, 3)$ is represented in figure 1(b):

Sid	Sequence
1	< 1 2 3 4 5 >
2	< 1 3 4 >
3	< 2 3 2 3 >
4	< 3 2 1 >
5	< 1 2 2 3 4 >

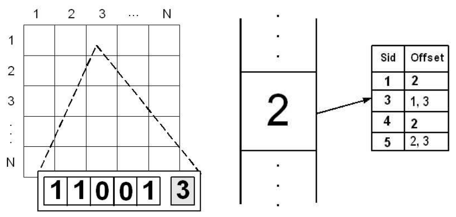

Fig. 1. (a) An example database; (b) Content of the Cell(1,3) in the Bitmap Matrix; (c) Representation of event 2 in the Offset Matrix

This means that the pair occurs in the database sequence 1, 2 and 5 and has a support of 3. Simultaneously, as each event in the database is being scanned, a second data structure called *Offset Matrix* is also being built. Conceptually, this data structure consists of an adjacency matrix that will contain all the offset (positions) of all the events in the entire database. Each event is a key that points to a list of pairs <*Sid*,*OffsetList*>, where *OffsetList* is a list of all the positions of the event in the sequence *Sid*. Thus, the Offset Matrix is a vertical representation of the database. Figure 1(c) shows the information stored in the Offset Matrix for the event 2.

3.2 Sequence Extension Phase

We start by presenting how arbitrary gap patterns are extracted. In section 3.4 we will show how easily our algorithm can be adapted to extract rigid gap patterns. For implementing the extension phase we present two tests (algorithms) that conjunctively are necessary and sufficient conditions to consider as frequent a new extended sequence.

```
input : $S(BaseSequence)$; $P(ExtensionPair)$; $\sigma(Min.Support)$
$C_S = bitmap(S)$ and $C_P = bitmap(P)$
$C_{S'} = C_S \cap C_P$
if $support(C_{S'}) \geq \sigma$ , then return OK.
```
Algorithm 1: Support Test

This is a quick test that implements property 1. The *bitmap* function gets the correspondent bitmaps of S and P. The intersection operation is also very fast and simple and the support function retrieves the support of the intersection bitmap. This test allows the verification of a *necessary but not sufficient* condition for the extended sequence to be frequent. A second test is necessary to guarantee that the order of the events is kept along the sequences that $C_{S'}$ bitmap points to.

Algorithm 2 assumes that, for each frequent sequence, additional information besides the sequence event list is kept during the extension phase. Namely, the corresponding bitmap that for the case exposed in algorithm 1 will be $C_{S'}$ if S'

```
   input : $C_{S'}(Bitmap); S(Base\ Seq); P(Ext.\ Pair); \sigma(Sup.)$
1  $seqLst = getSeqIdLst(C_{S'})$;
2  $E_v = succ(P)$;
3  $cnt = 0$;
4  foreach Sid in seqLst do
5      $O_v$ = offsetLst($Sid$, $E_v$);
6      $Y$ = offsetLastEvent($Sid$, $S$);
7      $W$ = offsetStartEvent($Sid$, $S$);
8      if $\exists X \in O_v,\ X > Y$ then
9          $cnt = cnt + 1$;
10         if $(X - Y) < n$ then $n = (X - Y)$
11         if $(X - Y) > m$ then $m = (X - Y)$
12     end
13     $diffTest(cnt, \sigma)$;
14 end
15 if $cnt \geq \sigma$ then
16     return OK;
17 end
```

Algorithm 2: Order Test

is determined to be frequent. Also two offset lists in the form <*Sid*, *offset*> are kept. One will contain the offset of the last event of the sequence, *offsetLastEvent*, and will be used for the "Order Test". The second, *offsetStartEvent*, contains the offset of the first event of the sequence pattern in all the Sid where it appears. This will be used when the verification of the window constraint is performed. In the Order Test, given a bitmap resulted from the support test, the *getSeqIdLst* function returns the list of the sequence identifiers for the bitmap. The function *offsetLst* returns a list of offset values of the event in the respective Sid. For each sequence identifier it is tested whether the extension pair has an offset greater than the offset value of the extended sequence. This implements the computation of C_E and the offsetList of *succ*(E) as in property 2. At line 13 the $diffTest$ function performs a simple test to check whether the minimum support is still reachable. At the end of the procedure (lines 15 to 17) it is tested whether the order of the extended sequence pattern is respected in a sufficient number of database sequences. In the positive case the extended sequence is considered frequent. Given algorithm 1 and 2, property 3 guarantees the necessary and sufficient conditions to safely extend a base sequence into a frequent one.

Property 3 (Frequent Extended Sequence) *Given a minimum support σ, a frequent base sequence $S =< E_1 \dots E_n >$, where $|S| \geq 2$ and a pair $P = E_k \rightarrow E_w$. If $E_n = E_k$, then $S' =< E_1 \dots E_n\ g_{n,k}\ E_k >$, where $g_{n,k} = -x(n,m)-$ if in arbitrary gap mode or $-r(n)-$ if in rigid gap mode, is frequent if algorithm 1 and 2 return OK.*

3.3 Space Search Traversal

Guided by the Bitmap Matrix the search space can be traversed using two possible approaches: *breadth first* or *depth first.* For both cases the set of the frequent sequences starts as the set of frequent pairs. In the depth first mode it starts with a sequence of size 2 that is successively expand until it can not be further extended. Then we backtrack and start extending another sequence. The advantage of this type of traversal is that we don't need to keep all the intermediary frequent sequence information, in contrast with the breadth first traversal where all the information of the sequences size k need to be kept before the sequences of size k+1 are generated. This yields is some cases, a significant memory reduction.

3.4 Rigid Gap Patterns

The algorithm described in 2 is designed to mine arbitrary gap patterns. Using gIL to mine rigid gap patterns requires only minor changes in the Order Test algorithm. Lines 4 to 11 in algorithm 2 are rewritten in algorithm 3. In this algorithm, first it is collected (in gapLst) the size of all the gaps for a certain sequence extension. Next, for each gap size it is tested whether the extended sequence is frequent. One should note that for rigid gap patterns, two sequence patterns with the same events are considered different if the gaps between the events have different size, e.g., $< 1 \cdot\cdot 2 >$ is different from $< 1 \cdots\ 2 >$.

```
1  foreach Sid in seqLst do
2      $O_v$ = offsetLst($Sid$, $E_v$);
3      $Y$ = offsetLastEvent($Sid$, $S$);
4      $W$ = offsetStartEvent($Sid$, $S$);
5      if $\exists X \in O_v,\ X > Y$ then
6          $gap = X - Y; gapLst.add(gap);$
7      end
8  end
9  foreach R in gapLst do
10     foreach Sid in seqLst do
11         Repeat Step 2 to 4;
12         if $\exists X \in O_v,\ (X - Y) = R$ then
13             $cnt = cnt + 1;$
14         end
15     end
16     if $cnt \geq \sigma$ then
17         return OK;
18     end
19 end
```

Algorithm 3: Algorithm changes to mine rigid gap patterns

4 Constraints

The introduction of constraints in the *gIL* algorithm like *min/max gap*, *window size*, *items exclusion* is a straightforward process and translates into a considerable performance gain. These efficiency improvements are naturally expected since (depending on the values of the constraints) the search space can be greatly reduced. The introduction of *substitution sets* is also very easy to achieve. Implementing events exclusion constraint and substitution sets turns out to be a natural operation. Simple changes in the Bitmap Matrix (that guides the sequence extension) and in the Offset Matrix (discriminates the positions of the events in every sequence where they occur) enable this implementations. The new features are introduced between the scanning phase and the sequence extension phase. The min/max gap and window constraints constitute an additional to be applied when the sequence is extended.

4.1 Events Exclusion, Start Events and Substitution Sets

The event exclusion constraint is applied by traversing the rows and columns of the Bitmap Matrix where the excluded events occurs. At that positions the support [1] count variable in the respective cells is set to zero. Start events constraints are also straightforwardly implemented by allowing extensions only to the events in $StartEventSets$.

When substitution sets are activated, one or more sets of equivalent events are available. For each set of equivalent events one has to form the union of the rows (horizontal union) and columns (vertical union) in the Bitmap Matrix, where those events occur. The vertical union is similar to the horizontal union. Moreover, for all the equivalent events, one needs to pairwisely intersect the sequences where they occur and then perform the union of the offsetLists for the intersected sequences. This results in the new offsetLists of the equivalent events.

4.2 Min / Max Gap and Window Size

These constraints are trivially introduced in the "Order Test". In algorithm 2, the test in line 8 is extended with three additional tests: $(X-Y) < maxGap\ AND\ (X-Y) > minGap\ AND\ (X-W) < windowSize$.

5 Experimental Evaluation

We evaluated our algorithm along different variables using two collections of synthetic and real datasets. To generate the synthetic datasets we developed a sequence generator based on the Zipfian distribution. This generator receives the following parameters (see table 1(a)): number of sequences, average length of the sequences, alphabet size and a parameter p that expresses the skewness of the distribution. This generator has allowed us to generate sequences with a relative small alphabet. The evaluated variables for this datasets were: *support*, *dataset size (number of sequences)*, and *sequence size*. Additionally, we tested the mushroom dataset used at the FIMI workshop [4]. To represent real life data, we used several datasets of proteins. The Yeast (*saccharomyces cerevisiae*) dataset is available at [5] and PSSP used for protein secondary structure prediction [3]. We also used a subset of the non Outer Membrane proteins obtained from [8]. The properties for this datasets are summarized in table 1(b). It is interesting to notice that, for all datasets, gIL's scanning phase time is residual (less than 0.4 seconds).

Since gIL finds two types of patterns we performed evaluation against two different algorithms. Both are in memory algorithms, assuming that the database completely fits into main memory. For the arbitrary gap patterns from the all patterns class we compared gIL with the SPAM [1] algorithm. SPAM has shown

[1] Future interactions on this dataset still have the Bitmap Matrix intact since the bitmaps remain unchanged

Symbol	Meaning
S	Number of Sequences (x 10^3)
L	Avg. Length of the sequences
R	Alphabet Size
P	Distribution Skewness

DataSet	NumSeq	AlphabetSize	AvgLen	MinLen	MaxLen
Yeast	393	21	256	15	1859
PSSP	396	22	158	21	577
nonOM	60	20	349	53	1161
mushroom	8124	120	23	23	23

Table 1. (a) Parameters used in the synthetic data generator; (b) Properties of the proteins datasets

to outperform SPADE [11] and PrefixSpan [7] and is a state-of-the-art algorithm in transactional sequence pattern mining. The datasets suffer a conversion into the transactional dataset format, in order to be processed by SPAM. In this conversion each customer is considered as a sequence and each *itemset* contains a unique item (event).

For the rigid gap patterns we compared gIL with Teiresias [6], a well known algorithm from the bioinformatics community. It can be obtained from [2]. It is, as far as we know, the most complete and efficient algorithm for mining closed (called "most specific" in their paper) frequent rigid gap patterns. Closed patterns are a subset of all frequent sequence patterns. In this sense, gIL (which derives all patterns) tackles a more general problem and consequently considers a much larger search space than Teiresias. Besides minimum support, Teiresias uses two additional parameters. L and W are respectively the number of non-wild cards events in a pattern and the maximum spanning between two consecutive events. Since gIL starts by enumerating patterns with size 2, we will set L=2 and W to the maxGap value. All the experiments[2] were performed using exact discovery, i.e. without the use of substitution sets, and on a 1.5GHz Intel Centrino machine with 512MB of main memory, running windows XP Professional. The applications were written in C/C++ language.

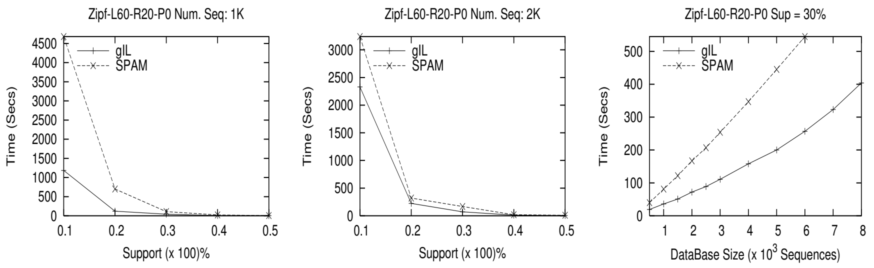

Fig. 2. (a) Support variation with Zipf database size=1K; (b) Support variation with Zipf database size=2K; (c) Scalability of gIL w.r.t. database size with a support of 30%

[2] Further details and results can be obtained from an extended version of this paper.

5.1 Arbitrary Gap Patterns Evaluation

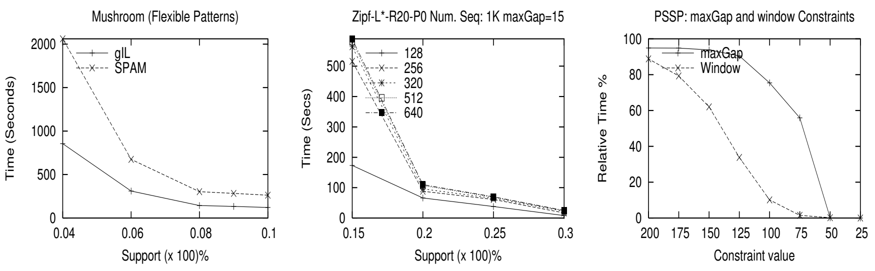

Fig. 3. (a) Support variation for the Mushroom dataset; (b) Scalability of gIL w.r.t sequence size for different support values (c) Performance evaluation using maxgap and windowgap constraints;

We start by comparing the efficiency of SPAM with the gIL algorithm without constraints. In figure 2(a) and 2(b) we tested different values of support for two datasets of $1K$ and $2K$ respectively. The sequences have an average length of 60 and an alphabet of 20 events. It was clear in these two experiments that for relative smaller dataset sizes and lower support values gIL becomes more efficient than SPAM. Figure 2(c) shows the scalability of the algorithms in respect to the dataset size for a support of 30%. This graphic shows that gIL scales well in relation to the dataset size.

In order to test a dataset with different characteristics, namely larger alphabet size, small length and greater dataset size, we used the Mushroom dataset, see figure 3(a). In figure 3(b) we have runtimes of gIL for datasets with one thousand sequences and different values of average sequence length. It was imposed a maxGap constraint of 15. As we observed during all the experiments, there is a critical point in the support variation, typically between 10% and 20%, that translates into an explosion of number of frequent patterns. This leads to an exponential behaviour in the algorithm's runtime. Even so, we can see that gIL shows similar behaviour for the different values of sequence length. Figure 3(c) measures the relative performance time, i.e. the ratio between the mining time with constraints and without constraints. These values were obtained for a support of 70%. Runtime without constraints was 305 seconds. It describes the behaviour of the algorithm when decreasing the maxGap and the Window values.

In respect to memory usage both algorithms showed a low memory demand for all the datasets. For the Mushroom dataset which was the most demanding in terms of memory, SPAM used a maximum of 9 MB for a support of 4% and gIL a constant memory usage of 26 MB for all the support values. gIL shows a constant and support independent memory usage since once the data structures are built for a given dataset they remain unchanged.

5.2 Rigid Gap Patterns Evaluation

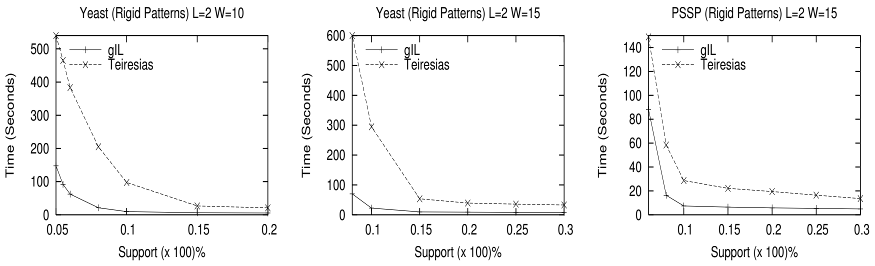

Fig. 4. (a) Support variation for the Yeast dataset, with L=2 and W(maxGap) = 10; (b) Support variation for the Yeast dataset, with L=2 and W(maxGap) = 15; (c) Support variation for the PSSP dataset, with L=2 and W(maxGap) = 15

In order to assess the performance of gIL in the mining of rigid gap patterns we compared it with Teiresias [6], for different proteins datasets. In figure 4(a) and 4(b) the Yeast dataset was evaluated for two values of maxGap(W), 10 and 15. The results showed that gIL outperforms Teiresias by an order of magnitude. When comparing the performance of the algorithms in relation to the PSSP (figure 4(c)) and the nonOM (figure 5(a)) datasets, for a maxGap of 15, gIL outperforms Teiresias by a factor of 2 in the first case. This difference becomes more significant in the second case. The nonOM dataset has a greater average sequence length, but a small dataset size. This last characteristic results into a smaller bitmap length yielding a significant performance improvement. As we already verified in the arbitrary gap experiments, gIL memory usage maintains nearly constant for all the tested support values (figure 5(b)). Figure 5(c) shows the linear scalability of gIL in relation to the number of frequent sequence patterns.

6 Conclusions

We presented an algorithm called *gIL*, suitable to work with databases of linear sequences with a long average length and a relative small alphabet size. Our experiments showed that for the particular case of the proteins datasets, gIL exhibits superior performance to state-of-the-art algorithms. The algorithm has a high adaptability, and thus it was easily changed to extract two different types of patterns: arbitrary and rigid gap patterns. Furthermore, the data organization allows a straightforward implementation of constraints and substitution sets. These features are pushed directly into the mining process, which in some cases enables the mining in useful time of otherwise untractable problems. In this sense gIL is an interesting and powerful algorithm to be applied in a broader range

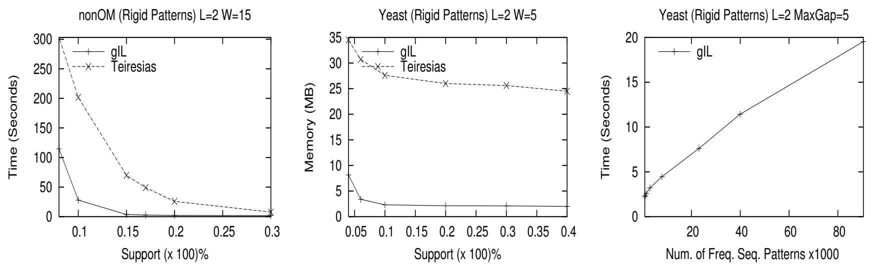

Fig. 5. (a) Support variation for the nonOM dataset, with L=2 and W(maxGap) = 15; (b) Memory usage for the Yeast dataset, with L=2 and W(maxGap) = 5; (c) Scalability of gIL w.r.t number of sequences for the Yeast dataset

of domains and in particular suitable for biological data. Thus, even when performing extensions an event at a time (using a smart combination of some of the most efficient techniques that have been used in the task of itemset and sequence mining) one can obtain an algorithm that efficiently handles the explosive nature of pattern search, inherent to the biological sequence datasets.

References

1. J. Ayres, J. Flannick, J. Gehrke, and T. Yiu. Sequential pattern mining using a bitmap representation. In *Proceedings of the 8th SIGKDD International Conference on KDD and Data Mining, 2002.*
2. IBM Bioinformatics. Teiresias. http://www.research.ibm.com/bioinformatics/.
3. James Cuff and Geoffrey J. Barton. Evaluation and improvement of multiple sequence methods for protein secondary structure prediction. In *PROTEINS: Structure, Function, and Genetics*, number 34. WILEY-LISS, INC, 1999.
4. Fimi. Fimi workshop 2003 (mushroom dataset). http://fimi.cs.helsinki.fi/fimi03.
5. GenBank. yeast (saccharomyces cerevisiae). www.maths.uq.edu.au.
6. A.Floratos I. Rigoutsos. Combinatorial pattern discovery in biological sequences: the teiresias algorithm. *Bioinformatics*, 1(14), January 1998.
7. J. Pei, J. Han, B. Mortazavi-Asl, H. Pinto, Q. Chen, U. Dayal, and M.-C. Hsu. PrefixSpan: Mining sequential patterns efficiently by prefix projected pattern growth. In *Proceedings of the International Conference on Data Engineering, ICDE 2001.*
8. Psort. Psort database. http://www.psort.org/.
9. Ramakrishnan Srikant and Rakesh Agrawal. Mining sequential patterns: Generalizations and performance improvements. In *Proceedings 5th International Conference on Extending DataBase Technology*, 1996.
10. Mohammed J. Zaki. Sequence mining in categorical domains: Incorporating constraints. In *In Proceedings of 9th International Conference on Information and Knowledge Management, CIKM 2000.*
11. Mohammed J. Zaki. Spade: An efficient algorithm for mining frequent sequences. *Machine Learning*, 42(1-2):31–60, 2001.