Mining Compressing Sequential Patterns

Hoang Thanh Lam[♯], Fabian Mörchen[♭], Dmitriy Fradkin[§] and Toon Calders[♯]

[§]Siemens Corporation — Corporate Research and Technology — Princeton, NJ, USA

[♭] Amazon.com Inc — 410 Terry Avenue North — Seattle WA 98109 USA

[♯]Technische Universiteit Eindhoven — Department of Maths and Computer Science — Eindhoven, Netherlands

Emails: t.l.hoang@tue.nl, moerchen@amazon.com, dmitriy.fradkin@siemens.com, and t.calders@tue.nl

Abstract

Pattern mining based on data compression has been successfully applied in many data mining tasks. For itemset data, the Krimp algorithm based on the *Minimum Description Length* (MDL) principle was shown to be very effective in solving the redundancy issue in descriptive pattern mining. However, for sequence data, the redundancy issue of the set of frequent sequential patterns is not fully addressed in the literature. In this paper we study MDL-based algorithms for mining non-redundant sets of sequential patterns from a sequence database. First, we propose an encoding scheme for compressing sequence data with sequential patterns. Second, we formulate the problem of mining the most compressing sequential patterns from a sequence database. We show that this problem is intractable and belongs to the class of inapproximable problems. Therefore, we propose two heuristic algorithms. The first of these uses a two-phase approach similar to Krimp for itemset data. To overcome performance issues in candidate generation, we also propose GoKrimp, an algorithm that directly mines compressing patterns by greedily extending a pattern until no additional compression benefit of adding the extension into the dictionary. Since checks for additional compression benefit of an extension are computationally expensive we propose a dependency test which only chooses related events for extending a given pattern. This technique improves the efficiency of the GoKrimp algorithm significantly while it still preserves the quality of the set of patterns. We conduct an empirical study on eight datasets to show the effectiveness of our approach in comparison to the state of the art algorithms in terms of interpretability of the extracted patterns, run time, compression ratio, and classification accuracy using the discovered patterns as features for different classifiers.

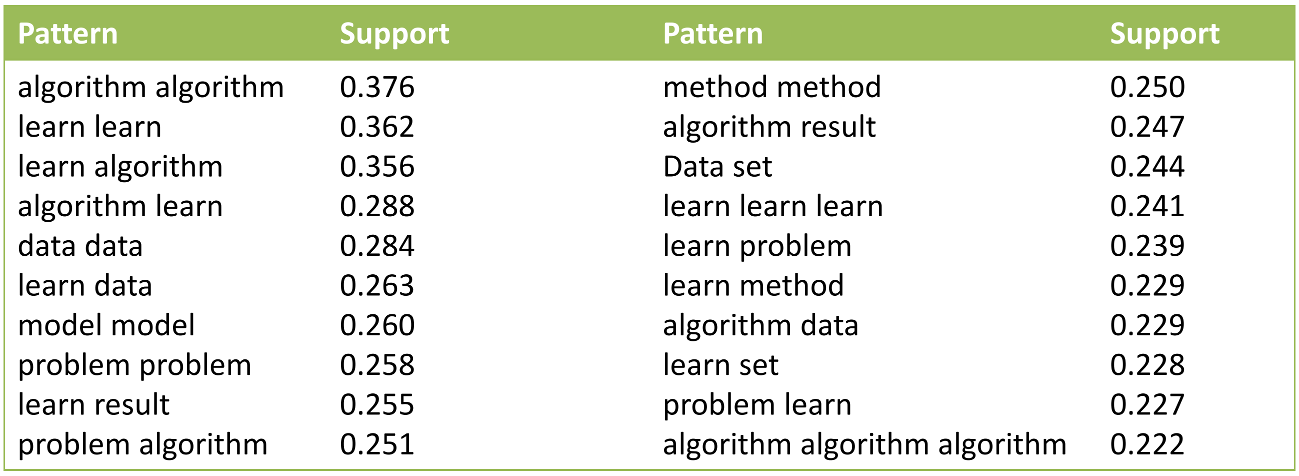
1 Introduction

Mining frequent sequential patterns from a sequence database is an important data mining problem which has been attracting researchers for more than a decade. Dozens of algorithms [12] to find sequential patterns effectively have been proposed. However, relatively few researchers have addressed the problem of reducing redundancy, ranking patterns by interestingness, or using the patterns for solving further data mining problems.

Redundancy is a well known problem in sequential pattern mining. Let us consider the JMLR dataset which contains a database of word sequences, each corresponding to an abstract of an article in the *Journals of Machine Learning Research.* Figure 1 shows the 20 most frequent closed sequential patterns ordered by decreasing frequency. This set of patterns is clearly very redundant, so many patterns with very similar meaning are shown to users.

Beside redundancy issues, the set of frequent patterns usually contain trivial and meaningless patterns. In fact, the set of frequent closed patterns in Figure 1 contains random combinations or repeats of frequent terms in the JMLR abstracts such as *algorithm, result, learn, data* and *problem.* These patterns are meaningless given our knowledge about the frequent terms.

In order to solve these issues, we have to find alternative interestingness measures rather than relying on frequency alone. For itemset data, an interesting approach has been proposed recently. The Krimp algorithm mines patterns that compress the data well [1] using the Minimum Description Length principle [10]. This approach has been shown to reduce redundancy and generate patterns that are useful for classification [1], component identification [3] and change detection [4]. We extend these ideas to sequential data. The key issue in designing an MDL-based algorithm for sequence data is the encoding scheme that determines how a sequence is compressed given some patterns. In contrast to itemsets we need to consider the ordering of elements in a sequence and need to be able to deal with gaps, as well as overlapping and repeating patterns; all properties not present in itemset data.



Pattern	Support	Pattern	Support
algorithm algorithm	0.376	method method	0.250
learn learn	0.362	algorithm result	0.247
learn algorithm	0.356	Data set	0.244
algorithm learn	0.288	learn learn learn	0.241
data data	0.284	learn problem	0.239
learn data	0.263	learn method	0.229
model model	0.260	algorithm data	0.229
problem problem	0.258	learn set	0.228
learn result	0.255	problem learn	0.227
problem algorithm	0.251	algorithm algorithm algorithm	0.222

Figure 1: The 20 most frequent non-singleton closed sequential patterns from the JMLR abstracts datasets. This set, despite containing some meaningful patterns, is very redundant.

In this paper we study MDL-based algorithms for mining non-redundant and meaningful patterns from a sequence database. The key contributions of this work can be summarized as follows:

1. We propose a novel encoding for sequence data. Our encoding assigns shorter codewords for small gaps, thus penalizing pattern occurrences with longer gaps. It is shown to be more effective than the encoding proposed in our prior work [32]. Moreover, by using the Elias code for gaps [33], it allows to encode interleaved patterns which is prohibited in the encoding proposed recently in [31].

2. We discuss the complexity of mining compressing patterns from sequence database. The main result shows that this problem is NP-hard and belongs to the class of inapproximable problems.

3. We propose SeqKrimp, a two-phase candidate-based algorithm for mining compressing patterns inspired by the original Krimp algorithm.

4. We propose GoKrimp, an efficient algorithm that directly mines compressing patterns from the data by greedily extending patterns until no additional compression benefit is observed. In order to avoid exhaustive checks of all possible extensions a dependency test technique is proposed which considers only related events for extension. This technique helps the GoKrimp algorithm to be faster than SeqKrimp and the state of the art algorithms while being able to find patterns with similar quality.

5. We perform an empirical study with one synthetic and eight real-life datasets to compare different sets of patterns based on the interpretability of the patterns and on the classification accuracy when they are used as attributes for classification tasks.

2 Related work

Mining useful patterns is an active research topic of data mining. Recent approaches can be classified into three major categories: statistical approaches based on hypothesis tests, MDL-based approaches and information-theoretic approaches.

The first direction is concerned with statistical hypothesis testing. The data is assumed to follow a user-defined null hypothesis. Subsequently, standard statistical hypothesis testing is used to test the significance of patterns assuming that the data follows the null hypothesis. If a pattern passes the test it is considered significant and interesting. For example, Gionis et al. [19, 30] use swap randomization to generate random transactional data from the original data. The significance of a given pattern is estimated on randomized data. A similar method is proposed for graph data by Hanhijärvi et al. [21] and Milo et al. [20]. In those works, random graphs with prescribed degree distribution are generated, and significance of a subgraph is estimated on the set of random generated graphs. A similar approach has also been applied to find interesting motifs in time-series data by Castro et al. [22].

A drawback of such approaches is that the null hypothesis must be chosen explicitly by the users. This task is not trivial in different types of data. Frequently, the null hypothesis is too naive and does not fit the real-life data. As a result, all the patterns may pass the test and be considered as significant.

Other research tries to identify interesting sets of patterns without making any assumptions on the underlying data distribution. The approach is based on the MDL principle: it searches for patterns that compress the given data most. Examples of this direction include the Krimp algorithm [1] and direct mining descriptive patterns algorithm [29] for itemset data and the algorithms for graph data [23, 24]. The usefulness of compressing patterns was demonstrated in various applications such as classification [1], component identification [3] and change detection [4].

The idea of using data compression for data mining was first proposed by Cilibrasi et al. [28] for data clustering problem. This idea was also explored by Keogh et al. [26], who proposed to use compressibility as a measure of distance between two sequences. They empirically showed that by using this measure for classification, they were able to avoid setting complicated parameters, which is not trivial in many data mining tasks, while obtaining promising classification results. Another related work by Faloutsos et al. [27] suggested that there is a connection between data mining and Kolmogorov complexity. While the connection was explained informally there, this notion quickly became the central idea for a lot of recent work on the same topic.

Our work is a continuation of this idea in the specific context of sequence data. In particular, it focuses on using the MDL principle to discover interesting sequential patterns. This paper is an extended version of our previous work on the same topic [32]. That work used an encoding scheme which assumes that the cost of storing a number or a symbol is always a constant. Therefore, it does not punish the gaps between events of a pattern which results in using a window constraint parameter to limit a match with a pattern within the constraint window size. Following that work, Tatti and Vreeken [31] proposed the SQS-Search (SQS) approach that punishes gaps by using an encoding with zero cost for encoding non-gaps and higher cost for encoding events with larger gaps. The approach was shown to be very effective in mining meaningful descriptive patterns in text data. However, it does not handle the case of interleaving patterns. In practice, patterns generated by independent processes may frequently overlap. In this work, we propose an encoding that both punishes gaps and handles interleaving patterns.

3 Preliminaries

3.1 Sequential Pattern Mining Let $S = (e_1, t(e_1)), (e_2, t(e_2)), \cdots, (e_n, t(e_n))$ denote a sequence of events, where $e_i \in \Sigma$ is an event symbol from an alphabet Σ and $t(e_i)$ is a timestamp of the event e_i.

Table 1: Notation

Notation	**Meaning**
$\mathfrak{S}$	a database
S	a sequence
D	a dictionary
$\mathcal{C}$	an encoding
$\sum$	an alphabet
e	an event represented by a symbol in $\sum$
$t(e)$	timestamp of the event e
$C(w)$	binary representation of w
$\|C(w)\|$	binary representation length
$L(\mathfrak{S})$	length of data before compression
$L^{\mathcal{C}}(D)$	length of the dictionary
$L^{\mathcal{C}}(\mathfrak{S}\|D)$	length of the data given it is encoded by D with the encoding $\mathcal{C}$
$L^{\mathcal{C}}_D(\mathfrak{S})$	total description length of the data in the encoding $\mathcal{C}$ with dictionary D

Given a sequence P, we say that S *matches* P if P is a subsequence of S.

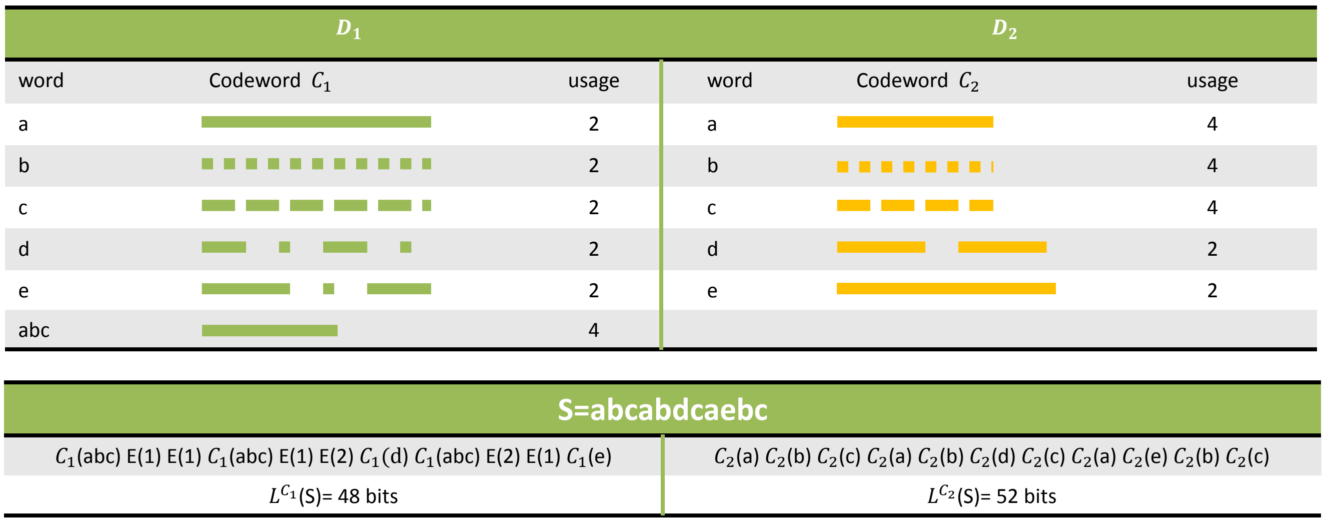
Let $\mathfrak{S} = \{S_1, S_2, \cdots, S_N\}$ be a database of sequences. The number of sequences in the database matching P is the *support* f_P of the given sequence. The frequent sequential pattern mining problem is defined as follows:

DEFINITION 1. (FREQUENT PATTERN MINING) *Given a sequence database $\mathfrak{S}$ a minimum support value minsup, find all sequences of events P such that $f_P \geq minsup$.*

A pattern P is called *closed* if it is frequent and there is no frequent pattern Q such that $f_P = f_Q$ and $P \subset Q$. The problem of mining all closed frequent patterns is formulated as follows:

DEFINITION 2. (CLOSED PATTERN MINING) *Given a database of sequences $\mathfrak{S}$ and a minimum support value minsup, find all patterns P such that $f_P \geq minsup$ and P is closed.*

3.2 Minimum Description Length Principle We briefly introduce the MDL principle and MDL-based pattern mining approaches in this subsection. A model M is a set of patterns $M = \{P_1, P_2, \cdots, P_m\}$ used to compress a database $\mathfrak{S}$. Let $L^{\mathcal{C}}(M)$ be the description length of the model M and $L^{\mathcal{C}}(\mathfrak{S}|M)$ be the description length of the database $\mathfrak{S}$ when it is encoded with the help of the model M in an encoding $\mathcal{C}$. Therefore the total description length of the data is $L^{\mathcal{C}}_M(\mathfrak{S}) = L^{\mathcal{C}}(M) + L^{\mathcal{C}}(\mathfrak{S}|M)$. Different models and encodings will lead to different description lengths of the database.



D_1			D_2		
word	Codeword C_1	usage	word	Codeword C_2	usage
a		2	a		4
b		2	b		4
c		2	c		4
d		2	d		2
e		2	e		2
abc		4			

S=abcabdcaebc	
C_1(abc) E(1) E(1) C_1(abc) E(1) E(2) C_1(d) C_1(abc) E(2) E(1) C_1(e)	C_2(a) C_2(b) C_2(c) C_2(a) C_2(b) C_2(d) C_2(c) C_2(a) C_2(e) C_2(b) C_2(c)
L^{C_1}(S)= 48 bits	L^{C_2}(S)= 52 bits

Figure 2: An example of two dictionaries and two encodings of the same sequence $S = abcadbcaebc$. In every dictionary, words are associated with codewords. Words with more usage are assigned with shorter codewords.

Informally, the MDL principle states that the best model is the one that compresses the data the most. Therefore, the MDL principle [10] suggests that we should look for the model M and the encoding $\mathcal{C}$ such that $L^{\mathcal{C}}_M(\mathfrak{S}) = L^{\mathcal{C}}(M) + L^{\mathcal{C}}(\mathfrak{S}|M)$ is minimized.

The central question in designing an MDL-based algorithm is how to encode data given a model. In an encoding, the data description length is fully determined by an implicit probability distribution assumed to be the true distribution generating the data. Therefore, designing an encoding scheme is as important as choosing an explicit probability distribution generating data in classical Bayesian statistics.

4 Data Encoding Scheme

In this section we explain how to encode the data given a set of sequential patterns.

4.1 Dictionary presentation Let $\sum = \{a_1, a_2, \cdots, a_n\}$ be an alphabet containing a set of characters a_i. A dictionary D is a table with two columns: the first column contains a list of words $w_1, w_2, \cdots, w_m$ including also all the characters in the alphabet $\sum$, while the second column contains a list of codewords of every word w_i in the dictionary denoted as $C(w_i)$. Codewords are unique identifiers of the corresponding words and may have different length depending on the word usage, as defined in subsection 4.3.

The binary representation of a dictionary is given as follows: it starts with n codewords of all the characters in the alphabet followed by the binary representations of all non-singleton dictionary words. For any non-singleton word w, its binary representation contains a sequence of codewords of its characters followed by its codeword $C(w)$. For instance, the word $w = abc$ is represented in the dictionary as $C(a)C(b)C(c)C(w)$. This binary representation of the dictionary allows us to get any word from the dictionary given its codeword.

Number	Elias code E(n)	Number	Elias code E(n)
1	1	5	00101
2	010	6	00110
3	011	7	00111
4	00100	8	0001000

Figure 3: An example of Elias codes of the first eight natural numbers. Code length of $E(n)$ is equal to $2\lfloor log_2(n)\rfloor + 1$.

Example 1. (Dictionary) *In Figure 2, two different dictionaries D_1 and D_2 are shown. The first dictionary contains both singleton and non-singleton words while the second one has only singletons. As an example, the binary representation of the first dictionary is $C_1(a)C_1(b)C_1(c)C_1(d)C_1(e)C_1(a)C_1(b)C_1(c)C_1(abc)$.*

4.2 Natural number encoding In our sequence encoding, we need a binary representation of natural numbers used to indicate gaps between characters in an encoded word. For any natural number n when the upper-bound on n is undefined in advance, the *Elias code* is usually used [33].

The Elias code of any natural number n denoted as $E(n)$ starts with exactly $\lfloor log_2(n)\rfloor$ zero bits followed by the actual binary representation of the natural number n. In this way, the Elias code length is equal to $2\lfloor log_2(n) \rfloor + 1$ bits which makes the encoding universal in the sense that when the upper-bound of n is unknown in advance, the Elias code length is at most twice as long as the optimal code length. In the Elias coding, the larger the value of n is the longer the code length $|E(n)|$ is, therefore, short gaps is encoded more succinct than long gaps.

EXAMPLE 2. (ELIAS ENCODING) *An example of Elias codes is depicted in Figure 3 where the Elias codes of the first eight natural numbers are shown. The number 8 has the Elias code as* $E(8) = 0001000$ *starting with* $\lfloor log_2(n) \rfloor = 3$ *zeros and followed by the binary representation* 1000 *of the number n=8*

Decoding a binary string containing several Elias codes is simple. In fact, the decoder first reads the leading zero bits until it reaches a one bit. At this moment, it knows how many more bits it needs to read to reach the end of the current Elias code. This process is repeated for every block of Elias code to decode the binary string completely.

EXAMPLE 3. (ELIAS DECODING) *The binary string* $\underline{000}1000\underline{00}100$ *can be decoded as follows: the decoder reads the first 3 zero bits, it knows that it needs to read 4 more bits to finish the current block. The obtained Elias code is decoded as the number 8. It continues to read the following 2 zero bits and reads another 3 bits to get the complete representation of the next number which is decoded as 4 in this case.*

4.3 Sequence encoding Given a dictionary D, a sequence S is encoded by replacing instances of dictionary words in the sequence by pointers. A pointer p replacing an instance of a word w in a sequence S is a sequence of bits starting by the codeword $C(w)$ followed by a list of Elias codes of the gaps indicating the difference between positions of consecutive characters of the instances of word in S. In the case the word is a singleton, the pointer contains only the codeword of the corresponding singleton.

EXAMPLE 4. (POINTERS) *In the sequence* $S = \underline{abcab}d\underline{ca}e\underline{bc}$ *three instances of the word* $w = abc$ *at positions* $(1, 2, 3)$, $(4, 5, 7)$ *and* $(8, 10, 11)$ *are underlined. If the word abc already exists in the dictionary with the codeword* $C(w)$ *then the three occurrences can be replaced by three pointers* $p_1 = C(w)E(1)E(1)$, $p_2 = C(w)E(1)E(2)$ *and* $p_3 = C(w)E(2)E(1)$.

A sequence encoding can be defined as follows:

DEFINITION 3. (SEQUENCE ENCODING) *Given a dictionary, a sequence encoding of* S *is a replacement of instances of dictionary words by pointers.*

The encoding in Definition 3 is complete if all characters in the sequence S are encoded. In this work, we consider only complete encoding. In an encoding C of a sequence S, the usage of a word w denoted as $f_C(w)$ is defined as the number of times the word w is replaced by a pointer plus the number of times the word is present in the binary representation of the dictionary.

EXAMPLE 5. (SEQUENCE ENCODING) *In Figure 2, two dictionaries* D_1 *and* D_2 *are created based upon two encodings* C_1 *and* C_2 *of the sequence* $S = abcabdcaebc$. *The first encoding* C_1 *replaces three occurrences of the word abc in the sequence S by pointers. Therefore, the usage of abc in that encoding is counted as the number of pointers replacing abc plus the number of the occurrences of abc in the dictionary, thus,* $f_{C_1}(abc) = 4$. *Meanwhile, although a is not replaced by any pointers it is present twice in the binary representation of the dictionary, so* $f_{C_1}(a) = 2$. *Similarly, the usages of the other words are shown in the same figure.*

For every word w, the binary representation of the codeword $C(w)$ depends on its usage in the encoding. Denote $F_C = \sum_{w \in D} f_C(w)$ as the sum of the usages of all dictionary words in an encoding C. Relative usages of every word w defined as $\frac{f_C(w)}{F_C}$ which can be considered as a probability distribution defined on the space of all dictionary words because $\sum_{w \in D} \frac{f_C(w)}{F_C} = 1$.

According to [10], there exists a prefix-free encoding $C(w)$ such that the codeword length $|C(w)|$ is proportional to the entropy of the word, i.e. $|C(w)| \sim -\log \frac{f_C(w)}{F_C}$, i.e. shorter codewords are assigned to words with more usage. Such encoding is optimal over all encodings resulting in the same usage distribution of the dictionary words [10]. When the dictionary contains only singletons, the aforementioned encoding corresponds to the Huffman code [33]. In this work, we denote Huffman code as C_0 and consider the data in this encoding as the uncompressed representation of the data. In Figure 2 the second encoding corresponds to the Huffman code.

4.4 Sequence decoding In this subsection, we discuss the decoding algorithm for an encoded sequence. First, we show how to read the content of a dictionary from its binary representation. A binary representation of a dictionary can be decoded as follows:

1. Read codewords of all singletons until encountering a duplicate of any singleton codeword.

2. Step by step read codewords of every non-singleton w by reading the contents of w (a sequence of familiar codewords of singletons) until reaching a completely unseen codeword $C(w)$ which is considered as the codeword of w in the dictionary.

Example 6. (Dictionary decoding) *The dictionary D_1 in Figure 2 has the binary representation $C_1(a)C_1(b)C_1(c)C_1(d)C_1(e)C_1(a)C_1(b)C_1(c)C_1(abc)$. The decoder starts by reading codewords of all singletons a, b, c, d and e. It stops when a repeat of a codeword of a singleton is encountered, in this particular case, when it sees a repeat of $C_1(a)$. The decoder knows that the codeword corresponds to the beginning of a non-singleton so it continuously reads the following codewords of singletons until reaching a never-seen-before codeword $C_1(abc)$. The latter codeword corresponds to the non-singleton abc in the dictionary.*

Given the dictionary, a sequence can be decoded by reading every block of the binary string corresponding to a word replaced by a pointer. Each block is read as follows:

1. Read the codeword $C(w)$ and refer to the dictionary to get information about the word w.
2. If the word w is a singleton then it continues reading the next block. Otherwise, it uses the Elias decoder to get $|w| - 1$ gap numbers before continuing with the next block.

The following example shows how to decode the sequence $C_1(abc)$ $E(1)$ $E(1)$ $C_1(abc)$ $E(1)$ $E(2)$ $C_1(d)$ $C_1(abc)$ $E(2)$ $E(1)$ $C_1(e)$ with the help of the dictionary D_1:

Example 7. (Sequence decoding) *The decoder first reads $C_1(abc)$ then it refers to the dictionary and knows that the word length is three, therefore, it reads two numbers by using the Elias decoder. The decoder continues reading the next block $C_1(abc)$ $E(1)$ $E(2)$ in the same way to decode another instance of abc. After that it reaches the codeword $C_1(d)$; a reference to the dictionary tells the decoder that there is no following gap number so the decoder continues to read the next blocks in a similar way to decode the last instance of abc and the singleton e.*

4.5 Data Description Length Denote $g_C(w)$ as the total cost of encoding the gaps by the Elias codes of the word w in an encoding C. It is important to notice that the gap cost of singleton is always equal to zero. The description length of the database $\mathfrak{S}$ encoded by the encoding C can be calculated as follows:

$$L^C(\mathfrak{S}) = \sum_{w \in D} (|C(w)| * f_C(w) + g_C(w)) \quad (4.1)$$

$$= \sum_{w \in D} \left(\log \frac{F_C}{f_C(w)} * f_C(w) + g_C(w) \right) \quad (4.2)$$

5 Problem Definition

We denote $L_D^{C_D^*}(\mathfrak{S})$ as the length of the database $\mathfrak{S}$ in the optimal encoding C_D^* when the dictionary D is given. The problem of finding compressing patterns is formulated as follows:

Definition 4. (Compressing Sequences Problem) *Given a sequence database $\mathfrak{S}$, find an optimal dictionary D^* and also optimal the encoding $C_{D^*}^*$ that use words in the dictionary D^* to encode the database $\mathfrak{S}$ such that $D^* = argmin_D L_D^{C_D^*}(\mathfrak{S})$.*

In order to solve the *Compressing Sequences Problem* we need to find at the same time the optimal dictionary D^* and the optimal encoding C_D^* that uses the dictionary D^* to encode the database $\mathfrak{S}$.

6 Complexity Analysis

This section discusses the complexity of the mining Compressing Sequences Problems. Finding a dictionary that compresses the database most is equivalent to finding a set of patterns that gives the most compression benefit defined as the difference between database description length before and after compression. The following theorem shows that even finding a dictionary containing all the singletons and one non-singleton pattern that gives the most compression benefit is inapproximable:

Theorem 1. *Finding the most compressing pattern is inapproximable.*

In order to prove Theorem 1, we reduce the most compressing pattern problem to the *Maximum Tile in Database problem* [16]. Given an itemset database $\mathfrak{D} = \{T_1, T_2, \cdots, T_n\}$, where every T_i is an itemset defined over an alphabet $\sum = \{a_1, a_2, \cdots, a_m\}$. The area of an itemset $I \subset \sum$ denoted as $A(I)$ is calculated as the size of I multiplying by the frequency of I in the database. The maximum tile problem looks for the itemset having the largest area. Mining the maximum tile is equivalent to finding the maximum clique in a bipartite graph known as an inapproximable problem in the literature [17].

From the itemset database $\mathfrak{D}$ we create a sequence database $\mathfrak{S} = \{S_1, S_2, \cdots, S_n\}$ as follows. First, distinct symbols $b_1, b_2, \cdots, b_M$ are added to $\sum$ to obtain a new alphabet $\sum^+$. Each transaction $T_i \in D$ is sorted increasingly according to any lexicographical order defined over $\sum^+$. Assume that T_i has the form $a_{i_1}, a_{i_2}, \cdots, a_{i_k}$ after sorting, therefore, a sequence S_i is created as such $S_i = (a_{i_1}, 1), (a_{i_2}, 2), \cdots, (a_{i_k}, k)$. Besides, in the database $\mathfrak{S}$ we add an additional sequence S_{n+1} such that it contains all the symbols in $\{b_1, b_2, \cdots, b_M\}$ sorted increasing according to the lexicographical order. Let $N > 1$ be the sum of the lengths of all sequences except the last sequence in $\mathfrak{S}$.

In the Huffman encoding C_0 of $\mathfrak{S}$ using only singletons the description length of $\mathfrak{S}$ is:

$$\begin{aligned} L^{C_0}(\mathfrak{S}) &= \sum_{i=1}^{m} f_{C_0}(a_i) \log \frac{F_{C_0}}{f_{C_0}(a_i)} + \\ &\quad \sum_{i=1}^{M} f_{C_0}(b_i) \log \frac{F_{C_0}}{f_{C_0}(b_i)} \\ &= F_{C_0} \log F_{C_0} - \sum_{i=1}^{m} f_{C_0}(a_i) \log f_{C_0}(a_i) \\ &\quad -2M \end{aligned}$$

Let $P = a_{i_1} a_{i_2} \cdots a_{i_{|P|}}$ be any non-singleton word with $|P|$ characters and let C_P be an encoding that use a dictionary D_P containing only one non-singleton P to encode the data $\mathfrak{S}$ by replacing $f_{C_P}(P) - 1$ occurrences of P in the database. The description length of the database $\mathfrak{S}$ is:

$$\begin{aligned} L^{C_P}(\mathfrak{S}) &= \sum_{i=1}^{m} f_{C_P}(a_i) \log \frac{F_{C_P}}{f_{C_P}(a_i)} \\ &\quad + \sum_{i=1}^{M} f_{C_P}(b_i) \log \frac{F_{C_P}}{f_{C_P}(b_i)} \\ &\quad + f_{C_P}(P) \log \frac{F_{C_P}}{f_{C_P}(P)} + g_{C_P}(P) \\ &= F_{C_P} \log F_{C_P} \\ &\quad - \sum_{i=1}^{m} f_{C_P}(a_i) \log f_{C_P}(a_i) - 2M \\ &\quad - f_{C_P}(P) \log f_{C_P}(P) + g_{C_P}(P) \end{aligned}$$

We first prove two supporting lemmas from which Theorem 1 is a direct consequence.

Lemma 1. *If M is chosen such that $F_{C_0} > N^8 + N$ then:*

$$0.5 \log \frac{F_{C_0} - N}{N^8} \leq \frac{L^{C_0}(\mathfrak{S}) - L^{C_P}(\mathfrak{S})}{|P| f_{C_P}(P)} \leq 3 \log F_{C_0}$$

Proof. First since function $\frac{x}{\log x}$ is increasing for any $x > 2$ so we have a support inequality $\frac{x}{\log x} > \frac{y}{\log y}$ for any $x > y > 2$.

Since $F_{C_0} = F_{C_P} + |P| f_{C_P}(P) - |P| - f_{C_P}(P)$ we have $F_{C_0} > F_{C_P}$ and $|P| f_{C_P}(P) > F_{C_0} - F_{C_P} > \frac{|P| f_{C_P}(P)}{2}$. From which we first imply that:

$$\begin{aligned} F_{C_0} \log F_{C_0} - F_{C_P} \log F_{C_P} &\geq \frac{|P| f_{C_P}(P) \log F_{C_P}}{2} \\ &\geq \frac{|P| f_{C_P}(P) \log (F_{C_0} - N)}{2} \end{aligned}$$

Moreover, since $F_{C_0} > F_{C_P}$ we have $\frac{F_{C_0}}{\log F_{C_0}} > \frac{F_{C_P}}{\log F_{C_P}}$ from which we further imply that:

$$\begin{aligned} (F_{C_0} - F_{C_P})(\log F_{C_0} + \log F_{C_P}) &\geq \\ F_{C_0} \log F_{C_0} - F_{C_P} \log F_{C_P} & \\ 2|P| f_{C_P}(P) \log F_{C_0} \geq F_{C_0} \log F_{C_0} - F_{C_P} \log F_{C_P} & \end{aligned}$$

Besides, $f_{C_0}(a_i) = f_{C_P}(a_i)\ \forall a_i \notin P$, $f_{C_P}(P) > f_{C_0}(a_i) - f_{C_P}(a_i) > 0\ \forall a_i \in P$ and $f_{C_0}(a_i) < N$. Therefore, we have:

$$\begin{aligned} 0 \geq \sum_{i=1}^{m} f_{C_P}(a_i) \log f_{C_P}(a_i) - \sum_{i=1}^{m} f_{C_0}(a_i) \log f_{C_0}(a_i) &\geq \\ \sum_{a_i \in P} (f_{C_P}(a_i) - f_{C_0}(a_i))(\log f_{C_P}(a_i) + \log f_{C_0}(a_i)) &\geq \\ -2|P| f_{C_P}(P) \log N & \end{aligned}$$

Moreover, since the gaps value always less than N, we have $0 > -g(P) > -2|P| f_{C_P}(P) \log N$ and $|P| f_{C_P}(P) \log F_{C_0} > f_{C_P}(P) \log f_{C_P}(P) > 0$. Sum up all the last obtained inequalities, we have:

$$0.5 \log \frac{F_{C_0} - N}{N^8} \leq \frac{L^{C_0}(\mathfrak{S}) - L^{C_P}(\mathfrak{S})}{|P| f_{C_P}(P)} \leq 3 \log F_{C_0}$$

from which the lemma is proved $\square$.

Lemma 2. *If there is an algorithm approximating the best compressing pattern of $\mathfrak{S}$ within a constant factor α in polynomial time then there exists a constant factor β such that we can approximate the maximum tile of the database $\mathfrak{D}$ within a constant factor β.*

Proof. Let denote P^* as the maximum tile of the database $\mathfrak{D}$. Let P be the pattern that approximates the best compressing pattern of $\mathfrak{S}$ within the constant factor α. We have:

$$L^{C_0}(\mathfrak{S}) - L^{C_P}(\mathfrak{S}) \geq \alpha(L^{C_0}(\mathfrak{S}) - L^{C_{P^*}}(\mathfrak{S}))$$

Based upon the results in Lemma 1, we can imply that:

$$3|P| f_{C_P}(P) \log F_{C_0} \geq 0.5\alpha |P^*| f_{C_{P^*}}(P^*) \log \frac{F_{C_0} - N}{N^8}$$

Algorithm 1 Compress($\mathfrak{S}|P$)

```
1: Input: A sequence database $\mathfrak{S} = \{S_1, S_2, \cdots, S_n\}$ and a pattern $P$
2: Output: the compress benefit of adding $P$ to the dictionary $D$
3: $L^{C_0}(\mathfrak{S}) \longleftarrow$ the original length of data
4: for $S_i \in \mathfrak{S}$ do
5:     while $S_i$ has an instance of P do
6:         $s \longleftarrow$ minGapMatch($S_i, P$)
7:         Replace $s$ by a pointer to $P$ in the dictionary
8:     end while
9: end for
10: $L(\mathfrak{S}|D \bigcup \{P\}) \longleftarrow$ length of the data after adding $P$
11: Return $L^{C_0} - L(\mathfrak{S}|D \bigcup \{P\})$
```

If M is chosen such that $F_{C_0} > N^{16} + 2N$, we have $\log \frac{F_{C_0} - N}{N^8} > \frac{\log F_{C_0}}{2}$, from which we further imply that:

$$|P| f_{C_P}(P) \geq \beta |P^*| f_{P^*}(P^*)$$

where $\beta = \frac{\alpha}{12}$ from which the lemma is proved. $\square$

It is obvious that Theorem 1 is a direct corollary of Lemma 2 because the reduction can be done in polynomial time of the size of the database (M is chosen such that F_{C_0} is a polynomial of the size of the data, in this case $F_{C_0} > N^{16} + 2N$). A direct corollary of Theorem 1 is that the *Compressing Sequences Problem* is NP-Hard:

THEOREM 2. *The* compressing pattern problem *is NP-hard.*

7 Algorithms

This section discusses two heuristic algorithms inspired by the idea of the Krimp algorithm to solve the compressing pattern mining problem. Before explaining these algorithms we first explain how to compress a sequence database using a single pattern as this procedure is used in both algorithms as a subtask.

7.1 Compress a database by a pattern Since mining compressing patterns is NP-hard, the heuristic solution greedily chooses the next pattern that gives the best compression benefit when added to the dictionary. Thus as a subtask of the greedy selection we need to evaluate the compression benefit of adding a given non-singleton pattern. This step can be performed by considering the following greedy encoding of the database $\mathfrak{S}$ using a pattern P.

Algorithm 1 looks for instances of P in S such that the positions of the characters in the match are close to each other. Intuitively, those matches give shorter encodings. Therefore, for every individual sequence S in the database it first looks for a match of P in S having the minimum cost to encode the gaps between consecutive characters of the match (line 6). Subsequently, this match is replaced with a pointer and is removed from the sequence (line 7). This step is repeated to find any other matches of P in S. The same procedure is applied for encoding the other sequences in the database. The algorithm returns the compression benefit of adding the pattern P to the dictionary and encoding the database by the greedy encoding procedure.

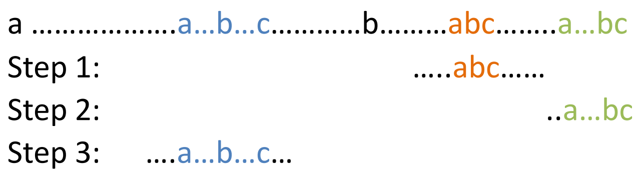

a ....a...b...c...........b.........abc........a...bc
Step 1: .....abc......
Step 2: ..a...bc
Step 3: ....a...b...c...

Figure 4: An example of the greedy encoding of the sequence S by the pattern $P = abc$. In every step it picks the match of P in S that has the minimum gap cost in the sequence S and replaces it with a pointer.

EXAMPLE 8. *As an example, Figure 4 shows every step of Algorithm 1 with a sequence S and a pattern $P = abc$. In the first step, the match with smallest gap cost is chosen and it is removed from the the sequence. The following two matches are chosen by the same procedure that looks for the match with minimum gap cost.*

An important task of the greedy encoding is to find the instance of $P = a_1 a_2 \cdots a_k$ having the minimum gap cost. This task can be done by using a dynamic programming method as follows. Let $l_{a_1}, l_{a_2}, \cdots, l_{a_k}$ be the lists associated with the characters of the pattern P: The $j-th$ element of a list l_{a_i} contains two fields denoted as $l^1_{a_i}[j]$ and $l^2_{a_i}[j]$. The first field $l^1_{a_i}[j]$ contains a position of a_i in the sequence S and the second field $l^2_{a_i}[j]$ contains the gap cost of the match of the word $a_1 a_2 \cdots a_i$ with minimum gap cost given that the match must end at the position $l^1_{a_i}[j]$. Algorithm 2 finds the match of the word P with minimum gap cost by scanning through all the lists l_{a_i} for $i = 1, 2, \cdots, k$ and for the j-th element of the list l_i it calculates $l^2_{a_i}[j]$ by using the following formula:

$$l^2_{a_i}[j] = \min_p \{l^2_{a_{i-1}}[p] + E(l^1_{a_i}[j] - l^1_{a_{i-1}}[p])\} \quad (7.3)$$

, where $l^2_{a_1}[j] = 0$ for $j = \overline{1, 2, \cdots, |l_{a_1}|}$.

EXAMPLE 9. (MATCH WITH MINIMUM GAP COST) *Figure 5 illustrates the basic steps of Algo-*

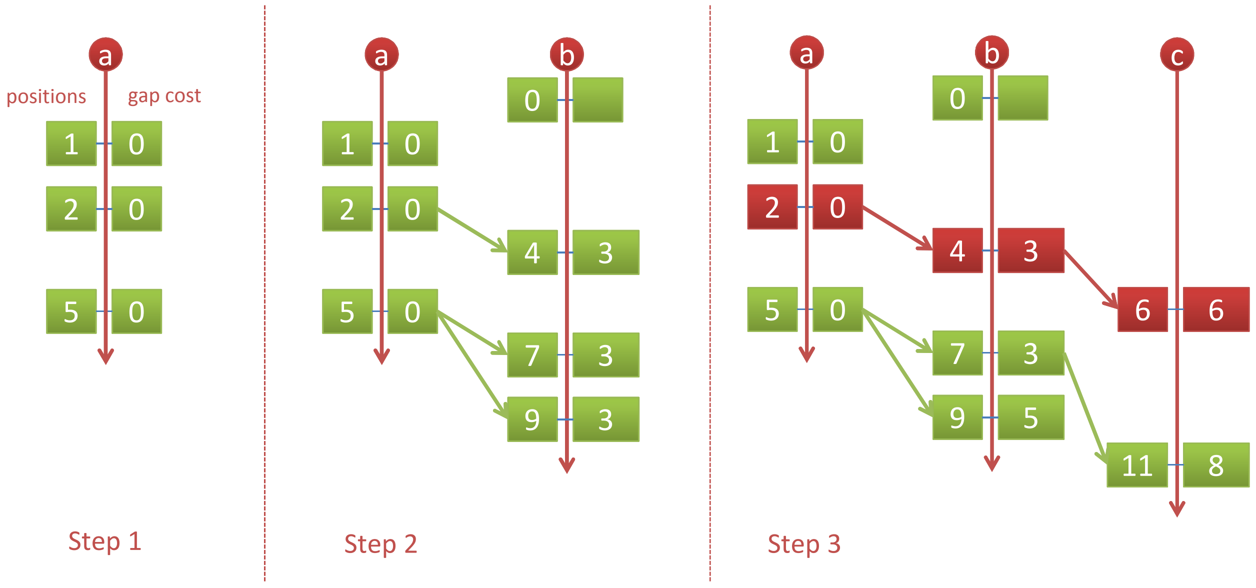


Figure 5: An example of dynamic programming algorithm to find the match of a pattern $P = abc$ with minimum gap cost in the sequence S

Algorithm 2 minGapMatch(S_t, P)

```
1: Input: a sequence $S_t$ and word $P = a_1 a_2 \cdots a_k$ and
   lists of positions $l_{a_1}, l_{a_2}, \cdots, l_{a_k}$
2: Output: the match with minimum gap cost
3: for $i = 1$ to $k$ do
4:   for $j = 1$ to $|l_{a_i}|$ do
5:     $l^2_{a_i}[j] = \min_p \{l^2_{a_{i-1}}[p] + E(l^1_{a_i}[j] - l^1_{a_{i-1}}[p])\}$
6:   end for
7: end for
8: Return match with minimum cost
```

rithm 2 finding the match with minimum gap cost of the word $w = abc$ in the sequence $S = (b,0)(a,1)(a,2)(b,4)(a,5)(c,6)(b,7)(b,9)(c,11)$.

Step 1: l_a contains 3 elements with $l^1_a[1] = 1$, $l^1_a[2] = 2$ and $l^1_a[3] = 5$ indicating the positions of a in the sequence S. First, we can initialize the second field of every element of the list l_a to zero.

Step 2: l_b contains 4 elements with $l^1_b[1] = 0$, $l^1_b[2] = 4$, $l^1_b[3] = 7$ and $l^1_b[4] = 9$ indicating the positions of b in the sequence S. According to formula 7.3 we can calculate the second field of every element of the list l_b as follows, for instance for $l^2_b[2]$:

$$\begin{aligned} l^2_b[2] &= \min_p \{l^2_a[p] + E(l^1_b[2] - l^1_a[p])\} \\ &= l^2_a[2] + E(l^1_b[2] - l^1_a[2]) \\ &= 3 \end{aligned}$$

We draw an arrow connecting $l_a[2]$ and $l_b[2]$ in order to keep track of the best match so far. The value of $l^2_a[3]$ and $l^2_a[4]$ can be calculated in a similar way.

Step 3: l_c contains 2 elements with $l^1_c[1] = 6$ and $l^1_c[2] = 11$ indicating the positions of c in the sequence S. The values of $l^2_c[1]$ and $l^2_c[2]$ can be obtained in the same way as in step 2. Among them $l^2_c[1] = 6$ bits is smallest so the match of abc in S with minimum gap cost corresponds to the instance of abc at positions $(2,4,6)$.

7.2 SeqKrimp, a Krimp-based algorithm for sequence Database In this subsection, we introduce an algorithm for mining compressing patterns from a sequence database similar to Krimp for itemset data. The SeqKrimp described in Algorithm 3 consists of two phases. In the first phase, a set of candidate patterns is generated by using a frequent closed sequential patterns mining algorithm (line 3).

In the second phase, the SeqKrimp algorithm chooses a good set of patterns from the set of candidates based upon a greedy procedure. It first calculates the compression benefit of adding a pattern $P \in \mathbf{C}$ to the current dictionary. The compression benefit is calculated with the help of Algorithm 1. The pattern P^* with the most additional compression benefit is included in the dictionary. Additionally, once P^* has been chosen, Algorithm 1 is used to replace all the instances of P^* in the data D by pointers to P^* in the dictionary. These actions are repeated as long as the candidate set $\mathbf{C}$ is not empty and there is still additional positive compression benefit to add a pattern.

Example 10. (SeqKrimp) *As an example, Figure 6*

Database	Dictionary D	Candidate set C	Benefit (P)
(a,1)(b,2)(c,3) (a,4)(b,5)(c,6)	$w_1 = a\ w_2 = b$	ab	-2
(a,1)(b,2)(c,3)(a,4)(b,5)(c,6)	$w_3 = c$	abc	11
(a,1)(b,2)(c,3)(a,4)(b,5)			
(a,1)(b,2)(c,3)(a,4)(b,5)		bc	-14
(a,1)(b,2)(c,3) (a,4)(b,5)(c,6)			
(a,1)(b,2)(c,3) (a,4)(b,5)(c,6)			

Step 1: abc is chosen

Database	Dictionary D	Candidate set C	Benefit (P)
……………………………………..	$w_1 = a\ w_2 = b$	ab	-4
……………………………………..	$w_3 = c$		
………………….(a,4)(b,5)	$w_4 = abc$		
………………….(a,4)(b,5)		cb	-∞
……………………………………..			
……………………………………..			

Step 2: stop as long as no positive additional benefit of adding a new pattern

Figure 6: An example illustrates how the SeqKrimp algorithm works

Algorithm 3 SeqKrimp($\mathfrak{S}$)

```
 1: Input: Database $\mathfrak{S}$
 2: Output: compressing patterns
 3: C ⟵ GetCandidate()
 4: $D \longleftarrow \sum$
 5: while $C \neq \emptyset$ and Benefit($P^*$) $> 0$ do
 6:    for $P \in$ C do
 7:       Benefit($P$) ⟵ Compress($\mathfrak{S}|P$)
 8:    end for
 9:    $P^* \longleftarrow argmax_P$Benefit($P$)
10:    if $Benefit(P^*) \leq 0$ then
11:       break
12:    end if
13:    $D \longleftarrow D \cup \{P^*\}$
14:    C ⟵ C $\setminus \{P^*\}$
15:    Using Algorithm 1 to replace all instances of $P^*$
       in $\mathfrak{S}$ by pointers
16: end while
17: Return $D$
```

Algorithm 4 GoKrimp($\mathfrak{S}$)

```
 1: Input: Database $\mathfrak{S} = \{S_1, S_2, \cdots, S_n\}$
 2: Output: the set of compressing patterns
 3: $D \longleftarrow \sum$
 4: while Benefit($P^*$)$> 0$ do
 5:    $P^* \longleftarrow$ GetNextPattern($\mathfrak{S}$))
 6:    if $Benefit(P^*) \leq 0$ then
 7:       break
 8:    end if
 9:    $D \longleftarrow D \cup \{P^*\}$
10:    Using Algorithm 1 to replace all instances of $P^*$
       in $\mathfrak{S}$ by pointers
11: end while
12: return $D$
```

shows each step of the SeqKrimp algorithm for a database and a candidate set. In the first step, the compression benefit of adding every candidate is calculated. The word abc is chosen because it gives the best additional compression benefit among the candidates. When abc is chosen the database is updated by replacing every instance of abc by a pointer. Subsequently, the compression benefit of the remaining candidates is recalculated accordingly. Finally, in step 2, since there is no additional compression benefit of adding a new pattern, the algorithm stops.

SeqKrimp suffers from the dependency on the candidate generation step that is very expensive for low minimum support thresholds. Even for moderate-size datasets state of the art algorithms for extracting frequent or closed patterns from sequence database such as PrefixSpan [11] or BIDE algorithm [9, 13] are very time-consuming.

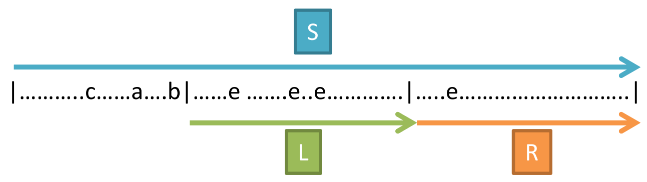

S
|..........c......a....b|......e ....e..e............|.....e...........................|
L
R

Figure 7: An example of how dependency test is carried out. If the event e is independent from the pattern $P = cab$ then it must occur in two equal-length subintervals L and R with the same chance.

Algorithm 5 GetNextPattern($\mathfrak{S}$)

```
1:  Input: Database $\mathfrak{S} = \{S_1, S_2, \cdots, S_n\}$
2:  Output: $P$ approximation of the best compressing pattern
3:  $P \longleftarrow \{\emptyset\}$
4:  $F \longleftarrow$ frequent events
5:  while (true) do
6:      for $e \in F$ do
7:          Benefit($e$) $\longleftarrow$ Compress($D|P.e$)
8:      end for
9:      $e^* \longleftarrow argmax_e$ Benefit($e$)
10:     if Benefit($e^*$) $< 0$ then
11:         Break
12:     end if
13:     $P \longleftarrow P.e^*$
14:     $\mathfrak{S} \longleftarrow \mathfrak{S}$ projected to $e^*$
15:     $F \longleftarrow$ related frequent events
16: end while
17: return $P$
```

7.3 Direct Mining Of Compressing Patterns

This subsection discusses a direct algorithm for mining compressing patterns. In particular, GoKrimp depicted in Algorithm 4 directly looks for the next most compressing pattern P^*. When a pattern has been obtained, the Compress procedure in Algorithm 1 is used to replace every instance of this pattern in the database by a pointer. These actions are repeated until there is no more additional compression benefit of adding a new pattern.

The most important subtask of the GoKrimp algorithm is a greedy procedure to obtain the next good compressing pattern from the data. *GetNextPattern*($\mathfrak{S}$) depicted in Algorithm 5 step by step extends every frequent event until no more additional compression benefit can be obtained. When all the extensions have been obtained the algorithm chooses among them the one with highest compression benefit to return as an output pattern.

The evaluation of each extension is very time consuming because it involves multiple searches for minimum gap matches of the extension in the database. Therefore the set of events chosen to extend a pattern is limited to the set of events being related to the occurrences of the given pattern. Indeed, the *GetNextPattern* algorithm adopts a dependency test to collect all the related events. Subsequently, the event when added to the given pattern giving the most compression benefit is chosen to extend that pattern. When an event has been chosen the database is projected to the event and the algorithm keeps extending the pattern as long as the extensions still adds more compression benefit.

In order to test the dependency between a pattern P and an event e we use the *Statistical Sign Test* [35]. Given m pairs of numbers $(X_1, Y_1)(X_2, Y_2), \cdots, (X_m, Y_m)$, denote N^+ as the number of pairs such that $X_i > Y_i$ for $i = 1, 2, \cdots, m$. If two sequences $X_1, X_2, \cdots, X_m$ and $Y_1, Y_2, \cdots, Y_m$ are generated by the same probability distribution then the test statistics N^+ follows a binomial distribution $B(0.5, m)$.

The sign test is applied to test the dependency between a pattern P and an event e as follows. For every sequence $S \in \mathfrak{S}$ and an event $c \in P$ denote $S(c)$ as the leftmost instance of c in S. Consider the interval right after the last position of $S(c)$ as illustrated in Figure 7. This interval is divided into two equal-length subintervals L and R. Denote the frequency of the event e in the two subintervals as L_e and R_e respectively. If the event e is independent from the occurrence of $S(c)$, we would expect that the chance e occurring in left and the right intervals is the same. Therefore, the number of sequences in which we observe $L_e > R_e$ can be used as a test statistics in the sign test for testing the dependency between the event e and the pattern c. The test is done for every event $c \in P$, an event e is considered as related to pattern P if it passes all the dependency tests regarding all the event belong to P. When a test has been done we keep log of the dependency results for reusing next time. In the next section, we empirically show that the dependency test speeds up the GoKrimp algorithm significantly while preserving the quality of the compressing patterns.

8 Experiments and Results

This section discusses results of experiments carried out several real-life and one sythetic dataset. We will compare the set of patterns produced by SeqKrimp and GoKrimp algorithms to the following baseline algorithms:

- BIDE: BIDE was chosen because it is a state of the art approach for closed sequential pattern mining. BIDE is also used to generate the set of candidates for SeqKrimp, i.e. an implementation of

Table 2: Summary of Datatsets

Datasets	Events	Sequences	Classes
jmlr	787	75,646	NA
parallel	1,000,000	10,000	NA
aslbu	36,500	441	7
aslgt	178,494	3,493	40
auslan2	1800	200	10
pioneer	9766	160	3
context	25,832	240	5
skating	37,186	530	7
unix	295,008	11,133	10

the **GetCandidate**(.) function in line 3 Algorithm 3.

- SQS: proposed recently by Tatti and Vreeken [31] for mining compressing patterns in sequence database
- pGOKRIMP : the prior version of the GoKrimp algorithm (denoted as pGoKrimp) published in our previous work [32]. We include pGOKRIMP in the comparison to demonstrate the effectiveness of the revised encoding adopted by the GOKRIMP algorithm.

We use seven different real-life datasets introduced in [14] to evaluate the proposed approaches in term of classification accuracy. Each dataset is a database of symbolic interval sequences with class labels. For our experiments the interval sequences are converted to event sequences by considering the start and end points of every interval as different events. A brief summary of the datasets is given in Table 2. All the benchmark datasets are available for download upon request at the website [1].

Besides, other two datasets are also used for evaluating the proposed approaches in term of pattern interpretability. The first dataset JMLR contains 787 abstracts of the Journal of Machine Learning research. JMLR is chosen because the potential important patterns are easily interpreted. The second dataset is a synthetic one with known patterns. For this dataset we evaluate the proposed algorithms based on the accuracy of the set of patterns returned by each algorithms. These datasets along with the source code of the GoKrimp and the SeqKrimp algorithms written in Java are available for download at our project website[2]. Evaluation was done in a 4 x 2.4 Ghz, 4 GB of RAM, Fedora 10 / 64-bit station.

[1]`http://www.timeseriesknowledgemining.org`

[2]`http://www.win.tue.nl/~lamthuy/gokrimp.htm`

In summary, the proposed approaches are evaluated according to the following criteria:

1. *Interpretability* - to informally assess the meaningfulness and redundancy of the patterns.
2. *Run time* - to measure the efficiency of the approaches.
3. *Compression* - to measure how well the data is compressed.
4. *Classification accuracy* - to measure the usefulness of a set of patterns.

8.1 Pattern Interpretability

8.1.1 JMLR Since descriptive pattern mining is unsupervised, it is very hard to compare different sets of patterns in the general case. However, for text data it is possible to interpret the extracted patterns. In this work, we compare different algorithms on the JMLR dataset.

For the GoKrimp algorithm, the significance level used in the sign test is set to 0.01 and the minimum number of pairs needed to perform a sign test is set to 25 as recommended in [35]. For the SeqKrimp algorithm the minimum support was set to 0.1 at which the top 20 patterns returned by each of these algorithm does not change when the minimum support is set smaller. Figure 8 shows the top 20 patterns from the JMLR dataset extracted by the SeqKrimp, the GoKrimp, the SQS and the pGoKrimp algorithm.

Comparing to the top 20 most frequent closed patterns depicted in Figure 1, these sets of patterns are obviously less redundant. The results of the GoKrimp, SeqKrimp and SQS are quite similar. Most of the patterns corresponds to well-known research topics in machine learning.

The pGoKrimp algorithm, i.e. a prior version of the GoKrimp algorithm returns a lots of uninteresting patterns being combinations of frequent events. A possible reason is that in contrast to the SQS and the GoKrimp algorithm, the pGoKrimp algorithm uses an encoding that does not punish gaps and it does not consider the usage of a pattern when assigning codeword to the patterns.

8.1.2 Parallel Parallel is a synthetic dataset which mimics a typical situation in practice where the data stream is generated by five independent parallel processes. Each process P_i generates one event from the set of events $\{A_i, B_i, C_i, D_i, E_i\}$ in that order. In each step, the generator chooses one of five process uniformly at random and generates an event by using that process

Method	Patterns			
	support vector machin	state art	compon analysi	solv problem
	real world	high dimension	sampl size	kernel kernel kernel kernel
SEQKRIMP	machin learn	larg scale	supervis learn	model select
	data set	futur selection	support vector	train set
	bayesian network	experiment result	loss function	loss loss
	support vector machin	state art	neural network	well known
	real world	high dimension	experiment result	special case
GOKRIMP	machin learn	reproduc hilbert space	sampl size	solv problem
	data set	larg scale	supervis learn	signific improv
	bayesian network	independ compon analysi	support vector	object function
	support vector machin	larg scale	featur select	sampl size
	machin learn	nearest neighbor	graphic model	learn algorithm
SEARCH_SQS	state art	decis tree	real world	princip compon analysi
	data set	neural network	high dimension	logist regress
	bayesian network	cross valid	mutual inform	model select
	machin lean learn learn	result show	result result	present algorithm
	algorithm algorithm algorithm algorithm	paper data	machine learn	such data
pGOKRIMP	data data data	problem problem	Perform perform	learn learn
	data set data set	set set	paper propose	show show
	method method method	model model gener	machin kernel kernel	Function function function

Figure 8: Patterns discovered by the SeqKrimp, GoKrimp, SQS and the pGoKrimp algorithm.

until the stream length is 1000000. For this dataset, we know the ground truth since all the sequences containing a mixture of events from different parallel processes are not the right patterns.

We get the first 10 patterns extracted by each algorithm and calculate the precision and recall at K. Precision at K is calculated as the fraction of the number of right patterns in the first K patterns selected by each algorithm. While the recall is measured as the fraction of the number of types of true patterns in the the first K patterns selected be each algorithm. For instance, if the set of the first 10 patterns contains only events from the set $\{A_i, B_i, C_i, D_i, E_i\}$ for a given i then the precision at $K = 10$ is 100% while the recall at $K = 10$ is 20%. The precision measures the accuracy of the set of patterns and the recall measures the diversity of the set of patterns.

For this dataset, the BIDE algorithm was not able to finish its running after a week even if the minimum support was set to 1.0. A reason is that all possible combination of the 25 events are frequent patterns. Therefore, the results of the BIDE and the SeqKrimp algorithm for this dataset are missing. Figure 9 shows the precision and the recall of the set of K patterns returned by the three algorithms SQS, GoKrimp and pGoKrimp when K (x-axis) is varied.

In term of precision all the algorithms are good because the top patterns selected by each of them are all correct ones. However, in term of recall the SQS algorithm is worse than the other two algorithms. A possible explanation is that the SQS algorithm uses an encoding that does not allow encoding interleaving patterns. For this particular dataset where interleaved patterns are observed frequently the SQS algorithm misses patterns that are interleaved with the chosen patterns.

8.2 Running Time We perform experiments to compare running time of different algorithms. For the SeqKrimp algorithm and the BIDE algorithm, we first fix the minimum support parameter to the smallest values used in the experiment where patterns are used as features for classification tasks in subsection 8.3. The SQS algorithm are parameter-free while the GoKrimp algorithm uses standard parameter setting recommended for *Sign Test* so their running time only depends upon the size of the data.

The experimental result is illustrated in Figure 10. As we can see in this figure, the SeqKrimp algorithm is always slower than the BIDE algorithm because it needs an extra procedure to select compressing patterns from the set of candidates returned by the BIDE algorithm. The GoKrimp algorithm is 1-2 orders of magnitude faster than SeqKrimp or the BIDE algorithms, giving results “to go” when in a hurry. The SQS algorithm is very fast on small datasets (though still slower than GoKrimp), however, it is several times slower than the other algorithms on larger datasets such as the unix and

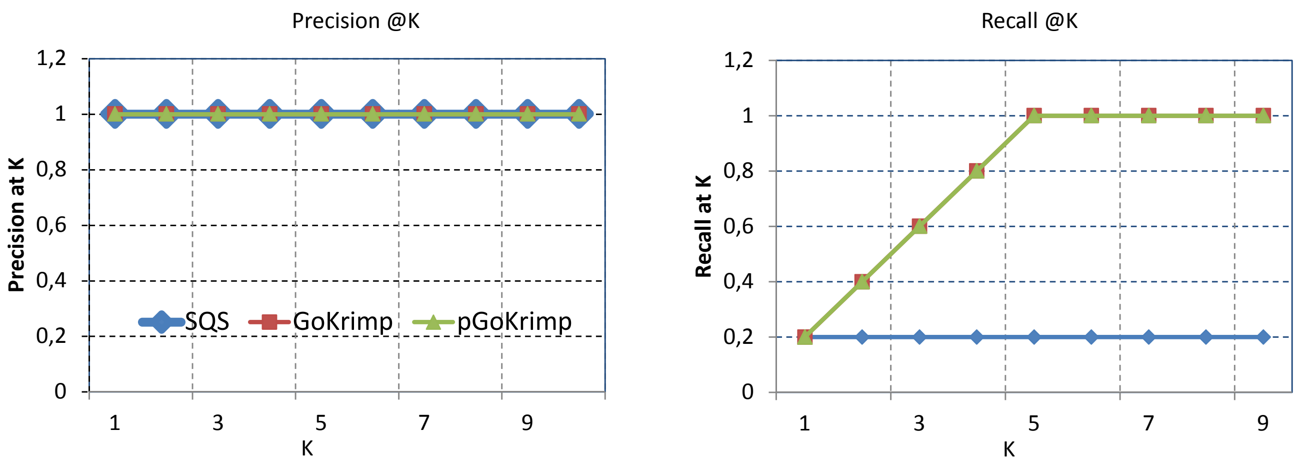


Figure 9: Precision and recall at K of the patterns discovered by the GoKrimp, SQS and the pGoKrimp algorithm in the *Parallel* dataset.

Run time in seconds

Datasets	Bide	SeqKrimp	SQS	GoKrimp
auslan2	0.85	1.0	1.0	0.40
aslbu	74.3	972	277	28
aslgt	73.7	1344	58501	1842
pioneer	11.4	65	15	9
skating	67.3	183	123	85
context	309	402	86	44
unix	1055	47111	84869	1824
jmlr	10	232	890	93
parallel	U/N	U/N	2066	342

The number of patterns

Datasets	Bide	SeqKrimp	SQS	GoKrimp
auslan2	128	4	13	4
aslbu	14620	52	195	67
aslgt	3472	56	1095	68
pioneer	5475	21	143	49
skating	3767	24	140	49
context	6760	15	138	33
unix	28477	75	1070	165
jmlr	4240	23	580	30
parallel	U/N	U/N	17	23

Figure 10: Running time in seconds and the number of patterns returned by each algorithm on nine datasets

the aslgt.

Figure 10 also reports the number of patterns returned by each of algorithms. The BIDE algorithm as usual returns a lot of patterns depending on the minimum support parameter. When this parameter is set low the number of patterns returned by the BIDE algorithm is even larger than the size of the datasets. On the other hand, the SeqKrimp, the SQS and the GoKrimp algorithm returned just a few patterns. The total number of patterns seems to be dependent only on the size of the datasets.

8.3 Classification Accuracy Classification is one of the most important applications of pattern mining algorithms. In this subsection, we discuss results of using the extracted patterns, together with all singletons, as binary attributes for classification tasks. We'll refer to the approach of using only singletons as features as *Singletons*. This algorithm together with the BIDE algorithm are considered as baseline approaches in our comparison.

We use the implementations of classification algorithms available in the Weka package[3]. All the parameters are set to default values. The classification results were obtained by averaging the classification accuracy over 10 folds cross-validations. In the experiments, there are two important parameters: the minimum support value for the BIDE and the SeqKrimp algorithm, and the classification algorithm used to build the classifiers.

Therefore, we perform two different experiments to evaluate the proposed approaches when these parameters are varied. In the first experiment, the minimum supports were set to the smallest values reported in Figure 12. At first, the parameter K is set to infinite to get as many patterns as possible. In doing so, we ob-

[3]http://www.cs.waikato.ac.nz/ml/weka/

Data	Algorithms	Naïve Bayes	Random Forest	J48	VFI	Linear SVM	RBF SVM	Kstar	IB1	Best
auslan2	BIDE	22.50	**29.00**	25.50	22.50	26.50	23.50	25.50	25.50	29.00
	SEQKRIMP	22.00	**30.50**	26.50	24.50	28.00	22.50	24.00	27.00	**30.50**
	GOKRIMP	20.50	29.00	26.00	24.00	**29.50**	23.50	26.00	26.00	29.50
	SINGLETONS	22.00	**29.00**	27.00	23.50	**29.00**	22.00	25.00	26.00	29.00
aslbu	BIDE	48.07	58.27	50.56	31.06	59.18	50.34	**59.41**	59.18	59.41
	SEQKRIMP	52.15	**60.31**	51.02	26.98	59.86	52.07	59.18	57.59	**60.31**
	GOKRIMP	52.38	54.87	50.34	24.26	**59.86**	53.28	59.18	58.27	59.86
	SINGLETONS	51.24	54.89	50.79	25.17	58.50	51.24	**59.64**	57.14	59.64
aslgt	BIDE	70.25	75.06	69.91	54.33	**81.82**	79.38	74.03	73.08	81.82
	SEQKRIMP	72.23	73.35	69.96	56.94	**82.27**	79.64	75.23	73.71	**82.27**
	GOKRIMP	72.17	76.35	70.59	56.65	**81.90**	80.36	76.00	74.83	81.90
	SINGLETONS	71.68	76.92	69.82	57.05	**81.04**	79.38	75.63	73.94	81.04
pioneer	BIDE	96.87	95.625	94.37	93.75	**99.37**	95.62	98.12	98.75	99.37
	SEQKRIMP	**100.0**	98.75	99.37	93.12	**100.0**	**100.0**	90.37	93.37	**100.0**
	GOKRIMP	**100.0**	99.37	99.37	95.12	**100.0**	**100.0**	99.37	99.37	**100.0**
	SINGLETONS	**100.0**	96.67	99.37	93.75	**100.0**	**100.0**	98.75	99.37	**100.0**
skating	BIDE	60.75	57.73	54.33	50.37	**63.77**	57.33	48.49	47.16	63.77
	SEQKRIMP	73.58	73.58	72.45	66.03	74.15	**74.33**	64.52	61.69	**74.33**
	GOKRIMP	67.54	59.81	62.45	57.92	**67.54**	66.98	53.58	52.64	67.54
	SINGLETONS	61.88	58.67	55.09	51.69	**64.71**	58.67	49.24	61.25	64.71
context	BIDE	**77.50**	70.83	75.00	71.25	74.56	70.41	70.41	61.66	77.50
	SEQKRIMP	**79.58**	72.91	77.91	74.58	76.25	73.75	72.08	65.00	79.58
	GOKRIMP	80.83	75.41	80.00	77.91	**82.08**	78.75	74.58	72.18	**82.08**
	SINGLETONS	**78.75**	68.33	75.41	74.16	76.66	74.16	67.50	61.25	78.75
unix	BIDE	42.15	72.15	71.25	29.08	**74.05**	44.43	67.71	63.36	74.05
	SEQKRIMP	54.80	73.81	72.09	37.48	**74.26**	45.90	70.25	65.85	74.26
	GOKRIMP	54.09	73.88	72.05	37.70	**74.33**	45.87	70.39	65.87	**74.52**
	SINGLETONS	57.77	73.90	72.05	38.06	**74.52**	44.43	70.77	66.35	**74.52**

Figure 11: Classification results with patterns used as binary attributes. The number of patterns used in each algorithm were balanced.

tain sets of patterns with different size and the patterns are ordered decreasingly according to the ranks defined by every algorithm. In order to make the comparison fair enough, the patterns at the end of each pattern set are removed such that all the sets have the same number of patterns being equal to the minimum number of patterns discovered by every algorithm. Moreover, different classifiers are used to evaluate the classification accuracy. This helps us to choose the best classifier for the next experiment.

Figure 11 shows the results of the first experiment. Eight different popular classifiers were chosen for classification. The numbers in each cell show the percentage of correctly classified instances. The last column in this figure summarizes the best result, i.e the highest number in each row. Besides, in each cell of this column, the highest value corresponding to the best classification result in a dataset is also highlighted.

The highlighted numbers in the last column show that the top patterns returned by the SeqKrimp and the GoKrimp algorithm are more predictive than the top patterns returned by the BIDE algorithms. On each dataset either SeqKrimp or GoKrimp achieved the best results. Besides, the highlighted numbers in each row show that the *Linear SVM classifier* is the most appropriate classifier for this type of data because it gives the best results in most of the cases.

In the next experiment, the minimum support parameter was varied to see how classification results change. Because the linear SVM classifier gave the best results in most of the datasets, we choose this classifier for this experiment. Figure 12 shows the results. Because the GoKrimp and Singletons features do not depend on minimum support settings, the results of these algorithms do not change across different minimum support settings and are shown as straight lines.

The results show that, in most of the datasets, adding more patterns to the singleton set gives better classification results. However, the benefit of adding more patterns is very sensitive to the minimum support

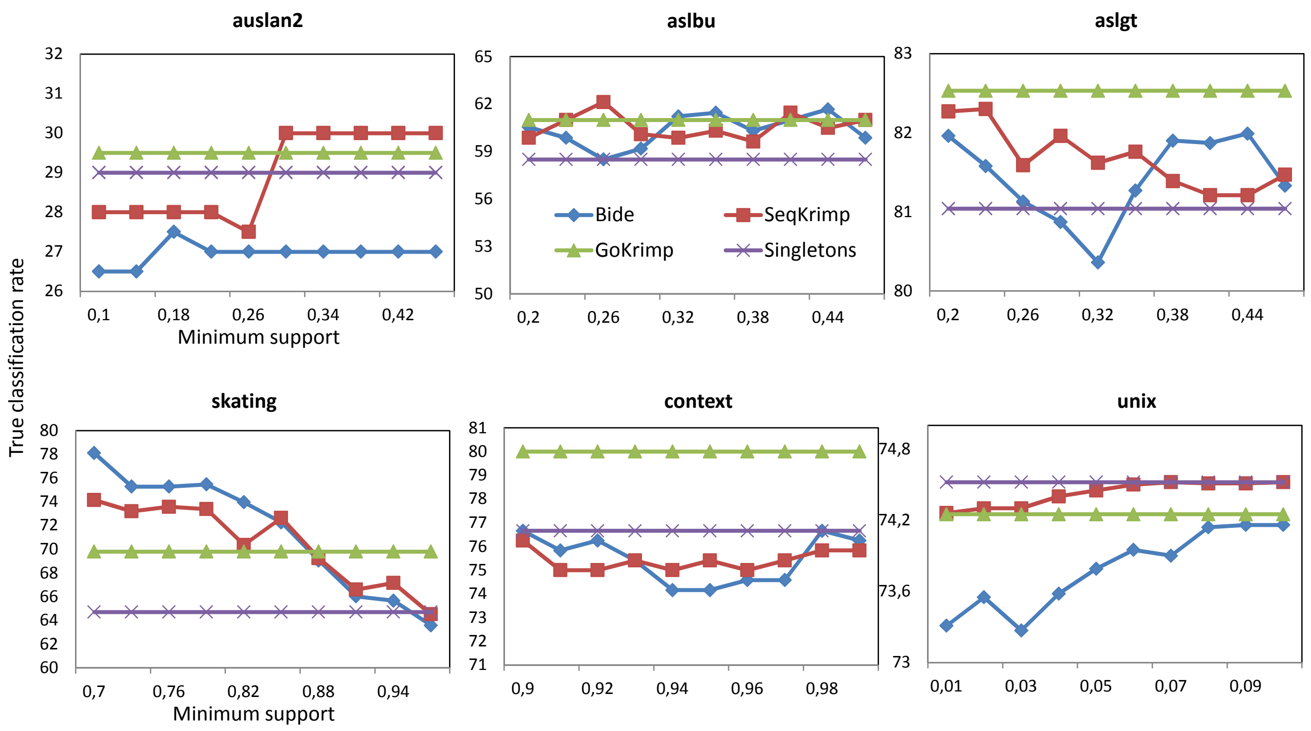


Figure 12: Classification results with linear SVM when using the full set of patterns and varying minimum support.

settings. Especially, it varies significantly from one dataset to another.

Behavior of the BIDE algorithm in particular is very unstable. For example, in the aslgt and the skating datasets adding more patterns, i.e lowering the minimum support, actually improves the classification results of the BIDE algorithm. However, in the auslan2, aslbu, context and unix datasets the effect of adding more patterns is very ambiguous. The behavior of the SeqKrimp algorithm is also very unstable as it uses patterns extracted by BIDE as candidate patterns. Therefore, in these cases, extra effort on parameter tuning is needed.

On the other hand, the classification results of the GoKrimp algorithm do not depend on minimum support. It is better than the singleton approach in most of the cases. It is also much better than the BIDE algorithm in dense datasets such as the context, the aslgt, and the unix data.

8.4 Compressibility

We calculate the compression benefit of the set of patterns returned by every algorithm. In order to make the comparison fair, all sets of patterns have the same size, being equal to the minimum of the number of patterns returned by all algorithms. For the SeqKrimp and the GoKrimp algorithms the compression benefits were calculated as the sum of the compression benefit returned after each greedy step. For Closed patterns, compression benefit was calculated according to the greedy encoding procedure used in the SeqKrimp algorithm. For the SeqKrimp and the BIDE algorithm, the minimum support is fixed to the smallest values in the corresponding experiment shown in Figure 12. Compression benefit is measured as the number of bits saved when encoding the original data using the pattern set as the dictionary. Because the SQS algorithm uses different encoding for data before compression so we cannot compare the compressibility of that algorithm to ours in terms of bits (see below for a comparison by ratios). Figure 13 shows the obtained results in eight different datasets (the result of the algorithm on the parallel dataset is omitted because both SeqKrimp and BIDE did not scale to the size of this dataset). As we expect, in most of the datasets, SeqKrimp and GoKrimp are able to find better compressing patterns than BIDE. Especially, in most of the large datasets such as *aslgt, aslbu, unix, context* and *skating* the differences between SeqKrimp, GoKrimp and BIDE are very significant. The GoKrimp algorithm is able to find compressing patterns with similar quality as the SeqKrimp algorithm in most of the datasets and is even better than the SeqKrimp algorithm in several cases such as in

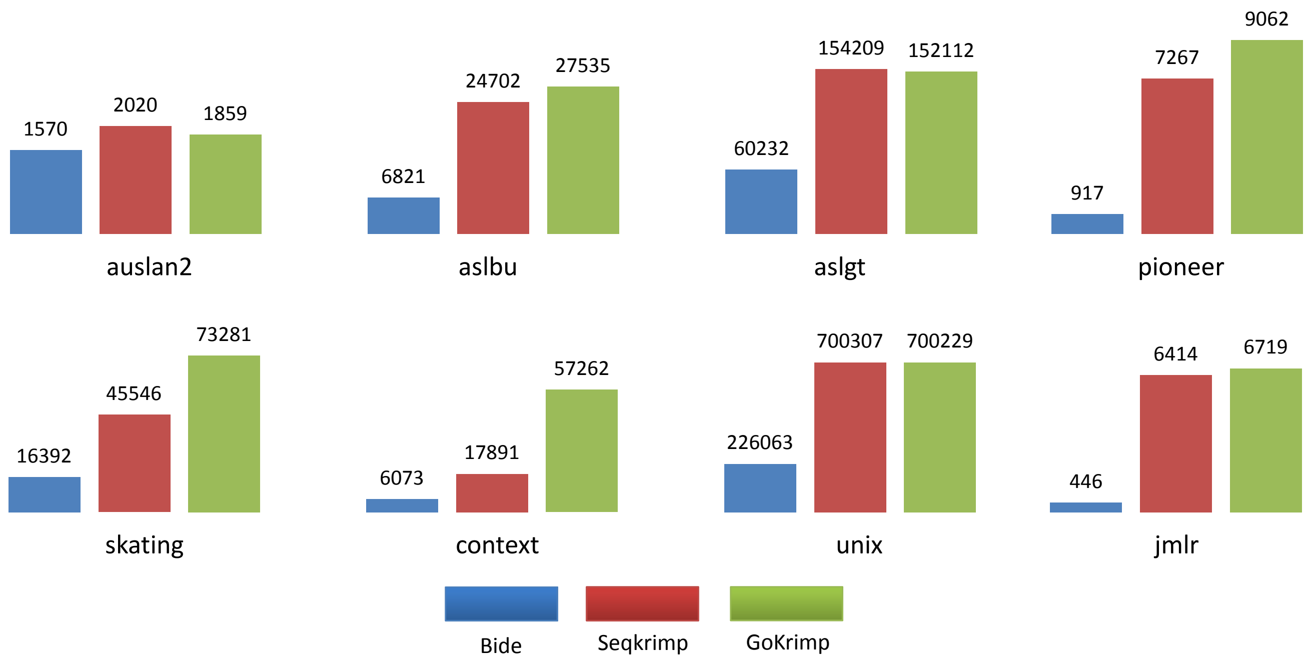


Figure 13: Compression benefit (in number of bits) when using the top patterns selected by each algorithm to compress the data.

	SQS	GoKrimp	GoKrimp*
auslan2	1.571	1.428	1.907
aslbu	1.155	1.123	1.284
aslgt	1.308	1.156	1.450
pioneer	1.302	1.171	1.243
skating	1.880	1.629	2.095
context	2.700	1.706	2.698
unix	2.230	1.638	1.880
jmlr	1.039	1.008	1.008
parallel	1.070	1.135	2.042

Figure 14: Compression ratio comparison of different algorithms

the pioneer, skating and context.

Finally we perform another experiment to compare the GoKrimp algorithm with the SQS algorithm based on compression ratio calculated by dividing the size of the data before compression with the size after compression. It is important to note that the compression ratio is highly dependent how we calculate the size of uncompressed data and how we choose the encoding for gaps. Therefore, in order to make the comparison fair the compression ratios were calculated when using the same uncompressed data representation. However, there is another practical issue of the comparison as follows.

The current implementation of SQS uses an ideal code length for gaps. It calculates the usage of a gap and a non-gap then assigns code length to a gap and a non-gap by considering the entropy of the gap or the non-gap. When the number of non-gaps dominates, which is actual the case in the experiments with our datasets, a non-gap can be assigned a codelength close to zero. This is an ideal case because in practice one cannot assign a codeword with length close to zero. In contrast, GoKrimp uses actual Elias codewords for gaps. Therefore there is a practical issue of comparing two algorithm one use ideal code length and another use actual code length for gaps. Therefore, for GoKrimp we calculate the ideal code length of a gap n as $\log n$, the result of this ideal case will be reported as GoKrimp* in the experiments.

Figure 14 shows the compression ratio of three algorithms on nine datasets. The SQS algorithms show a better compression ratio in most of the case except for the parallel dataset when non-gap is not popular. For that dataset the effect of using ideal codelength is not visible. However, a version of GoKrimp with ideal code length for gaps gives better compression ratios than SQS in most of the cases. These results shows that variation of codeword length calculation can influence the compression ratio significantly. Therefore, interpretation of the results with compression ratios is quite hard in such cases.

8.5 Effectiveness of dependencies test: In this subsection, we perform experiments to demonstrate the effectiveness of the dependency test proposed for speeding up the GoKrimp algorithm. We recall that the dependency test is proposed to avoid exhaustive evaluation of all possible extensions of a pattern. Once a test is done, the results of the test is kept for the next time so in the worst case the maximum number of tests is at most equal to the size of the alphabet. Besides, the set of related events to a given event is quite small compared to the size of the alphabet so the dependency test also helps to reduce the number of extension evaluations.

Figure 15 shows the running time of the GoKrimp algorithm with and without dependency test. It is obvious that the GoKrimp algorithm is much more efficient when dependency testing is used. More importantly, the compression ratio is almost the same in both cases. Therefore the dependency test helps speed up the GoKrimp algorithm significantly while preserving the quality of the pattern set in all the datasets. This result is consistent with an intuition that using patterns with unrelated events for compression does not result in good compression ratios.

9 Conclusions and future work

We have explored mining of sequential patterns that compress the data well utilizing the MDL principle. A key contribution is our encoding scheme targeted at sequence data. We have shown that mining the most compressing pattern set is NP-Hard and designed two algorithms to approach the problem. SeqKrimp is a candidate-based algorithm that turned out to be sensitive to parameter settings and inefficient due to the candidate generation phase. GoKrimp is an algorithm that directly looks for compressing patterns and was shown to be effective and efficient.

The experiments show that the most compressing patterns are less redundant and better than the frequent closed patterns as feature sets for different classifiers. The dependency test technique used in the GoKrimp algorithm was shown to be very useful to speed up the GoKrimp algorithm significantly. Both GoKrimp and SeqKrimp are shown to be effective in finding non-redundant and meaningful patterns. However, the GoKrimp algorithm is 1-2 orders of magnitude faster than the SeqKrimp algorithm and the SQS algorithm.

As is the case on itemset data, compressing patterns are likely to be useful for other data mining tasks where class labels are unavailable or rare, such as change detection or outlier detection. Future work will include further improvements to the mining algorithms using ideas from compression but keeping the focus on usefulness for data mining.

Acknowledgements

This work was done when one of the authors was visiting Siemens Corporate Research, a division of Siemens Corporation, in Princeton, NJ. The work is also funded by the NWO project Mining Complex Pattern in Streams (COMPASS). We would like to thank all the anonymous reviewers for the useful comments which help improving our work significantly.

References

[1] Jilles Vreeken, Matthijs van Leeuwen and Arno Siebes, A. Krimp: Mining Itemsets that Compress. Data Mining and Knowledge Discovery, vol.23(1), Springer, 2011.

[2] Jilles Vreeken Making Pattern Mining Useful. ACM SIGKDD Explorations, vol.12(1), ACM Press, 2010.

[3] Matthijs van Leeuwen, Jilles Vreeken, Arno Siebes. Identifying the components. Data Mining and Knowledge Discovery 19(2): 176-193 (2009)

[4] Matthijs van Leeuwen, Arno Siebes. StreamKrimp: Detecting Change in Data Streams. ECML/PKDD (1) 2008: 672-687

[5] Nikolaj Tatti, Jilles Vreeken. Finding Good Itemsets by Packing Data. ICDM 2008: 588-597

[6] Tijl De Bie. Maximum entropy models and subjective interestingness: an application to tiles in binary databases. DMKD Journal 2011

[7] Tijl De Bie, Kleanthis-Nikolaos Kontonasios, Eirini Spyropoulou. A framework for mining interesting pattern sets. SIGKDD Explorations 12(2): 92-100 (2010)

[8] Jiawei Han. Mining Useful Patterns: My Evolutionary View. Keynote talk at the Mining Useful Patterns workshop KDD 2010

[9] Jianyong and Jiawei Han. BIDE: Efficient mining of frequent closed sequences. In *Proc. of the 20th Intl. Conf. on Data Engineering (ICDE)*, 79–90. IEEE Press, 2004.

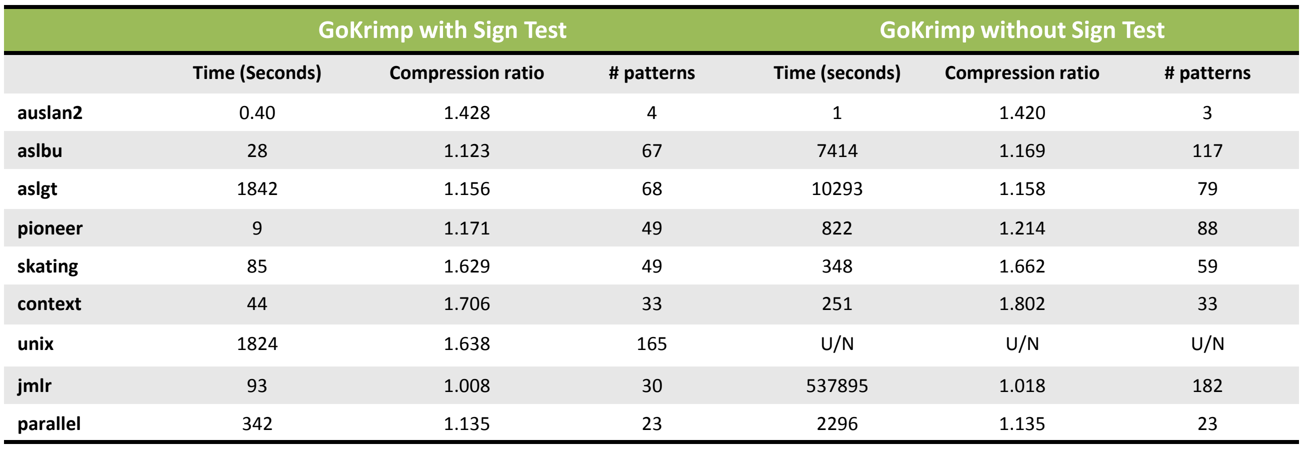
[10] Peter Grünwald. The Minimum Description Length Principle. The MIT Press 2007

[11] Jian Pei, Jiawei Han, Mortazavi-Asl, Jianyong Wang Pinto, Qiming Chen Dayal, Mei-Chun Hsu. Mining sequential patterns by pattern-growth: the PrefixSpan approach. TKDE 2004

[12] Fabian Mörchen. Unsupervised pattern mining from symbolic temporal data. *SIGKDD Explor. Newsl.*, 2007.

[13] Dmitriy Fradkin and Fabian Moerchen. Margin-Closed Frequent Sequential Pattern Mining. Workshop on Mining Useful Patterns. KDD 2010

[14] Fabian Moerchen and Dmitriy Fradkin. Robust mining of time intervals with semi-interval partial order patterns, In Proc. SIAM SDM 2010, pp. 315-326



	GoKrimp with Sign Test			GoKrimp without Sign Test		
	Time (Seconds)	Compression ratio	# patterns	Time (seconds)	Compression ratio	# patterns
auslan2	0.40	1.428	4	1	1.420	3
aslbu	28	1.123	67	7414	1.169	117
aslgt	1842	1.156	68	10293	1.158	79
pioneer	9	1.171	49	822	1.214	88
skating	85	1.629	49	348	1.662	59
context	44	1.706	33	251	1.802	33
unix	1824	1.638	165	U/N	U/N	U/N
jmlr	93	1.008	30	537895	1.018	182
parallel	342	1.135	23	2296	1.135	23

Figure 15: The compression ratios of patterns by the GoKrimp algorithm with and without sign test are almost the same but with sign test the GoKrimp algorithm is much more efficient.

[15] Fabian Moerchen, Thies Michael and Ultsch Alfred: Efficient mining of all margin-closed itemsets with applications in temporal knowledge discovery and classification by compression, Knowledge and Information Systems 2010

[16] Floris Geerts , Bart Goethals , Taneli Mielikainen. Tiling Databases. Discovery Science 2004

[17] Christoph Ambuhl, Monaldo Mastrolilli, and Ola Svensson. Inapproximability Results for Maximum Edge Biclique, Minimum Linear Arrangement, and Sparsest Cut. SIAM J. on Computing volume 40 2011

[18] David Huffman. A Method for the Construction of Minimum-Redundancy Codes. Proceedings of the I.R.E., September 1952, pp 1098-1102.

[19] Aristides Gionis, Heikki Mannila, Taneli Mielikäinen, Panayiotis Tsaparas. Assessing data mining results via swap randomization. TKDD 1(3): (2007)

[20] Ron Milo, Shai Shen-Orr, Shalev Itzkovitz, Nadav Kashtan, Dmitri Chklovskii, Uri Alon. Network Motifs: Simple Building Blocks of Complex Networks. Science, Vol. 298, No. 5594. 2002

[21] Sami Hanhijärvi, Gemma C. Garriga, Kai Puolamäki. Randomization Techniques for Graphs. SDM 2009

[22] Nuno Castro, Paulo Azevedo. Time Series Motifs Statistical Significance. SDM 2011: 687-698

[23] Lawrence Holder, Diane Cook, Surnjani Djoko. Substucture Discovery in the SUBDUE System. KDD Workshop 1994: 169-180

[24] Deepayan Chakrabarti, Spiros Papadimitriou, Dharmendra Modha, Christos Faloutsos: Fully automatic cross-associations. KDD 2004: 79-88

[25] James Storer. Data compression via textual substitution Journal of the ACM (JACM) 1982

[26] Eamonn Keogh, Stefano Lonardi, Chotirat Ann Ratanamahatana, Li Wei, Sang-Hee Lee, John Handley. Compression-based data mining of sequential data. Data Mining Knowledge Discovery 14(1) (2007)

[27] Christos Faloutsos, Vasileios Megalooikonomou. On data mining, compression, and Kolmogorov complexity. Data Mining Knowledge Discovery 15(1) (2007)

[28] Rudi Cilibrasi and Paul Vitányi, Clustering by compression, IEEE Transaction Information Theory, 51:4 2005

[29] Koen Smets and Jiless Vreeken Slim: Directly Mining Descriptive Patterns. SIAM SDM 2012.

[30] Antti Miettinen, Taneli Mielikainen, Aristide Gionis, Gautam Das and Heikki Mannila; The Discrete Basis Problem Knowledge and Data Engineering, IEEE Transactions on, 2008

[31] Jiless Vreeken and Nikolaj Tatti The Long and the Short of It: Summarizing Event Sequences with Serial Episodes, SIGKDD, ACM 2012

[32] Hoang Thanh Lam, Fabian Moerchen, Dmitriy Fradkin and Toon Calders; Mining Compressing Sequential Patterns, SDM, SIAM 2012

[33] Ian Witten, Alistair Moffat and Timothy Bell Managing Gigabytes: Compressing and Indexing Documents and Images, Morgan Kaufmann 1999

[34] Manfred Warmuth and David Haussler, On the complexity of iterated shuffle *J. Comput. Syst. Sci.*, vol. 28, no. 3, pp. 345–358, 1984.

[35] William Conover(1980). Practical Nonparametric Statistics, 2nd ed. Wiley, New York.