Zips: Mining Compressing Sequential Patterns in Streams

Hoang Thanh Lam  
TU Eindhoven  
Eindhoven, the Netherlands  
t.l.hoang@tue.nl

Toon Calders  
TU Eindhoven  
Eindhoven, the Netherlands  
t.calders@tue.nl

Jie Yang  
TU Eindhoven  
Eindhoven, the Netherlands  
j.yang.1@student.tue.nl

Fabian Mörchen  
Amazon.com Inc  
Seattle, WA, USA  
moerchen@amazon.com

Dmitriy Fradkin  
Siemens Corporate Research  
Princeton NJ, USA  
dmitriy.fradkin@siemens.com

ABSTRACT

We propose a streaming algorithm, based on the minimal description length (MDL) principle, for extracting non-redundant sequential patterns. For static databases, the MDL-based approach that selects patterns based on their capacity to compress data rather than their frequency, was shown to be remarkably effective for extracting meaningful patterns and solving the redundancy issue in frequent itemset and sequence mining. The existing MDL-based algorithms, however, either start from a seed set of frequent patterns, or require multiple passes through the data. As such, the existing approaches scale poorly and are unsuitable for large datasets. Therefore, our main contribution is the proposal of a new, streaming algorithm, called Zips, that does not require a seed set of patterns and requires only one scan over the data. For Zips, we extended the Lempel-Ziv (LZ) compression algorithm in three ways: first, whereas LZ assigns codes uniformly as it builds up its dictionary while scanning the input, Zips assigns codewords according to the usage of the dictionary words; more heavily used words get shorter code-lengths. Secondly, Zips exploits also non-consecutive occurrences of dictionary words for compression. And, third, the well-known space-saving algorithm is used to evict un-promising words from the dictionary. Experiments on one synthetic and two real-world large-scale datasets show that our approach extracts meaningful compressing patterns with similar quality to the state-of-the-art multi-pass algorithms proposed for static databases of sequences. Moreover, our approach scales linearly with the size of data streams while all the existing algorithms do not.

1. INTRODUCTION

Mining frequent patterns is an important research topic in data mining. It has been shown that frequent pattern mining helps finding interesting association rules, or can be useful for classification and clustering tasks when the extracted patterns are used as features [1]. However, in descriptive data mining the pattern frequency is not a reliable measure. In fact, it is often the case that highly frequent patterns are just a combination of very frequent yet independent items.

There are many approaches that address the aforementioned issues in the literature. One of the most successful approaches is based on data compression which looks for the set of patterns that compresses the data most. The main idea is based on the Minimum Description Length Principle (MDL) [5] stating that the best model describing data is the one that together with the description of the model, it compresses the data most. The MDL principle has been successfully applied to solve the redundancy issue in pattern mining and to return meaningful patterns [4, 6].

So far most of the work focussed on mining compressing patterns from static datasets or from modestly-sized data. In practice, however databases are often very large. In some applications, data instances arrive continuously with high speed in a streaming fashion. In both cases, the algorithms must ideally scale linearly with the data size and be able to quickly handle fast data updates. In the streaming case, the main challenge is that whole data cannot be kept in memory and hence the algorithm has to be single-pass. None of the approaches described in the literature scales up to the arbitrarily large data or obey the single-pass constraint.

In this work, we study the problem of mining compressing sequential patterns in a data stream where events arrive in batches, e.g. like stream of tweets. We first introduce a novel encoding that encodes sequences with the help of patterns. Different from the encodings used in recent work [2, 3, 7], the new encoding is online which enables us to design online algorithms for efficiently mining compressing patterns from a data stream. We prove that there is a simple algorithm using the proposed online encoding scheme and achieving a near optimal compression ratio for data streams generated by an independent and identical distributed source, i.e. the same assumption that guarantees the optimality of the Huffman encoding in the offline case [9].

Subsequently, we formulate the problem of mining compressing patterns from a data stream. Generally, the data compression problem is NP-complete [11]. Under the streaming context with the additional single pass constraint, we propose a heuristic algorithm to solve the problem. The proposed algorithm scales linearly with the size of data. In the experiments with one synthetic and two real-life large-scale datasets, the proposed algorithm was able to extract mean-
ingful patterns from the streams while being much more scalable than the state-of-the-art algorithms.

2. RELATED WORK

The SubDue system [6] is the first work exploiting the MDL principle for mining useful frequent subgraphs. In the field of frequent itemset mining, the well-known Krimp algorithm [4] was shown to be very good at solving the redundancy issue and at finding meaningful patterns. The MDL principle was first applied for mining compressing patterns in sequence data in [2, 3] and in [7]. The GoKrimp algorithm in the former work solved the redundancy issue effectively. However, the first version of the GoKrimp algorithm [2] used an encoding scheme that does not punish large gaps between events in a pattern. In an extended version of the GoKrimp algorithm [3] this issue is solved by introducing gaps into the encoding scheme based on Elias codes. Besides, a dependency test technique is proposed to filter out meaningless patterns. Meanwhile, in the latter work the SQS algorithm proposed a clever encoding scheme punishing large gaps. In doing so, the SQS was able to solve the redundancy issue effectively. At the same time it was able to return meaningful patterns based solely on the MDL principle.

However, a disadvantage of the encoding defined by the SQS algorithm is that it does not allow encoding of overlapping patterns. Situations where patterns in sequences overlap are common in practice, e.g. message logs produced by different independent components of a machine, network logs through a router etc. Moreover, neither the GoKrimp algorithm nor the SQS algorithm were intended for mining logs through a router etc. Moreover, neither the GoKrimp algorithm in the former work solved the redundancy issue effectively. At the same time it was able to return meaningful patterns based solely on the MDL principle.

In this work, we assume that events in a data stream arrive in batches. This assumption covers broad types of data streams such as tweets, web-access sequences, search engine query logs, etc. This section discusses online encodings that compress a data stream by a set of patterns. For education reasons, we first start with the simplest case when only singletons are used to encode the data. The generalized case with non-singletons is described in the next subsection.

3. DATA STREAM ENCODING

In this work, we assume that events in a data stream arrive in batches. This assumption covers broad types of data streams such as tweets, web-access sequences, search engine query logs, etc. This section discusses online encodings that compress a data stream by a set of patterns. For education reasons, we first start with the simplest case when only singletons are used to encode the data. The generalized case with non-singletons is described in the next subsection.

3.1 Online encoding using singletons:

We discuss an online encoding that uses only singletons to compress the data. Since this encoding does not exploit any pattern for compress the data, we consider the representation of the data in this encoding as an uncompressed form of that data. Let $\sum = \{a_1, a_2, \cdots, a_n\}$ be an alphabet containing a set of characters $a_i$, the online data encoding problem can be formulated as follows:

**Definition 1 (Online Encoding Problem).** Let $A$ be a sender and let $B$ be a receiver. $A$ and $B$ communicate over some network, where sending information is expensive. $A$ observes a data stream $S_t = b_1b_2\cdots b_t$. Upon observing a character $b_t$, $A$ needs to compress the character and transmit it to the receiver $B$, who must be able to uncompressed it. Since sending information on the network is expensive, the goal of $A$ and $B$ is to compress the stream as much as possible to save up the network bandwidth.

In the offline scenario, i.e., when $S_t$ is finite and given in advance, one possible solution is to first calculate the frequency of every item $a$ (denoted as $f(a)$) of the alphabet in the sequence $S_t$. Then assign each item $a$ a codeword with length proportional to its entropy, i.e. $−\log f(a)$. It has been shown that when the stream is independent and identically distributed (i.i.d) this encoding, known as the Huffman code in the literature, is optimal [9]. However, in the streaming scenario, the frequency of every item $a$ is unknown and the codeword of $a$ must be assigned at the time $a$ arrives and $B$ must know that codeword to decode the compressed item.

We propose a simple solution for Problem 1 as follows. First, in our proposed encoding we need codewords for natural numbers. This work uses the Elias Delta code [8] denoted $E(n)$ to encode the number $n$. The Elias was chosen because it was provable as near optimal when the upper bound on $n$ is unknown in advance. The length of the codeword $E(n)$ is $\lfloor \log_2 n \rfloor + 2 \lfloor \log_2 (\lfloor \log_2 n \rfloor + 1) \rfloor + 1$ bits.

$A$ first notifies $B$ of the size of the alphabet by sending $E(\lfloor \sum \rfloor)$ to $B$. Then it sends $B$ the dictionary containing all
The reference encoding can be extended to the case using singleton together with non-singleton patterns to encode a data stream. Let $\mathcal{S} = S_1S_2\cdots S_t$ denote a stream of sequences where each $S_i$ is a sequence of events. Let $D$ be a dictionary containing all characters of the alphabet and some non-singletons. **Reference encodings** compress a data stream $\mathcal{S}$ by replacing instances of words in the dictionary by references to the most recent occurrences of the words. If the words are non-singletons, beside the references, gaps between characters of the encoded words must be stored together with the references. Therefore, in a reference encoding, beside the reference stream we also have a gap stream.

Similar to the case with non-singleton, first we need to encode the dictionary $D$ (now contains both singleton and non-singleton). We add a special symbol $\sharp$ to the alphabet. The binary representation of the dictionary starts with the codeword of the size of the dictionary. It is followed by the codewords of all the characters in the alphabet each with length $\lceil \log_2 |D| \rceil$. The representations of every non-singleton follow right after that. The binary representation of a non-singleton contains codewords of the characters of the non-singleton. Non-singletons are separated from each other by the special character $\sharp$.

**Example 2** (Dictionary representation). The dictionary $D = \{a, b, c, \sharp, ab, abc\}$ can be represented as follows $E(6)C(a)C(b)C(c)C(\sharp)C(a)C(b)C(\sharp)C(a)C(b)C(c)$. The representation starts with $E(6)$ indicating the size of $D$. It follows by the codewords of all the characters and the binary representation of the non-singletons separated by $\sharp$.

Having the binary representation of the dictionary, A first sends that representation to B. After that A send the encoding of the actual stream with the reference stream and the gap stream. The following example show how to encode a sequence with a reference encoding.

**Example 3** (Reference encoding). Given a dictionary $D = \{a, b, c, \sharp, ab, abc\}$, a sequence $S = abbcabcaabcbac$. Figure 2 show an encoding of $S$ using dictionary words (the numbers below the characters denote the positions of the character in the original sequence). The reference stream is $E(1)E(7)E(5)E(8)E(2)E(2)E(2)E(2)E(8)$, where $E(1)$ is the reference of the first abc to the position of abc in the
dictionary, \( E(7) \) is the reference of the following \( b \) to the position of \( b \) in the dictionary and so on. The gap stream is \( E(1)E(2)E(2)E(2)E(1)E(1) \), where for instance, the first codewords \( E(1)E(2) \) indicate the gaps between \( a \) and \( b \), \( b \) and \( c \) in the first occurrence of \( abc \) at position \( 1, 2, 4 \) in the stream. For non-singleton, there is no gap information representing in the gap stream.

**Example 4 (Decoding).** In Figure 2, the sequence can be decoded as follows. Reading the first codeword of the reference stream, i.e. \( E(1) \), the decoder refers one step back to get \( abc \). There will be two gaps (between \( a, b \) and \( b, c \)) so the decoder reads the next two codewords from the gap stream, i.e. \( E(1) \) and \( E(2) \). Knowing the gaps it can infer the positions of \( abc \). In this case, the positions are \( 1, 2 \) and \( 4 \). Subsequently, the decoder reads the next codeword from the reference stream, i.e. \( E(7) \), it refers seven steps back and decode the current reference as \( b \). There is no gap because the word is a singleton, the position of \( b \) in the stream corresponds to the earliest position that has not been occupied by any decoded character, i.e. \( 3 \). The decoder continues decode the other references of the stream in the same way.

Different from the singleton case, there might be a lot of different reference encodings for a stream given a dictionary. Each reference encoding incurs different description lengths. Finding an optimal dictionary and an optimal reference encoding is the main problem we solve in this paper.

### 4. Problem Definition

Given a data stream \( \mathcal{S} \) and a dictionary \( D \) denote \( L_D^\mathcal{S} \) as the description length of the data (including the cost to store the dictionary) in the encoding \( C \). The problem of mining compressing sequential patterns in data stream can be formulated as follows:

**Definition 2 (Compressing patterns mining).** Given a stream of sequences \( \mathcal{S} \), find a dictionary \( D \) and an encoding \( C \) such that \( L_D^\mathcal{S} \) is minimized.

Generally, the problem of finding the optimal lossless compressed form of a sequence is NP-complete [11]. In this work, Problem 2 is similar to the data compression problem but with additional constraint on the number of passes through data. Therefore, in next section we discuss a heuristic algorithm inspired by the idea of the Lempel-Ziv’s data compression algorithm [9] to solve this problem.

### 5. Algorithms

In this section, we discuss an algorithm for finding a good set of compressing patterns from a data stream. Our algorithm is single-pass, memory-efficient and scalable. We call our algorithm Zips as for Zip a stream. There are three important subproblems that Zips will solve. The first problem concerns how to grow a dictionary of promising candidate patterns for encoding the stream. Since the memory is limited, the second problem is how to keep a small set of important candidates and evict from the dictionary unpromising candidates. The last problem is that having a dictionary how to encode the next sequence effectively with existing words in the dictionary.

The pseudo-code depicted in Algorithm 1 shows how Zips work. It has three subroutines each of them solves one of the three subproblems:

1. Compress a sequence given a dictionary: for every new sequence \( S_t \) in the stream, Zips uses the subroutine \( encode(S_t) \) to find the word \( w \) in the dictionary which gives the most compression benefit when it is used to

![Figure 2: An example of encoding the sequence \( S = abbcababcabcabcabcabcabcabcabcabcabcabc \) with a reference encoding. The arrows represent the references between two consecutive encoded occurrences of a word which represent as a reference stream \( E(1)E(7)E(5)E(8)E(2)E(2)E(2)E(2)E(2)E(2)E(8) \). Having the reference stream we can reconstruct the stream but not in the same order of event occurrence. Therefore, we need a gap stream indicating the gaps between consecutive characters in each encoded non-singletons. For example, the gap stream is \( E(1)E(2)E(2)E(2)E(2)E(1)E(1) \).](image)
encode the unencoded part of the sequence. Subsequently, the instance corresponds to the best chosen word is removed from \( S_1 \). Detail about how to choose \( w \) is discussed in subsection 5.1.

2. Grow a dictionary: Zips uses \( \text{extend}(w) \) to extend the word \( w \) chosen by the procedure \( \text{encode}(S_1) \). Word extensions are discussed in subsection 5.2.

3. Dictionary update: the new extension is added to the dictionary. When the dictionary size exceeds the memory limit, a space-saving algorithm is used to evict unpromising words from the dictionary (subsection 5.3).

These steps are iteratively repeated as long as \( S_1 \) is not encoded completely. When compression of the sequence finishes, Zips continues to compress the next in a similar way.

### 5.1 Compress a sequence:

Let \( S \) be a sequence, consider a dictionary word \( w = a_1a_2 \cdots a_k \), let \( S(w) \) denote an instance of \( w \) in \( S \). Let \( g_1, g_2, \ldots, g_k \) be the gaps between the consecutive characters in \( S(w) \). We denote the gap between the current occurrence and the most recent encoded occurrence of \( w \) by \( g \). Let \( i = 1, \ldots, k \) be the gap between the current and the most recent occurrence of \( a_i \). Therefore, to calculate the compression benefit we subtract the size of encoding \( S(w) \) and the cost of encoding the gap to the last encoded occurrence of \( S(w) \) from the size of encoding each singleton:

\[
B(S(w)) = \sum_{i=1}^{k} |E(g_i)| - |E(g)| - \sum_{i=2}^{k} |E(g_i)|
\]

**Example 5 (Compress a sequence).** Figure 3.a shows a sequence \( S \) in the uncompressed form and Figure 3.b shows the current form of \( S \). Assume that the instance of \( w = abc \) at positions 1, 2, 4 is already compressed. Consider two instances of \( abc \) in the unencoded part of \( S \):

1. \( S^1(w) = (a, 3)(b, 5)(c, 7) \): the cost to replace this instance by a pointer is equal to the sum of the cost to encode the reference to the previous encoded instance of \( abc \) plus the cost of gaps \( |E(2)| + |E(2)| + |E(2)| \). The cost of representing this instance in an uncompressed form is \( |E(2)| + |E(3)| + |E(3)| \). Therefore, the compression benefit of using this instance to encode the sequence is \( B(S^1(w)) = |E(2)| + |E(3)| + |E(3)| - |E(2)| - |E(2)| - |E(2)| = 3 \) bits.

2. \( S^2(w) = (a, 3)(b, 6)(c, 8) \): the compression benefit of using \( S^2(w) \) to encode the sequence is calculated in a similar way: \( B(S^2(w)) = |E(2)| + |E(1)| + |E(1)| - |E(1)| - |E(3)| - |E(2)| - |E(2)| = -3 \) bits.

In order to ensure that every symbol of the sequence is encoded, the next instance considered for encoding has to start at the first non-encoded symbol of the sequence. There may be many instances of \( w \) in \( S \) that start with the first non-encoded symbol of \( S \), denote \( S^*(w) = \arg\max B(S(w)) \) as the one that results in the maximum compression benefit. We call \( S^*(w) \) the best match of \( w \) in \( S \). Given a dictionary, the encoding function depicted in Algorithm 2 first goes through the dictionary and finds the best match starting at the next uncompressed character of every dictionary word in the sequence \( S \) (line 4). Among all the best matches, it greedily chooses the one that results in the maximum compression benefit (line 6).

For any given dictionary word \( w = a_1a_2 \cdots a_k \), the most important subroutine of Algorithm 2 is to find the best match \( S^*(w) \). This problem can be solved by creating a directed acyclic graph \( G(V, E) \) as follows:

1. Initially, \( V \) contains a start node \( s \) and an end node \( e \)
2. For every occurrence of \( a_i \) at position \( p \) in \( S \), add a vertex \( (a_i, p) \) to \( V \)
3. Connect \( s \) with the node \( (a_i, p) \) by a directed edge and add to that edge a weight value equal to \( |E(\pi_i)| - |E(g)| \)
4. Connect every vertex \( (a_i, p) \) with \( e \) by a directed edge with weight 0
5. For all \( q > p \) connect \( (a_i, p) \) to \( (a_{i+1}, q) \) by a directed edge with weight value \( |E(\pi_{i+1})| - |E(g - p)| \)

**Algorithm 2 encode(S)**

1: **Input**: a sequence \( S \) and dictionary \( D = w_1w_2 \cdots w_N \)
2: **Output**: the word \( w \) that starts at the first non-encoded symbol gives the most additional compression benefit
3: for \( i = 1 \) to \( N \) do
4: \( S^*(w_i) = \text{bestmatch}(S, w_i) \)
5: end for
6: \( max = \arg\max_i B(S^*(w_i)) \)
7: Return \( w_{max} \)
The best match $S^*(w)$ corresponds to the directed path from $s$ to $e$ with the maximum sum of the weight values along the path.

The proof of theorem 2 is trivial since any instance of $w$ in $S$ corresponds to a directed path in the directed acyclic graph and vice versa. The sum of the weights along a directed path is equal to the benefit of using the corresponding instance of $w$ to encode the sequence. Finding the directed path with maximum weight sum in a directed graph is a well-known problem in graph theory. That problem can be solved by a simple dynamic programming algorithm in linear time of the size of the graph, i.e. $O(|S|^2)[12]$.

Example 6 (The best match in a graph). Figure 4 shows the directed acyclic graph created from the instances of $abc$ in the uncompressed part of $S$ shown in Figure 3b. The best match of $abc$ corresponds to the path $s(a, 3)(b, 5)(c, 7)e$ with the maximum sum of weights equal to 3 bits.

It is important to notice that in order to evaluate the compression benefit of a dictionary word, Equation 1 only requires bookkeeping the position of the most recent encoded instance of the word. This is in contrast to the offline encodings used in recent work [2, 3, 7] in which the bookkeeping of the word usage and the gaps cost is a must. When a new instance of a word is replaced by a pointer, the relative usage and the codewords of all dictionary words also change. As a result, the compression benefit needs to be recalculated by a pass through the dictionary. This operation is an expensive task when the dictionary size is unbounded.

5.2 Dictionary extension:

Initially, the dictionary contains all singletons; it is iteratively expanded with the locally best words. In each step, when the best match of a dictionary word has been found, Zips extends the best match with one extra character and adds this extension to the dictionary. There are different options to choose the character for extension. In this work, Zips chooses the next uncompressed character right after the word. The choice is inspired by the same extension method suggested by the Lempel-Ziv compression algorithms.

Moreover, there is another reason behind our choice. When $w$ is encoded for the first time, the reference to the previous encoded instance of $w$ is undefined although the word has been already added to the dictionary. Under that circumstance, we have to differentiate between two cases: either a reference to an extension or to an encoded word. To achieve this goal one extra flag bit is added to every reference. When the flag bit is equal to 1, the reference refers to an extension of an encoded word. Otherwise, it refers to an encoded word. When the former case happens by extending the word with the character right after it, the decoder always knows where to find the last character of the extension. When a word is added to a dictionary all of its prefixes have been already added to the dictionary. This property enables us to store the dictionary by using a prefix tree.

Example 7. Figure 5 shows the first moment $t_0$ when the word $w = abc$ is added to the dictionary and two other moments $t_1$ and $t_2$ when it is used to encode the stream. At $t_1$, the flag bit 1 is used to notify the decoder that the gap $g_1$ is a reference to an extension of an encoded word, while at $t_2$ the decoder understands that the gap $g_2$ is a reference to the previous encoded instance of $w$.

References to extensions may cause ambiguous references. For instance, a potential case of ambiguous reference called reference loop is discussed in the following example:

Example 8 (Reference loops). At time point to the word $w = ab$ at positions $p_1p_2$ is extended by $c$ at position $p_3$. Later on at another time point $t_1 > t_0$, another instance of $abc$ at $q_1q_2q_3$ ($q_1 > p_1$) refers back to the instance $abc$ at $p_1p_2p_3$. At time point $t_2 > t_1$, another instance of $abc$ at $r_1r_2r_3$ ($r_1 > q_1$) refers back to the instance $abc$ at $q_1q_2q_3$. In this case, $c$ at $p_3$ and $c$ at $q_3$ refers to each other forming a reference loop.

Reference loops result in ambiguity when we decode the sequence. Therefore, when we look for the next best matches, in order to avoid reference loops, we always check if the new match incurs a loop. The match that incurs a loop is not considered for encoding the sequence. Checks for ambiguity can be done efficiently by creating paths of references. Vertices of a reference path are events in the sequence. Edge between two consecutive vertices of the path corresponds to a reference to the associated events. Since the total sizes of all the paths is at most the sequence length, its is cheap to store the paths for a bounded size sequence. Moreover, updating and checking if a path is a loop can be done in $O(1)$ if vertices of the path are stored in a hashmap.

Example 9 (Reference paths). The references paths of the encoding in Example 8 are: $(a, r_1) \mapsto (a, q_1) \mapsto (a, p_1)$, $(b, r_2) \mapsto (b, q_2) \mapsto (b, p_2)$ and $(c, p_3) \mapsto (c, q_3) \mapsto (c, p_3)$. The last path is a loop.

5.3 Dictionary update:

New extensions are added to the dictionary until the dictionary exceeds memory limit. When it happens the space-saving algorithm is used to evict unpromising words from the dictionary. The space-saving algorithm [10] is a well-known

![Figure 4](image1.png)

Figure 4: A directed acyclic graph created from the instances of $abc$ in the uncompressed part of $S$ shown in Figure 3b.

![Figure 5](image2.png)

Figure 5: An example of word is extended and encoded the first time and the second time. One extra flag bit is added to every reference to differentiate two cases.
The algorithm works as follows, for every non-singleton word \( w \) it maintains a counter with two fields. The first field denoted as \( w[1] \) contains an over-estimate of the compression benefit of \( w \). The second field denoted as \( w[2] \) contains the compression benefit of the word with least compression benefit in the dictionary at the moment that \( w \) is inserted into the dictionary.

Every time when a word \( w \) is chosen by Algorithm 2 to encode its best match in the sequence \( S_t \), the compression benefit of the word is updated. The word \( w \) is then extended to \( w^* \) with an extra character by the extension subroutine. In its turn, Algorithm 3 checks if the dictionary already contains \( w^* \). If the dictionary does not contain \( w^* \) and it is full with \( M \) non-singleton words, the least compressing word \( v \) resident at a leaf of the dictionary prefix-tree is removed from the tree. Subsequently, the word \( w^* \) is inserted into the tree and its compression benefit can be over-estimated as \( w[1] = w[2] = v[1] \). The first counter of every word is always greater than the true compression benefit of the word.

The number of sequences Tweets running time

Figure 6: The running time and the average update time per sequence of the Zips algorithm in the Tweets dataset when the stream size increases. Zips scales linearly with the size of the stream.

words. This property ensures that the new emerging word is not removed very quickly because its accumulated compression benefit is dominated by long lasting words in the dictionary. For any word \( w \), the difference between \( w[2] \) and \( w[1] \) is the actual compression benefit of \( w \) since the moment that \( w \) is inserted into the dictionary. At anytime point when we need to find the most compressing patterns, we compare the value of \( w[2] - w[1] \) and select those with highest \( w[2] - w[1] \). In section 6 we show empirical results with different datasets that this algorithm is very effective in finding the most compressing patterns with high accuracy even with limited memory.

6. EXPERIMENTS

We perform experiments with one synthetic dataset with known ground truth and two real-world datasets. Our implementation of the Zips algorithm in C++ together with the datasets are available for download at our project website\(^2\). All the experiments were carried out on a machine

\(^2\)www.win.tue.nl/~lamthuy/zips.html

Algorithm 3 update(\( w^* \))

1: **Input:** a word \( w^* \) and dictionary \( D = \{w_1, w_2, \ldots, w_N \} \)
2: **Output:** the dictionary \( D \)
3: \( m \leftarrow |v : w_i \text{ is a non-singleton}| \)
4: \( v = \arg\min w_i[1] \) and \( v \) is non-singleton at a leaf of the prefix-tree
5: if \( m > M \) and \( w^* \not\in D \) then
6: \( D = D \setminus \{v\} \)
7: \( w^*[1] = w^*[2] = v[1] \)
8: \( D = D \cup \{w^*\} \)
9: else if \( w^* \not\in D \) then
10: \( w^*[1] = w^*[2] = 0 \)
11: \( D = D \cup \{w^*\} \)
12: else
13: add additional compression benefit to \( w^*[1] \)
14: end if
15: Return \( D \)
with 16 processor cores, 2 Ghz, 12 GB memory, 194 GB local disk, Fedora 14 / 64-bit. As baseline algorithms, we choose the GoKrimp algorithm [2, 3] and the SQS algorithm [7] for comparison in terms of running time, scalability, and interpretability of the set of patterns.

6.1 Data
Three different datasets are:

1. JMLR: contains 787 abstracts of articles in the Journal of Machine Learning Research. English words are stemmed and stop words are removed. JMLR is small but it is considered as a benchmark dataset in the recent work [2, 3, 7]. The dataset is chosen also because the set of extracted patterns can be easily interpreted.

2. Tweets: contains over 1270000 tweets from 1250 different twitter accounts. All tweets are ordered ascending by timestamp. English words are stemmed and stop words are removed. After preprocessing, the dataset contains over 900000 tweets. Similar to the JMLR dataset, this dataset is chosen because the set of extracted patterns can be easily interpreted.

3. Plant: is a synthetic dataset generated in the same way as the generation of the plant10 and plant50 dataset used in [7]. The plant10 and plant50 are small so we generate a larger one with ten patterns each with 5 events occurs 100 times at random positions in a sequence with length 100000 generated by 1000 independent noise event types.

6.2 Running time and Scalability
Figure 6 plots the running time and the average update time per sequence of the Zips algorithm in the Tweets dataset when the data stream size (the number of sequences) increases. Three different lines correspond to different maximum dictionary size settings $M = 1000$, $M = 5000$ and $M = 10000$ respectively. The results show that the Zips algorithm scales linearly with the size of the stream. The average update time per sequence is constant given a maximum dictionary size setting. For example, when $M = 10000$, Zips can handle one update in about 20-100 milliseconds.

Figure 7 shows the running time in y-axis of the Zips algorithm against the stream size in x-axis in the Tweets dataset when the maximum dictionary size is set to 1000. In the same figure, the running time of the baseline algorithms GoKrimp and SQS are also shown. There are some missing points in the results corresponding to the SQS algorithm because we set a deadline of ten hours to get the results corresponding to a point. The missing points corresponding to the cases when the SQS program did not finish in time.

In the log-log scale, running time lines resemble straight lines. This result shows that the running time of Zips, GoKrimp and SQS is the power of data size, i.e. $T \sim \alpha |S|^\beta$. Using linear fitting functions in log-log scale we found that with the Zips algorithm $\beta = 1.01$, i.e. Zips scales linearly with the data size. Meanwhile, for the SQS algorithm the exponent is $\beta = 2.2$ and for the GoKrimp algorithm the exponent is $\beta = 2.01$. Therefore, both GoKrimp and SQS do not scale linearly with the data size and hence they are not suitable for data stream applications.

6.3 JMLR
In Figure 8, we show the first 20 patterns extracted by two baseline algorithms GoKrimp and SQS and the Zips algorithm from the JMLR dataset. Three lists are slightly different but the important patterns such as “support vector machine”, “data set”, “machine learn”, “bayesian network” or “state art” were discovered by all of the three algorithms. This experiment confirms that the Zips algorithm was able to find important patterns that are consistent with the results of state-of-the-art algorithms.

6.4 Tweets
Since the tweet dataset is large, we schedule the programs so that they terminate their running after two weeks. The SQS algorithm was not able to finish its running before the deadline while GoKrimp finished running after three days and Zips finished running after 35 hours. The set of patterns extracted by the Zips algorithm and the GoKrimp algorithm are shown in Figure 9. Patterns are visualized by the wordcloud tool in R such that more important patterns are represented as larger words. In both algorithms, the sets of patterns are very similar. The result shows the daily discussions of the 1250 twitter accounts about the topics regarding “social media”, “Blog post”, about “youtube video”, about “iphone apps”, about greetings such as “happy birthday”, “good morning” and “good night”, about custom service complaint etc.

6.5 Plants
It has been shown [7] that SQS successfully returned all ten patterns. We obtain the same result with GoKrimp and Zips. All three algorithm ranked 10 true patterns with highest scores.

7. CONCLUSIONS AND FUTURE WORK
In this paper we studied the problem of mining compressing patterns from a data stream. A new encoding scheme for sequence is proposed. The new encoding is convenient for streaming applications because it allows encoding the data in an online manner. Because the problem of mining the best set of patterns with respect to the given encoding is shown to be unsolvable under the streaming context, we propose a heuristic solution that solves the mining compressing problem effectively. In the experiments with one synthetic dataset with known ground-truths Zips was able to extract the most compressing patterns with high accuracy. Meanwhile, in the experiments with two real-world datasets it can find patterns that are similar to the state-of-the-art algorithms extract from these datasets. More importantly, the proposed algorithm was able to scale linearly with the size of the stream while the state-of-the-art algorithms were not. There are several options to extend the current work. One of the most promising future work is to study the problem of mining compressing patterns for different kinds of data stream such as a stream of graphs.

8. REFERENCES
[1] Hong Cheng, XiFeng Yan, Jiawei Han, Philip S. Yu: Direct Discriminative Pattern Mining for Effective Classification. ICDE 2008: 169-178

\[ \text{http://user.informatik.uni-goettingen.de/~txu/cuckoo/dataset.html} \]
<table>
<thead>
<tr>
<th>Method</th>
<th>Patterns</th>
</tr>
</thead>
<tbody>
<tr>
<td>SOS</td>
<td>support vector machine</td>
</tr>
<tr>
<td></td>
<td>learning</td>
</tr>
<tr>
<td></td>
<td>state art</td>
</tr>
<tr>
<td></td>
<td>data set</td>
</tr>
<tr>
<td></td>
<td>bayesian network</td>
</tr>
<tr>
<td></td>
<td>larg scale</td>
</tr>
<tr>
<td></td>
<td>nearest neighbor</td>
</tr>
<tr>
<td></td>
<td>decision tree</td>
</tr>
<tr>
<td></td>
<td>neural network</td>
</tr>
<tr>
<td></td>
<td>cross validation</td>
</tr>
<tr>
<td></td>
<td>feature selection</td>
</tr>
<tr>
<td></td>
<td>graphical model</td>
</tr>
<tr>
<td></td>
<td>real world</td>
</tr>
<tr>
<td></td>
<td>high dimension</td>
</tr>
<tr>
<td></td>
<td>mutual information</td>
</tr>
<tr>
<td></td>
<td>sample size</td>
</tr>
<tr>
<td></td>
<td>learning algorithm</td>
</tr>
<tr>
<td></td>
<td>principal component analysis</td>
</tr>
<tr>
<td></td>
<td>logistic regression</td>
</tr>
<tr>
<td></td>
<td>model selection</td>
</tr>
<tr>
<td>GOKRIMP</td>
<td>support vector machine</td>
</tr>
<tr>
<td></td>
<td>real world</td>
</tr>
<tr>
<td></td>
<td>learning</td>
</tr>
<tr>
<td></td>
<td>data set</td>
</tr>
<tr>
<td></td>
<td>bayesian network</td>
</tr>
<tr>
<td></td>
<td>state art</td>
</tr>
<tr>
<td></td>
<td>larg scale</td>
</tr>
<tr>
<td></td>
<td>reproducing hilbert space</td>
</tr>
<tr>
<td></td>
<td>independ component analysis</td>
</tr>
<tr>
<td></td>
<td>neural network</td>
</tr>
<tr>
<td></td>
<td>experiment result</td>
</tr>
<tr>
<td></td>
<td>sample size</td>
</tr>
<tr>
<td></td>
<td>supervised learning</td>
</tr>
<tr>
<td></td>
<td>support vector</td>
</tr>
<tr>
<td></td>
<td>well known</td>
</tr>
<tr>
<td></td>
<td>special case</td>
</tr>
<tr>
<td></td>
<td>solving problem</td>
</tr>
<tr>
<td></td>
<td>significant improvement</td>
</tr>
<tr>
<td></td>
<td>object function</td>
</tr>
<tr>
<td>Zips</td>
<td>support vector machine</td>
</tr>
<tr>
<td></td>
<td>data set</td>
</tr>
<tr>
<td></td>
<td>real world</td>
</tr>
<tr>
<td></td>
<td>learning</td>
</tr>
<tr>
<td></td>
<td>state art</td>
</tr>
<tr>
<td></td>
<td>feature selection</td>
</tr>
<tr>
<td></td>
<td>machine learning</td>
</tr>
<tr>
<td></td>
<td>bayesian network</td>
</tr>
<tr>
<td></td>
<td>model selection</td>
</tr>
<tr>
<td></td>
<td>optim problem</td>
</tr>
<tr>
<td></td>
<td>high dimension</td>
</tr>
<tr>
<td></td>
<td>paper proposal</td>
</tr>
<tr>
<td></td>
<td>graphical model</td>
</tr>
<tr>
<td></td>
<td>result show</td>
</tr>
<tr>
<td></td>
<td>cross validation</td>
</tr>
<tr>
<td></td>
<td>decision tree</td>
</tr>
<tr>
<td></td>
<td>neural network</td>
</tr>
<tr>
<td></td>
<td>well known</td>
</tr>
<tr>
<td></td>
<td>hilbert space</td>
</tr>
</tbody>
</table>

Figure 8: The first 20 patterns extracted from the JMLR dataset by two baseline algorithms GoKrimp and SQS and the Zips algorithm. Common patterns discovered by all the three algorithms are bold.

Figure 9: The top 20 most compressing patterns extracted by Zips (left) and by GoKrimp (right) from the Tweets dataset.