Discovering compressing serial episodes from event sequences

A. Ibrahim¹ · Shivakumar Sastry² · P. S. Sastry¹

Abstract Most pattern mining methods yield a large number of frequent patterns, and isolating a small relevant subset of patterns is a challenging problem of current interest. In this paper, we address this problem in the context of discovering frequent episodes from symbolic time-series data. Motivated by the Minimum Description Length principle, we formulate the problem of selecting relevant subset of patterns as one of searching for a subset of patterns that achieves best data compression. We present algorithms for discovering small sets of relevant non-redundant episodes that achieve good data compression. The algorithms employ a novel encoding scheme and use serial episodes with inter-event constraints as the patterns. We present extensive simulation studies with both synthetic and real data, comparing our method with the existing schemes such as GoKrimp and SQS. We also demonstrate the effectiveness of these algorithms on event sequences from a composable conveyor system; this system represents a new application area where use of frequent patterns for compressing the event sequence is likely to be important for decision support and control.

Keywords Frequent episodes · Serial episodes · Mining event sequences · Discovering compressing patterns · MDL · Inter-event time constraints

1 Introduction

Frequent pattern mining is an important problem in the area of data mining that has diverse applications in a variety of domains [11]. Even though many algorithms have been proposed for frequent pattern mining, most of these methods produce a large number of frequent patterns. In addition, the patterns found are often redundant in the sense that many patterns are very similar. The redundancy and the large volume of the discovered patterns make it
difficult to use the mined patterns to gain useful insights into the data or to use them to extract rules which are effective for prediction, classification, etc. in the application domain. Thus, finding a small set of non-redundant, relevant and informative patterns that succinctly characterize the data is an important problem of current interest.

In this paper, we address the problem of discovering a set of patterns that can achieve succinct lossless representation of temporal sequence data, in the frequent episodes framework. We present algorithms that discover a small set of relevant patterns (which are special forms of serial episodes) which summarize the data well. We use the MDL principle [10] to define what we mean by summarizing the data well. The basic idea is that a set of patterns summarizes (or characterizes) the data sequence well, if the set of patterns can be used as a model to encode the data to achieve good compression.

There are many methods that are proposed for reducing the number of extracted frequent patterns. Many such methods concentrate on eliminating patterns that are deemed to be non-informative given the other frequent patterns. For example, in the context of transaction datasets, concepts such as closed [23, 31], non-derivable [5] and maximal [4, 18] itemsets were suggested to reduce the number of frequent itemsets extracted. Similarly, closed sequential patterns were proposed for sequence datasets [6, 30, 35]. Even though such methods result in some reduction in the number of patterns returned by the algorithm, the number of patterns still remains substantial. Also, the redundancy in the final set of patterns is, often, still large.

Recently, there have been other efforts for finding a small set of informative patterns that best describes the data. For example, Chandola and Kumar [7] propose a method for summarization of transaction datasets based on some ideas from information theory. They propose a method of selecting a subset of frequent itemsets to achieve a good lossy summarization of the database. Here, each transaction is summarized by one itemset with as little loss of information as possible. In Wang and Karypis [32], which also proposes a lossy summarization, each transaction is covered, partially, by the largest frequent itemset. In contrast to these methods, Siebes et al. [26] and Vreeken et al. [29] propose lossless summarization of transaction datasets using the Minimum Description Length (MDL) principle. A related approach called Tiling was used in Geerts et al. [9, 34], again for a lossless summarization of the data.

The Krimp algorithm proposed in Vreeken et al. [29] is one of the first methods to use MDL principle for identifying a small subset of relevant patterns in the context of frequent itemset mining. This algorithm selects a subset of frequent itemsets which, when used for encoding the database, achieves good compression. Each selected itemset is assigned a code with shorter code lengths assigned to higher-frequency itemsets. The algorithm tries to encode each transaction with the codes of itemsets which have minimal code lengths and which cover maximum number of items.

Similar strategies have been proposed for sequence data also [15, 16, 27]. For sequential data, unlike in the case of transaction data, the temporal ordering is important and this presents additional complications while encoding the data. For example, consider a single transaction, \( t = ABCD \) from a transaction database and two itemsets \( AC \) and \( BD \). The itemsets \( AC \) and \( BD \) can encode the transaction \( t \) (since the transaction is just a set of items). Now consider a sequence \( s = ABCD \) and two serial episodes \( A \rightarrow C \) and \( B \rightarrow D \). Even though the occurrences of \( A \rightarrow C \) and \( B \rightarrow D \) would cover the sequence \( s \), this information alone is insufficient for encoding the sequence. Since the order of events is important in sequential data, in order to get back the exact sequence, one needs to specify where exactly the occurrences of the episodes happen in the sequence. For example, we need to know that the \( A \) and \( C \) in the occurrence of \( A \rightarrow C \) are not contiguous and that there is a \( B \) in the gap between them. One needs to have some way of taking care of such gaps while encoding the
data with the occurrences of some frequent episodes. In general, the events in a sequence constituting an episode occurrence need not be contiguous, and different occurrences can have arbitrary temporal overlaps. An encoding scheme should be able to properly take care of this.

The previous approaches for using the MDL principle to summarize sequence data [15, 16, 27] explicitly record such gaps while encoding data, thus significantly increasing the encoding length. While the methods presented in Tatti and Vreeken [27] and Lam et al. [15] consider only sequential data without time stamps, the method in Lam et al. [16] does encode event sequences with time stamps also; but the encoding scheme needs to individually encode each event time stamp. In some cases, the resulting encoding may become even longer than the raw data [16]. For the problem of identifying a relevant subset of frequent patterns, we are using the encoded length (of the data encoded with a subset of patterns), only as a figure of merit to compare different subsets. Hence, the encoding length becoming more than the raw data is, per se, not disallowed. However, the underlying philosophy of MDL principle suggests that one needs a good level of data compression to have confidence in a model. Hence, if the best encoded length is more than the raw data, we cannot be confident that the subset of patterns that we used for encoding represents some useful regularities in the data. For example, even if the sequence data are iid noise and have no temporal structure, there would be some subset of patterns that would achieve lower encoded data length than other subsets. However, one expects that even the best such subset here would not achieve any appreciable level of data compression, thus suggesting that there are no significant temporal regularities in the data. On the other hand, for a sequence with significant temporal regularities, one expects good compression of the data sequence, when the sequence is encoded with a very good subset of temporal patterns. In general, if we can discover some long episodes which occur many times, then their occurrences can encode many events in the data sequence, thus giving rise to the possibility of data compression.

In this paper, we consider summarizing event sequences (having time stamps on events) using a pattern class consisting of serial episodes with fixed inter-event times. We present algorithms for discovering a small subset of relevant frequent episodes that result in good compression of the data sequence. The novelty of our approach is that, in contrast to the existing schemes [15, 16, 27], our method does not need to explicitly encode gaps in episode occurrences, and the encoding scheme is such that we can retrieve the full data sequence with the time stamps on events, from the encoded sequence. The encoding of the data consists of only the start times of occurrences of various episodes; the gaps are determined from the fixed inter-event time constraints of the episodes. We show through simulations that our method results in better data compression. We also show, through empirical experiments, that the episodes that result in good data compression are also highly relevant for the dataset.

We also illustrate the benefits of our algorithm using an application, where it is important to both find relevant patterns and achieve good data compression. We consider streams of sensor data from a composable conveyor system (CCS) [3, 25] that is useful for materials handling. In this system, several conveying units are dynamically composed to achieve the application objectives; consequently, utilizing the data streams to diagnose or reconfigure the system is important. The data consist of a sequence of predefined events such as package entered a unit, package exited a unit and package arrived at an input port. Such events occur at various units in the conveyor system during its routine operation. On this data stream, frequent serial episodes represent the routes (sequence of units) over which packages were transported in the conveyor system. The inter-event times correspond to the various physical constraints such as time required for a package to move through a specific unit and the time required for two adjacent units to complete a handshake protocol to transfer packages.
between them. Thus, a small set of relevant episodes can provide a good summary of the events in the conveyor system. We can use the discovered set of relevant episodes to achieve a lossless compression of the original temporal event sequence to support remote monitoring, diagnostics and visualization activities. We explain the system in more detail in Sect. 5.1.1. Using data obtained from a high-fidelity discrete event simulator of such conveyor systems, we demonstrate that (a) our algorithms unearth a small set of relevant episodes that capture the essence of the transport through the system and (b) our scheme achieves good data compression.

Even though our method is motivated by the above application, we show that our method is effective with other general sequential data as well. Apart from conveyor system data streams, we show the effectiveness of our methods with text data as well as on a few other real data sequences. These are the datasets that are used to illustrate the effectiveness of the algorithms presented in Tatti and Vreeken [27] and Lam et al. [15, 16]. We compare the performance of our algorithm with these methods on these datasets as well as on the composable conveyor system data.

The rest of the paper is organized as follows: in Sect. 2, we briefly review the formalism of episodes, introduce the new subclass of serial episodes and formally state the problem. Section 3 describes our encoding scheme for temporal data using our episodes. The various algorithms for mining and subset selection are explained in Sect. 4, and the experimental results are given in Sect. 5. We conclude the paper in Sect. 6.

2 Problem statement

2.1 Fixed interval serial episodes

The data we consider are a sequence of $n$ abstract events denoted as $D = ((E_1, t_1), (E_2, t_2), \ldots, (E_n, t_n))$, where $n$ is the number of events in the data stream. In each tuple $(E_i, t_i)$, which is also called an event, $E_i$ denotes the event type and $t_i \in \mathbb{Z}^+$ denotes the time of occurrence of the event. The event types $E_i$ take values from a finite alphabet, $\Sigma$. The sequence is ordered so that $t_i \leq t_{i+1}$ for all $i = 1, \ldots, n - 1$. Note that we can have multiple events (of different types, but not of the same type), all occurring at the same time instant.

A $k$-node serial episode $\alpha$ is denoted as $e_1 \rightarrow e_2 \rightarrow \cdots \rightarrow e_k$ where $e_i \in \Sigma, \forall i$. An occurrence of $\alpha$ in $D$ is a mapping $h : \{1 \ldots k\} \rightarrow \{1 \ldots n\}$, such that $e_i = E_{h(i)}, 1 \leq i \leq k$ and $t_{h(i)} < t_{h(j)}$, for $i < j$. An occurrence can be denoted by $(t_{h(1)}, \ldots, t_{h(k)})$, the event times of the events constituting the occurrence. We call the interval $[t_{h(1)}, t_{h(k)}]$ as the occurrence window of this occurrence. (If $k = 1$, then for the 1-node episode, the occurrence window is essentially a number which is the event time of that event). Consider an example event sequence

$$D_1 = ((A, 1), (A, 2), (B, 3), (E, 4), (A, 5), (B, 6), (C, 6), (B, 7), (D, 8), (C, 10), (E, 11))$$

In the data sequence given in (1), a few occurrences of episode $A \rightarrow B \rightarrow C$ are $(1, 3, 6), (2, 3, 6), (5, 6, 10)$.

A fixed interval serial episode is a serial episode with fixed inter-event gaps. A fixed interval serial episode is denoted as $\beta = e_1 \stackrel{\Delta_1}{\rightarrow} e_2 \stackrel{\Delta_2}{\rightarrow} \cdots \stackrel{\Delta_{k-1}}{\rightarrow} e_k$. We will be considering the class of fixed interval serial episodes, where $\Delta_i \leq T_g, \forall i$, with $T_g$ being a user specified upper bound on allowable gap. An occurrence of $\beta$ in $D$ is a mapping $h : \{1 \ldots k\} \rightarrow \{1 \ldots n\}$,
Discovering compressing serial episodes from event sequences 409

such that \( e_i = E_{h(i)}, 1 \leq i \leq k \) and \( t_{h(i+1)} - t_{h(i)} = \Delta_j > 0 \), for \( 1 \leq i < k \). For example, in sequence \( D_1 \) in (1), there are two occurrences of episode \( A \rightarrow B \rightarrow C \), namely (1, 3, 6) and (5, 7, 10). Note that the time of the first event of an occurrence completely specifies the entire occurrence. This property of the fixed interval serial episodes allows us to design a coding scheme that results in data compression. A \( k \)-node fixed interval serial episode \( \alpha = e_1 \xrightarrow{\Delta_1} e_2 \xrightarrow{\Delta_2} \cdots \xrightarrow{\Delta_{k-1}} e_k \) is called injective if \( e_i \neq e_j, \forall i, j, i \neq j \).

In the literature, different notions of frequency are defined for episodes depending on the type of occurrences we count. (For a discussion on various frequencies, see [2]). An episode is said to be frequent if its frequency is above a given threshold. In this paper, we consider the number of distinct occurrences as the frequency. Two occurrences are distinct if none of the events of one occurrence is among events of the other. More formally, a set of occurrences, \( \{h_1, h_2, \ldots, h_m\} \) of an episode \( \alpha \) are distinct if for any \( k \neq k', h_k(i) \neq h_{k'}(j), \forall i, j \). This is a natural notion of frequency for an injective fixed interval serial episode because any pair of its occurrences with different start times will always be distinct.

In this paper, we consider injective fixed interval serial episodes, and from now on, we refer to injective fixed interval serial episodes simply as episodes whenever there is no scope for confusion.

2.2 Selecting a subset of episodes using the MDL principle

Pattern mining algorithms often output a large number of frequent episodes. Our goal is to isolate a small subset of them which are non-redundant and are relevant for the data. To formalize this goal, we use the MDL principle which views learning as data compression. The idea is that if we can discover all the relevant regularities in the data, then an encoding based on these would result in data compression [10]. Thus, the goal is to find a model which allows us to encode the data in a compact fashion.

Given any model, \( H \), let \( L(H) \) denote the length for encoding the model \( H \) and let \( L(D|H) \) be the length of the data when encoded using the model \( H \). Given an encoding scheme, under the MDL principle our goal is to find a model \( H \) that minimizes total encoded length, \( L(H, D) = L(H) + L(D|H) \).

For us, different models correspond to different subsets of the set of frequent fixed interval serial episodes. As mentioned earlier, an occurrence of such an episode is uniquely specified by its start time. Hence, by giving the code for the identity of the episodes and a list of start times, we can code all the events constituting the occurrences of this episode. (We explain our encoding scheme in the next section). Thus, large episodes with many occurrences would account for a large number of events in the data sequence thus decreasing \( L(D|H) \). Another advantage of our use of the MDL principle is that it inherently takes care of redundancy. Selecting episodes with minimal overlap among their occurrences would help reduce the final encoded length.

Under MDL, we are looking at lossless coding and hence the occurrences of the selected subset of episodes have to cover the entire dataset; i.e., every event in the data sequence should be part of an occurrence of (at least) one of the selected set of episodes. We can always ensure this by adding a few 1-node episodes, as needed. We will give details of our encoding in the next section. Our main problem can now be stated as below:

**Problem 1** Given a data sequence \( D \) and a set of (frequent) fixed interval serial episodes, \( \mathcal{C} = \{C_1, C_2, \ldots, C_N\} \), find the subset \( \mathcal{H}^* \subseteq \mathcal{C} \) such that

\[
\mathcal{H}^* = \arg \min_{H \subseteq \mathcal{C}} \{L(H) + L(D|H)\}
\]
3 The encoding scheme for data

In this section, we explain our encoding scheme in detail and derive the expression for encoded data length. We also show how to decode the encoded data.

3.1 Encoding

Each model $H$ is a set of some fixed interval serial episodes whose occurrences cover the data. Given such an $H$, which forms the dictionary, the data are then encoded by specifying the start times of selected occurrences of the episodes.

We explain our encoding scheme through an example. Table 1 shows a data sequence $D_2$ and its encoding using three arbitrarily selected episodes. Each row in the table describes one of the episodes used and the encoding for the part of the data covered by the occurrences of that episode. There are four columns in the table. Column 1 gives the size of the episode in that row, and the second column specifies the episode. The third column gives the number of occurrences of that episode (used for encoding), and the last column gives a list of start times. Hence the first row of Table 1 specifies: a 3-node episode, namely $A \xrightarrow{2} B \xrightarrow{1} C$, and two of its occurrences starting at times 2 and 4. Thus, the first row of Table 1 accounts for the events $(A, 2), (B, 4), (C, 5)$ and $(A, 4), (B, 6), (C, 7)$ in the data, which are the events constituting the two occurrences of the episode $A \xrightarrow{2} B \xrightarrow{1} C$, starting at times 2 and 4, respectively. Similarly, the second row of the table specifies two occurrences of the 3-node episode $D \xrightarrow{2} E \xrightarrow{2} C$ with start times 1 and 5. Thus the second row accounts for the events $(D, 1), (E, 3), (C, 5)$ and $(D, 5), (E, 7), (C, 9)$. Now only two events in the data, namely $(C, 3)$ and $(C, 8)$, are left out, which are encoded by a 1-node episode as shown by the third row of the table. Thus we see that the selected subset of episodes covers the data. We can think of the first two columns of the table as our dictionary and the last two columns of the table as the encoding of the data. Note that the seventh event in $D_2$, namely $(C, 5)$, is part of the occurrence of $A \xrightarrow{2} B \xrightarrow{1} C$ starting at 2 and of $D \xrightarrow{2} E \xrightarrow{2} C$ starting at 1. While this is allowed in our encoding, minimizing such overlaps would improve total encoded length. In fact, we use injective fixed interval serial episodes in order to avoid overlap of different occurrences of the same episode, since, as we mentioned earlier, occurrences of injective fixed interval serial episodes starting at different time instants are distinct. Our final encoding of the sequence would be a table like this. Each row of the table is essentially a series of integers denoting first, the size of the episode, then the description of the episode, namely its event types with fixed inter-event gaps, its number of occurrences and the list of start times of the occurrences. This particular ordering enables exact decoding. By stringing together the sequence of integers in each row, we get the final encoding of the data.

Table 1 A data sequence and its encoding. $D_2 = (\langle D, 1 \rangle (A, 2) (C, 3) (E, 5) (A, 4) (B, 4) (C, 5) (D, 5) (B, 6) (C, 7) (E, 7) (C, 8) (C, 9))$

| Size of episode | Episode name | No. of occurrences | List of occurrences |
|-----------------|--------------|---------------------|---------------------|
| 3               | $A \xrightarrow{2} B \xrightarrow{1} C$ | 2                   | $\langle 2, 4 \rangle$ |
| 3               | $D \xrightarrow{2} E \xrightarrow{2} C$ | 2                   | $\langle 1, 5 \rangle$ |
| 1               | $C$          | 2                   | $\langle 3, 8 \rangle$ |
3.2 Decoding

In this section, we discuss how to decode the encoded data. The encoded data consist of rows of a table, with each row specifying an episode and its occurrences. In each row, we read the first value, which is the size of the pattern. If this value is \( k \), the next \( 2k - 1 \) integers correspond to the codes of the event types (\( k \) units) followed by the inter-event gaps (\( k - 1 \) units). The next value in the encoded sequence corresponds to the number of occurrences of the episode. We then need to read that many values to obtain all the occurrence start times of that episode and complete reading the current row. Since we know when the row is complete, the next integer would be the first entry of the next row and we repeat the same process as above. From each start time of occurrence of an episode, we can roll out the corresponding events because we know the event types and the inter-event gaps. Once we are done with rolling out all the occurrences of all the episodes, we have to just sort the events based on the time stamps and delete duplicate occurrences\(^1\) to retrieve back the original data sequence.

3.3 Length of the encoding

We have seen that once the model (or the dictionary) is fixed, the data encoding is just a series of integers denoting the start times of occurrences of the patterns in the dictionary. Even though we could use bit-level integer encoding schemes such as Elias codes [15,33] and Universal codes [24,27] for encoding integers (and have the size of encoding dependent on the value of the integer), we use the notion of fixed memory units instead. The reason is that the MDL principle looks at utilizing the regularity in data to compress the data and hence, in our context, the level of compression should not depend on the magnitude of data items. For example, the value of a time stamp, per se, does not have any regularity and the compression achieved by the encoding scheme, hence, should not be dependent on the values of time stamps. Therefore, for calculating the total encoded lengths we consider event types and times to be integers and assume that each such integer accounts for one unit in the encoded length. Since our aim is to compare different models, keeping all lengths in terms of one unit per integer is sufficient for us. (Here we are assuming that, in describing episodes in the dictionary, all event types take the same amount of memory. We could, of course, use codes such as Hamming codes, to reduce expected length of representation of episodes. We do not consider such extra compression here). Later on, while comparing our method with the various other methods, we use bit-level encoding for calculating lengths of integers so that we can easily compare with the results of other algorithms.

Let model \( H = \{F_1, F_2, \ldots, F_K\} \), with \( |F_i| \) denoting the size of episode \( F_i \). Each episode needs one integer to represent its size, \( |F_i| \) integers for representing the event types and \((|F_i| - 1)\) integers for inter-event gaps. Since we have \( K \) episodes in our subset, our table encodes the data with \( K \) rows. In the \( i \)th row, the first two columns would require \( 1 + |F_i| + (|F_i| - 1) = 2|F_i| \) integers. Hence, the first two columns of the table need \( \sum_{i=1}^{K} 2|F_i| \) integers. This is \( L(H) \).

Let \( f_i \) be the number of occurrences of \( F_i \) listed in the column 4 of our table. Hence, in the \( i \)th row of the table, column 3 needs one integer and column 4 needs \( f_i \) integers to specify the start times of the \( f_i \) occurrences. Thus, columns 3 and 4 of the table together need \( \sum_{i=1}^{K} (f_i + 1) \) integers. This is \( L(D|H) \). Thus for the model \( H \), the total encoded length is

\(^1\) Note that, in the data sequence, while events of different types can occur with the same time stamp, events with same event type cannot co-occur at the same time instant; hence, we can easily spot duplicates while decoding.
\[ L(H, D) = L(H) + L(D|H) = \sum_{i=1}^{K} 2|F_i| + \sum_{i=1}^{K} (f_i + 1) \quad (2) \]

For the encoding given in Table 1, the length for the first row is \(1 + (3 + 2) + 1 + 2 = 9\), and it is easy to see that the total encoded length is \(9 + 9 + 5 = 23\).

The length of raw data can be taken to be \(2|D|\) where \(|D|\) is the number of events in the data. (Recall that each element in \(D\) is a tuple of event type and the time of occurrence.) However, taking this as the length of uncompressed data may result in higher value for the compression achieved by an algorithm. This is because, even without finding any patterns, we can represent the raw data more compactly by simply using only 1-node episodes in our encoding. If we have \(M\) event types, then we use \(M\) 1-node episodes for encoding. The total encoded length, using Equation (2), and taking \(K = M\) would be \(2M + |D| + M\). (Note that \(\sum_{i=1}^{M} f_i = |D|\), because all occurrences of the \(M\) 1-node episodes together would exactly cover the data; also no event in the data would be part of occurrences of two different 1-node episodes.) We call such an encoding trivial encoding. For the data sequence \(D_2\), the length for trivial encoding would be \(5 \times 2 + 13 + 5 = 28\). Even though in this example the length of the trivial encoding is more than \(2|D|\), for real datasets we would have \(|D| \gg M\) and hence total encoded length of trivial encoding would be less than \(2|D|\). Hence, in calculating data compression with our method, we compare the length of the trivial encoding with the length of the encoding using selected episodes.

4 Algorithms

In this section, we consider algorithms for discovering a subset of episodes which achieves good compression. Finding the optimal subset of episodes to minimize total encoded length is known to be NP-Hard \([16,27]\). Hence, the methods we present here are approximation algorithms to Problem 1 in Sect. 2.2. We begin by presenting algorithm CSC-1 (CSC is for Constrained Serial episode Coding), which is a two-phase method. This consists of discovering all frequent episodes through a depth-first search algorithm followed by a greedy method of selecting a subset based on maximum coverage and minimum overlap. We then present algorithm CSC-2, which directly mines for relevant fixed interval serial episodes from the data without first discovering all frequent episodes.

4.1 First algorithm: CSC-1

We first explain our depth-first mining algorithm and the basis for the greedy strategy for subset selection before describing the full CSC-1 algorithm (which is listed in Algorithm 4).

4.1.1 Mining

To obtain all frequent injective fixed interval serial episodes, we use a depth-first (also known as pattern-growth) strategy using occurrence windows. See Achar et al. \([1]\) and Méger and Rigotti \([21]\) for more details on depth-first strategies using occurrence windows. The basic idea is as follows. First, we find all 1-node frequent episodes (which are event types that occur often enough) and, for each frequent 1-node episode, keep its occurrence list, which is a list of event times where the 1-node episode occurs in the data. Let \(\alpha\) be an episode and suppose we are given a list of all of its occurrence windows (also called occurrence list). Recall that the occurrence window of an episode \(\alpha\) is an interval \([t_s, t_e]\), where \(t_s\) and \(t_e\) are...
Algorithm 1 MineEpisodes(\( \mathcal{D} \), \( T_g \), \( f_{th} \))

\textbf{Input:} \( \mathcal{D} \): Sequence data; \( T_g \): Maximum inter-event gap; \( f_{th} \): Frequency threshold.

\textbf{Output:} \( \mathcal{C} \): The set of frequent episodes in \( \mathcal{D} \).

1: \( \mathcal{A} \leftarrow \) Set of all frequent 1-node episodes in \( \mathcal{D} \), along with their occurrence lists.
2: for all \( A \in \mathcal{A} \) do
3: \( \mathcal{C} \leftarrow \text{ExploreDFS}(A, \mathcal{A}, \mathcal{D}, T_g, f_{th}, \emptyset) \)
4: end for
5: return \( \mathcal{C} \)

Algorithm 2 ExploreDFS(\( \alpha \), \( \mathcal{A} \), \( \mathcal{D} \), \( T_g \), \( f_{th} \), \( \mathcal{C}' \))

\textbf{Input:} \( \alpha \): Episode named \( \alpha \) with its occurrence list; \( \mathcal{A} \): Set of frequent one node episodes with their occurrence lists; \( \mathcal{D} \): Sequence data; \( T_g \): Maximum inter-event gap; \( f_{th} \): Frequency threshold; \( \mathcal{C}' \): Set of all already mined frequent episodes.

\textbf{Output:} Union of the set of all already mined frequent episodes, \( \mathcal{C}' \) and the set of all frequent episodes with \( \alpha \) as prefix; the union being assigned back to \( \mathcal{C}' \).

1: for all \( A \in \mathcal{A} \setminus \{ \text{set of event types in } \alpha \} \) do
2: \( \text{occurrlist-for-delta} \leftarrow \text{Find-Lists}(\alpha, A, T_g) \)
3: for \( j = 1 \) to \( T_g \) do
4: if \( |\text{occurrlist-for-delta}(j)| \geq f_{th} \times |\mathcal{D}| \) then
5: \( \beta \leftarrow (\alpha \uparrow_j A) \)
6: \( \beta.\text{occurrencelist} \leftarrow \text{occurrlist-for-delta}(j) \)
7: \( \mathcal{C}' \leftarrow \mathcal{C}' \cup \beta \)
8: \( \mathcal{C}' \leftarrow \text{ExploreDFS}(\beta, \mathcal{A}, \mathcal{D}, T_g, f_{th}, \mathcal{C}') \)
9: end if
10: end for
11: end for
12: return \( \mathcal{C}' \)

Algorithm 3 Find-Lists(\( \alpha \), \( A \), \( T_g \))

\textbf{Input:} \( \alpha \): Episode named \( \alpha \) with its occurrence list; \( A \): One node episode with its occurrence list; \( T_g \): Maximum inter-event gap.

\textbf{Output:} \( \text{occurrlist-for-delta} \): Array storing occurrence lists for episodes \( \alpha \uparrow_j A \), 1 \( \leq j \leq T_g \).

1: Initialize \( \text{occurrlist-for-delta}(j) = \emptyset \), \( \forall j = 1 \ldots T_g \).
2: for all \( [t_s, t_e] \in \alpha.\text{occurrencelist} \) do
3: Let \( t^A \) be the first occurrence of \( A \) after \( t^\alpha \). \( \triangleright \) NULL if no such occurrence
4: while \( t^A \neq \text{NULL} \) and \( t^A - t^\alpha \leq T_g \) do
5: \( j \leftarrow t^A - t^\alpha ; \)
6: Add \([t^\alpha, t^A]\) to \( \text{occurrlist-for-delta}(j) \). \( \triangleright \) Corresponding to \( \alpha \uparrow_j A \)
7: \( t^A \leftarrow \text{next occurrence of } A \). \( \triangleright \) NULL if there is no next occurrence
8: end while
9: end for
10: return \( \text{occurrlist-for-delta} \)

The main function is MineEpisodes, listed as Algorithm 1. This is a wrapper function, which finds, for each event type \( A \in \mathcal{A} \), where \( \mathcal{A} \) is the set of frequent 1-node episodes, all
the frequent fixed interval serial episodes with A as the prefix, using the ExploreDFS function (in line 3). The frequency threshold value, $f_{th}$, is user specified and is assumed to be given as a fraction of the data length.

The function ExploreDFS, listed as Algorithm 2, is a recursive function. Given an input episode $\alpha$, it finds all the frequent right extensions of $\alpha$, i.e., all the frequent episodes with $\alpha$ as prefix. For each $A$, the function initially finds the occurrence windows for the episodes $\alpha \rightarrow A$, 1 $\leq j \leq T_g$, where $T_g$ is the maximum allowed inter-event gap (line 2, Algorithm 2), by calling the function Find-Lists, which is explained below. The function then recursively goes deeper for each frequent episode, $\alpha \rightarrow A$ (line 8).

The Find-Lists function (listed as Algorithm 3) takes as input, episode $\alpha$ and event type $A$ (which is also a 1-node episode) and finds the occurrence windows for all the episodes $\alpha \rightarrow A$, $j \leq T_g$. For each occurrence window $[t_{\alpha}^s, t_{\alpha}^e]$ of $\alpha$, it looks for all the occurrences $t^A$ of $A$ such that the new occurrence window satisfies the maximum inter-event gap constraint $T_g$ (the condition for while loop in line 4). An occurrence window satisfying the constraint is then added to the occurrence list corresponding to the episode $\alpha \rightarrow A$, where $j = t^A - t_{\alpha}^e$ (line 6).

Using these algorithms, we get all injective frequent fixed interval serial episodes. We then go on to select the best representative subset.

### 4.1.2 Selection strategy

Given a data sequence $D$, let $\alpha$ be an $N$-node fixed interval serial episode with frequency $f_\alpha^D$. We define the score of $\alpha$ in $D$ as

$$score(\alpha, D) = f_\alpha^D N - (2N + f_\alpha^D + 1)$$  \hspace{1cm} (3)

Recall that $2N + f_\alpha^D + 1$ is the total encoded length for encoding all the events that constitute the $f_\alpha^D$ occurrences of $\alpha$. Here, $2N$ is the length for encoding the description of episode $\alpha$, 1 unit is to mention the number of occurrences ($f_\alpha^D$) and $f_\alpha^D$ is the length for encoding the start times of the occurrences. It is easy to see that $f_\alpha^D N$ is a lower bound on the encoded data length for trivially encoding all the events in the $f_\alpha^D$ occurrences of (the $N$-node episode) $\alpha$ with 1-node episodes. Thus $score(\alpha, D)$ measures the gain in encoding length by encoding the $f_\alpha^D$ occurrences of $\alpha$ with $\alpha$, as compared to trivially encoding them with 1-node episodes.

If $score(\alpha, D) > 0$, then $\alpha$ is called a useful candidate since selecting it can improve encoding length by at least the value of $score(\alpha, D)$, in comparison with trivial encoding. From Equation (3), we can easily see that $score(\alpha, D) > 0$, if $f_\alpha^D > \frac{2N+1}{N-1}$. Thus the episode $\alpha$ will be a useful candidate if $f_\alpha^D > 5$ for $|\alpha| = 2$ and $f_\alpha^D > 3$ for $|\alpha| \geq 3$. But selecting any useful candidates would not lead to efficient encoding. For any pair of selected episodes, we also want their occurrences to have least number of events in common. Our subset selection procedure for encoding the data is based on greedy selection of episodes whose occurrences cover large number of events in the data and have low level of overlap with the occurrences of other selected episodes.

---

2 We note that this is a lower bound because $\alpha$ is an injective episode. When $\alpha$ is an injective episode, no two occurrences of $\alpha$ can share an event and hence $f_\alpha^D$ occurrences would contain $f_\alpha^D N$ events in the data sequence. If the episodes were non-injective, then there is a possibility of events being shared by different occurrences of the same episode and hence $f_\alpha^D N$ would not be the lower bound.
Let \( \mathcal{F}_s \) be a set of episodes of size greater than one. Given any such \( \mathcal{F}_s \), let \( L(\mathcal{F}_s, D) \) denote the total encoded length of \( D \), when we encode all the events which are part of the occurrences of episodes in \( \mathcal{F}_s \), by using episodes in \( \mathcal{F}_s \) as per our encoding scheme and encode the remaining events in data, if any, by episodes of size one.

Given any two episodes \( \alpha, \beta \), let \( OM(\alpha, \beta) \) denote the number of events in the data that are covered by occurrences of both \( \alpha \) and \( \beta \). We call \( OM \) the Overlap Matrix. We define, for \( \alpha \notin \mathcal{F}_s \)

\[
\text{overlap-score}(\alpha, D, \mathcal{F}_s) = f^\alpha_D N - \sum_{\beta \in \mathcal{F}_s} OM(\alpha, \beta) - (2N + f^\alpha_D + 1)
\]

(4)

Note that \( \text{overlap-score}(\alpha, D, \mathcal{F}_s) = \text{score}(\alpha, D) - \sum_{\beta \in \mathcal{F}_s} OM(\alpha, \beta) \) and is another measure for the gain in encoding length, when we add \( \alpha \) to \( \mathcal{F}_s \). The measure has an interesting property as explained below.

**Proposition 1** If \( \text{overlap-score}(\alpha, D, \mathcal{F}_s) > 0 \), then \( L(\mathcal{F}_s, D) > L(\mathcal{F}_s \cup \{\alpha\}, D) \).

**Proof** First, note that the difference in encoding will only be in the section of the data, where the encodings using \( \mathcal{F}_s \) and \( \mathcal{F}_s \cup \{\alpha\} \) differ. As is easy to see, \( \sum_{\beta \in \mathcal{F}_s} OM(\alpha, \beta) \) is an upper bound on the number of events of the occurrences of \( \alpha \), which are shared with the occurrences of episodes in \( \mathcal{F}_s \). Hence, \( f^\alpha_D N - \sum_{\beta \in \mathcal{F}_s} OM(\alpha, \beta) \) is a lower bound on the number of events not covered by anyone in \( \mathcal{F}_s \), which are covered by the occurrences of \( \alpha \). Hence, if we use \( \mathcal{F}_s \), it takes at least \( f^\alpha_D N - \sum_{\beta \in \mathcal{F}_s} OM(\alpha, \beta) \) units for encoding these events using size-1 episodes. In contrast, by adding \( \alpha \) to \( \mathcal{F}_s \), it takes \( 2N + f^\alpha_D + 1 \) units to encode those occurrences, independent of the number of events \( \alpha \) shares with other episodes. Thus the reduction in encoding length between the two is at least \( f^\alpha_D N - \sum_{\beta \in \mathcal{F}_s} OM(\alpha, \beta) - (2N + f^\alpha_D + 1) \), which is the \( \text{overlap-score}(\alpha, D, \mathcal{F}_s) \). Hence, the result. \( \square \)

Proposition 1 says that, with respect to an already selected set of episodes \( \mathcal{F}_s \), adding an episode \( \alpha \) with \( \text{overlap-score}(\alpha, D, \mathcal{F}_s) > 0 \) to the set \( \mathcal{F}_s \) would only reduce the total length of encoded data. Our greedy heuristic is to select the episode with maximum overlap-score. (Note that by definition, \( \text{overlap-score}(\alpha, D, \mathcal{F}_s) = \text{score}(\alpha, D) \), if \( \mathcal{F}_s = \emptyset \).

### 4.1.3 CSC-1

The CSC-1 algorithm selects a good subset of the frequent episodes based on minimizing encoding length. The algorithm takes as input, \( K \), the maximum number of episodes (of size greater than 1) in the final selected subset. Thus, it can be used a method to select the ‘best-\( K \)’ episodes or to select the best subset to achieve maximum compression (by choosing a very large value of \( K \)).

The CSC-1 algorithm for selecting a good subset of (maximum \( K \)) episodes is listed as Algorithm 4. The algorithm runs in iterations of mining frequent fixed interval serial episodes from the data sequence, then selecting, one by one, a set of good encoding episodes from the mined set and finally deleting the occurrences of the selected episodes from the sequence. The process is repeated until we found \( K \) good episodes or we cannot find any episode that can give any gain in encoding.

In each iteration of the while loop (lines 3-18), we first mine the set of frequent fixed interval serial episodes, \( C \), using the \text{MineEpisodes} function, explained in Sect. 4.1.1 (line 5). We next calculate the \( OM \) matrix (line 6) and then calculate the \( \text{overlap-score} \). Each iteration of the repeat loop (lines 7–15) looks for an episode in the current candidates set,
Algorithm 4 CSC-1(D, Tg, fth, K)

Input: D: Sequence data; Tg: Maximum inter-event gap; fth: Frequency threshold; K: Maximum number of selected episodes.

Output: F: Selected set of frequent episodes.

1: F ← ∅
2: coveringexists ← true
3: while coveringexists and |F| < K do
4:  F_s ← ∅
5:  C ← MineEpisodes(D, Tg, fth)
6:  OM ← FindOverlapMatrix(D, C)
7:  repeat
8:    α ← arg maxγ∈C overlap-score(γ, D, F_s)
9:    if overlap-score(α, D, F_s) > 0 then
10:       F_s ← F_s ∪ {α}
11:    else if overlap-score(α, D, F_s) ≤ 0 and F_s = ∅ then
12:       coveringexists ← false
13:    end if
14:  until overlap-score(α, D, F_s) ≤ 0 or |F ∪ F_s| = K
15:  D ← D\(occurrences of F_s)
16:  F ← F ∪ F_s
17: end while
18: A ← Size-1 episodes in remaining D
19: F ← F ∪ A
20: return F

C, which has the highest positive overlap-score. If such an episode is found, it is added to the set F_s. This greedy strategy is justified by Proposition 1. The set F_s thus contains all the episodes selected in the repeat loop. The repeat loop is broken, when no episode with positive overlap-score exists in the current candidate set or we have selected K episodes (line 15). Then all the events in the occurrences of the episodes in F_s are deleted from the data (line 16). We then once again repeat the process of finding frequent episodes from this modified data and selecting a subset of episodes from this episode set.

The while loop runs as long as the selected set size is less than K, and it finds at least one episode that increases encoding efficiency. This condition is checked in lines 12–13. When we cannot find any more episodes with positive overlap-score, we encode the remaining events in the data with 1-node episodes (lines 19–20).

The only remaining part in the CSC-1 algorithm is the calculation of the matrix OM (line 6), which we explain now. The function FindOverlapMatrix, listed as Algorithm 5, utilizes the occurrence lists for all the frequent episodes (obtained from Algorithms 2, 3), to calculate OM matrix by one more pass over data using the standard finite state automata (FSA)-based method for tracking episode occurrences [2,17,19]. FSAs in Mannila et al. [19], Laxman et al. [17] and Achar et al. [2] are used to track occurrences of episodes. Algorithm 5 uses FSAs for a different purpose since we already have the occurrences of the episodes. Here, the FSAs associated with episodes, and hence some times called episode automata, are used to find, for each event (E_i, t_i) in the sequence, the set of episodes for which this event is part of one of their occurrences. For all such pairs of episodes in that set, the OM matrix is incremented by 1.

In Algorithm 5, each state of an episode automaton specifies the state of a current occurrence and is denoted by (α, j, t_s), where α is the episode, j is the state of the automaton to which it is expecting to transit (this means that for the current occurrence of the episode,
it has seen events for $\alpha[1]$ to $\alpha[j - 1]$ satisfying the inter-event constraints and is waiting for the event to occur with event type $\alpha[j]^3$, and $t_s$ denotes the start time of the current occurrence. An episode automaton corresponding to a $N$-sized episode has $(N + 1)$ states. An automaton for episode $\alpha$ is in state $j = 1$ while it is waiting for the event with event type $\alpha[1]$ and is in state $j = |\alpha| + 1$ at the end of its occurrence.

Each event type, $E$, is associated with a data structure called $waits$. $waits(E)$ contains the list of automaton waiting for the event with event type $E$ to occur. Whenever an automaton corresponding to an episode $\alpha$ transits to a state $j$, in wait for an event type $E$, that corresponding automaton state $(\alpha, j, t_s)$ is added to $waits(E)$ (lines 12 and 19, Algorithm 5).

For each episode, $\alpha$, as we go along the sequence, the start of one of its occurrences (which we know apriori) initiates an automaton for the episode $\alpha$. As we parse through the events in $D$ (for loop in line 6), each automaton corresponding to occurrences of the episodes will be waiting for a specific event to come up. On seeing the event $(E_i, t_i)$, those episodes (automaton) that were waiting for the event type, $E_i$ (which is obtained from $waits(E_i)$ (the inner for loop)), compare the start times with $t_i$ to see whether the event is part of the current occurrence (line 10 and lines 15,16, Algorithm 5). If the constraints are satisfied, then the episode is noted as having accepted that event (in line 11 and line 17, Algorithm 5) and moves to the next state. Finally, for those episodes that have accepted the current event, the corresponding $OM$ counts are incremented (in line 25).

---

3 $\alpha[i]$ denotes the $i$th episode event of $\alpha$. 

---

**Algorithm 5** FindOverlapMatrix($D, C$)

**Input:** $D$: Sequence dataset; $C$: Set of frequent multiple node episodes in $D$ with their occurrence lists.

**Output:** $OverlapMatrix$: $|C| \times |C|$ matrix containing the number of shared events between pairs of episodes in $C$.

1: Initialize $OverlapMatrix[\alpha][\beta] = 0, \forall \alpha, \beta \in C$
2: Initialize $waits(E) = \emptyset, \forall E \in \Sigma$
3: for all $\alpha \in C$ do
4: Add automata state $(\alpha, 1, \emptyset)$ to $waits(\alpha[1])$
5: end for
6: for all event $(E_i, t_i) \in D$ do
7: overlaplist $\leftarrow \emptyset$
8: for all automata $(\alpha, j, t_s) \in waits(E_i)$ do
9: if $j = 1$ then
10: if $\exists$ occurrence of $\alpha$ starting at $t_i$ then
11: Add $\alpha$ to overlaplist
12: Add automata state $(\alpha, j + 1, t_i)$ to $waits(\alpha[j + 1])$
13: end if
14: else
15: $t'_i = t_s + \Sigma_{i=1}^{j-2} \alpha.\Delta[i]$
16: if $(t_i - t'_i) = \alpha.\Delta[j - 1]$ then
17: Add $\alpha$ to overlaplist
18: if $j \neq |\alpha|$ then $\triangleright$ If $j = |\alpha|$, we just retire the automaton
19: Add automata state $(\alpha, j + 1, t_s)$ to $waits(\alpha[j + 1])$
20: end if
21: Remove automata state $(\alpha, j, t_s)$ from $waits(E_i)$
22: end if
23: end if
24: end for
25: Increment $OverlapMatrix[\alpha][\beta]$ by 1, $\forall \alpha, \beta \in overlaplist, \alpha \neq \beta$
26: end for
27: return $OverlapMatrix$
4.1.4 Complexity of the algorithm

There are three main steps in each iteration of Algorithm 4: the mining step (call to MineEpisodes in line 5), the calculation of the OM matrix (line 6) and the selection step (the repeat loop). The mining step involves the generation of occurrence start times of the selected episodes. For each episode \(\alpha\), this is done by a single pass over the occurrences of its prefix subepisode of size \((|\alpha| - 1)\) and 1-node suffix subepisode. The number of occurrences is of the order of \(|\mathcal{D}|\), and hence, for \(C\) selected episodes, the mining step takes \(O(|C||\mathcal{D}|)\).

The OM matrix calculation depends on the number of episodes accepting each event in \(\mathcal{D}\). Let \(K_i\) denote the number of episodes in \(C\) containing event type \(A_i\). Let \(P_i\) denote the fraction of the events of type \(A_i\) in \(\mathcal{D}\). Then each event type \(A_i\) occurs \(P_i|\mathcal{D}|\) times, and each of its occurrence is associated with \(O(K_i^2)\) updates of the OM matrix. Hence, the runtime for the calculation of the OM matrix is \(O((\sum_{i=1}^{M} P_i K_i^2)|\mathcal{D}|) = O(|C|^2|\mathcal{D}|)\), since \(K_i < |C|\).

The repeat loop involves calculating the overlap-score(\(\gamma, \mathcal{D}, \mathcal{F}_s\), \(\forall \gamma \in \mathcal{C}\), which takes \(O(|\mathcal{C}||\mathcal{F}_3|)\) and hence for \(|\mathcal{F}_3|\) iterations takes \(O(|\mathcal{C}||\mathcal{F}_3|^2)\). Since \(|\mathcal{F}_3| < |\mathcal{C}|\) the total computation for each iteration of our method is \(O(|\mathcal{C}|^2|\mathcal{D}| + |\mathcal{C}|^3)\). Hence, the total execution is critically dependent on \(|\mathcal{C}|\), the number of frequent episodes mined from the data.

4.2 An improved algorithm: CSC-2

The runtime of the CSC-1 method as shown in the previous section is \(O(|\mathcal{C}|^2|\mathcal{D}| + |\mathcal{C}|^3)\) and hence depends directly on the size of the mined candidate set of episodes, \(|\mathcal{C}|\). Thus an increase in the size of \(\mathcal{C}\) would increase the runtime by a cubic order. CSC-2 addresses exactly this problem. CSC-2 proceeds exactly as CSC-1 except that the set \(\mathcal{C}\) of candidate frequent episodes that it considers at each iteration is a small set rather than the full set of frequent fixed interval serial episodes. Earlier, in the MineEpisodes function, we have mined for all frequent episodes in a depth-first manner and gave that as the set \(\mathcal{C}\). Now, instead of that, we obtain a set of episodes, each one of which is a best possible episode in one of the paths in the depth-first search tree. Here, the best episode is decided in a greedy fashion based on its contribution to coding efficiency. Note that the CSC-2 algorithm does not need any frequency threshold to be specified by the user because we do not mine for ‘frequent’ episodes.

The algorithm CSC-2 is same as Algorithm 4 except that the call to MineEpisodes (line 5, Algorithm 4) is replaced by a call to a new function BestExtensions. The rest of Algorithm 4 remains the same for both CSC-1 and CSC-2. Hence, we do not provide separate pseudocode for CSC-2. The function BestExtensions, which selects a set of candidate episodes, \(\mathcal{C}\), is listed as Algorithm 6.

For each event type \(A\), Algorithm 6 extends the 1-node episode \(A\) with an event type \(B\) and gap \(i\) to form the episode \(A \rightarrow B\) such that

\[
(B, i) = \arg \max_{\mathcal{C}, j} \frac{\text{score}(A \xrightarrow{j} \mathcal{C}, \mathcal{D})}{C \neq A, j \leq T_{\mathcal{C}}}
\]

The episodes are grown by right extension until none of the immediate extensions has a better score than the current episode. At this point, we add the episode to the list of candidate episodes \(\mathcal{C}\).

The extension algorithm is given in Algorithm 7. Among the immediate extensions, Algorithm 7 selects the one with the maximum frequency. In case of ties with multiple extensions having the same frequency, then the one with the minimum gap is selected (condition on lines 7, 8, Algorithm 7).
Algorithm 6 BestExtensions(D, Tg)

**Input:** D: Sequence data; Tg: Maximum inter-event gap.

**Output:** C: Selected set of relevant candidate episodes.

1: $C \leftarrow \emptyset$
2: $A \leftarrow$ Set of all 1-node episodes in $D$, along with their occurrence lists.
3: for all $A \in A$ do
4: $patt \leftarrow A$
5: $newpatt \leftarrow patt$
6: while $(patt \leftarrow Extensions(patt, A, Tg)) \neq NULL$ do
7: $newpatt \leftarrow patt$
8: end while
9: if $score(newpatt, D) > 0$ then
10: $C \leftarrow C \cup \{newpatt\}$
11: end if
12: end for
13: return $C$

Algorithm 7 Extensions($\alpha, A, Tg$)

**Input:** $\alpha$: Episode named $\alpha$ with its occurrence list; $A$: Set of frequent one node episodes with their occurrence lists; $Tg$: Maximum inter-event gap.

**Output:** Episode extension with highest frequency and minimum inter-event gap.

1: $freq_{max} \leftarrow 2^{2(|\alpha|+1)+1} - |\alpha|$ $\triangleright$ Min frequency for which coding efficiency can be achieved
2: $gap_{min} \leftarrow Tg$
3: $maxpatt \leftarrow NULL$
4: for all $A \in A \setminus \{set of event types in $\alpha\}$ do
5: $occurrlist-for-delta \leftarrow \text{Find-Lists}(\alpha, A, Tg)$ $\triangleright$ Find-Lists in Algorithm 3
6: $gap_{best} \leftarrow \arg \max_{j \leq Tg} |\text{occurrlist-for-delta}(j)|$
7: if $|\text{occurrlist-for-delta}(gap_{best})| \geq freq_{max}$ then
8: if $|\text{occurrlist-for-delta}(gap_{best})| > freq_{max}$ OR $gap_{best} < gap_{min}$ then
9: $\beta \leftarrow (\alpha \xrightarrow{gap_{best}} A)$
10: $\beta.\text{occurrlist} \leftarrow \text{occurrlist-for-delta}(gap_{best})$
11: $\beta.\text{freq} \leftarrow |\text{occurrlist-for-delta}(gap_{best})|$
12: $maxpatt \leftarrow \beta$
13: $freq_{max} \leftarrow \beta.\text{freq}$
14: $gap_{min} \leftarrow gap_{best}$
15: end if
16: end if
17: end for
18: return $maxpatt$

The number of candidates generated using this procedure is at most $M$, the size of the alphabet. In general, this is much smaller than the size of $C$ in CSC-1. However, for large alphabet (i.e., large number of event types), even this could be costly. Then we can modify CSC-2 to keep only the ‘best few’ episodes in $C$.

5 Experiments

In this section, we present the experimental results for our method and compare its performance with that of SQS [27] and GoKrimp [15]. Since we observed that algorithm CSC-2 gives similar patterns as CSC-1, but is much more efficient time wise, we present simulation results with CSC-2 only.
We consider three different types of data. The first type of data is the conveyor system data, since this application was the motivation for us to come up with the new subclass of fixed interval serial episodes and our novel encoding scheme. The set of data sequences that we consider is generated by a detailed simulator of composable conveyor systems. As we mentioned earlier, this is an application area where compression achieved may be useful on its own. We explain more about the problem and the data in Sect. 5.1.1.

The second type of data that we consider is text data. We consider JMLR dataset which contains 787 abstracts from the Journal of Machine Learning Research. This is a sequential symbolic data, but the time stamps, so to say, are just the serial number of the word in the sequence. This dataset is used to see how the various algorithms perform in unearthing relevant phrases (patterns) related to machine learning research. We show the top 20 patterns found by different methods on this dataset.

The final collection of data that we consider consists of five real-world datasets introduced in [22]. Each of these is a database of symbolic interval sequences with class labels; i.e., events in these databases are denoted by a symbol and an interval of its occurrence. As in Lam et al. [15], we consider the start and end of the interval to be two different events with different event types. For example, the event interval \((e, t_1, t_2)\), where \(e\) is the symbol and \([t_1, t_2]\) is the interval of its occurrence, would be considered as two events \((e^-, t_1)\) and \((e^+, t_2)\).

Table 2 gives some relevant details regarding these datasets. The reason for using these datasets is that these are the datasets on which results of performance by SQS and GoKrimp are reported. On all these datasets, we compare the data compression and the classification accuracy of our method with that achieved by SQS and GoKrimp.

Our algorithms are implemented in C++, and the experiments were executed single threaded on an Intel i7 4-core processor with 16 GB of memory running over a Linux OS. The source code for our algorithms will be made available on request. The implementations of SQS and GoKrimp were obtained from the respective authors. For the GoKrimp algorithm, there is a significance level parameter (for a test called sign test), which is set to the default value of 0.01, and the minimum number of pairs needed to perform a sign test is set to the their default value of 25. For our algorithm, the maximum inter-event gap was set to 5 for all the datasets. The \(K\) value in the CSC-2 algorithm denoting the maximum number of patterns to be selected is set to infinity (which means maximum possible selection of episodes) unless otherwise stated.

### 5.1 Results on composable conveyor system data sequences

Before giving the results, we first give a brief explanation about composable conveyor systems [3,25].

| Datasets   | Events | Sequences | Classes | Alphabet size, \(M\) |
|------------|--------|-----------|---------|----------------------|
| jmlr       | 75,646 | 787       | NA      | 3846                 |
| aslbu      | 36,500 | 441       | 7       | 190                  |
| aslgt      | 178,494| 3493      | 40      | 47                   |
| auslan2    | 1800   | 200       | 10      | 16                   |
| context    | 25,832 | 240       | 5       | 56                   |
| pioneer    | 9766   | 160       | 3       | 92                   |
5.1.1 Conveyor system

Material handling conveyor systems move packages or material units from one or more inputs to specified outputs along predetermined paths. Such systems are used extensively in manufacturing, packaging, packet sorting, etc. A typical conveyor system with two inputs and two outputs is shown in Fig. 1. This system is composed using instances of Segment and Turn units that each operate autonomously. A Segment moves a package over a predetermined length over its belt. A Turn is a unit that can serve as a merger or splitter for package flow. Each segment and turn unit has a predetermined maximum speed of operation, and each unit has local sensors and actuators that are used to autonomously control its local behavior. The simulator we used produces a detailed event trace of every change of state in the sensors and actuators of each unit as packages move through the system.

The movement of the packages in the conveyor system is transformed to a temporal sequence as follows. We label the exit points of each input, output, turn and segment. We consider the transfer of a package from one conveyor unit to another as an event. The unit represents the type of the event, and the time at which each event occurs is recorded as the event time. Hence, when a package moves along a path, an ordered sequence of events is recorded. All the events corresponding to one package may not be contiguous because during the same time other packages may be moving along other paths.

We consider three conveyor system datasets coming from three different topologies. Each topology can be identified by the various paths from the inputs to the outputs and the package input rates. We consider the package input arrival rates to be Poisson. The details of the three datasets are given in Table 3, and the structure of the topologies is given in Figs. 1, 2 and 3.

For the conveyor system data, there is only one data sequence corresponding to each topology. The GoKrimp algorithm works using a statistical dependency test called Sign Test, which does not work on single sequence datasets. The implementation of GoKrimp asks such sequences to be broken into a number smaller sequences. Hence, the datasets were broken into sequences of size 25, 50 and 100, respectively, for the 2I-2O, 3I-3O and the package sorter topologies (creating longer sizes for sequences with longer and higher number of paths). Also, the conveyor system datasets are time stamped datasets. The SQS and GoKrimp algorithms do not take time stamped data. Hence, experiments for SQS and GoKrimp on these datasets are carried out after removing the time stamps, while retaining the temporal ordering of the symbols.
### Various conveyor system topologies, input rates, alphabet size $M$, number of events (in the final dataset with the topology) and the paths in each topology

| Topology          | Input rate | $M$ | Events | Paths                                      |
|-------------------|------------|-----|--------|--------------------------------------------|
| 2I-2O             | 0.6        | 16  | 7659   | **P1.** [I1 S1 T1 S4 T3 S7 T4 S8 O2]       |
| (see Fig. 1)      |            |     |        | **P2.** [I2 S6 T3 S7 T4 S5 T2 S3 O1]       |
| 3I-3O             | 0.4        | 33  | 12,497 | **P1.** [I1 S1 T1 S4 T3 S12 T7 S16 T8 S17 T9 S18 O3] |
| (see Fig. 2)      |            |     |        | **P2.** [I2 S6 T3 S7 T4 S5 T2 S3 T5 S9 O1]   |
|                   |            |     |        | **P3.** [I3 S15 T7 S16 T8 S17 T9 S14 T6 S11 O2] |
| Package sorter    | 0.2        | 64  | 18,489 | **P1.** [I1 S39 T13 S32 S33 S34 T14 S28 S25 S22 T8 S19 T9 S20 T10 S21 T11 S24 T12 S27 O2] |
| (see Fig. 3)      |            |     |        | **P2.** [I2 S31 T13 S32 S33 S34 T14 S35 T15 S36 S37 S38 T16 S40 O3] |
|                   |            |     |        | **P3.** [I3 S16 T6 S17 T7 S13 T3 S5 S6 S7 S8 S9 T4 S10 T5 S11 O1] |
|                   |            |     |        | **P4.** [I4 S2 T1 S3 T2 S4 T3 S5 S6 S7 S8 S9 T4 S10 S5 S15 T11 S24 T12 S27 O2] |
|                   |            |     |        | **P5.** [I5 S1 T1 S3 T2 S12 T6 S17 T7 S18 T8 S19 T9 S23 S26 S29 T15 S36 S37 S38 T16 S40 O3] |

The best values of the results are in bold

---

**Fig. 2** 3I-3O: a three-input, three-output conveyor system

#### 5.1.2 Interpretability of selected patterns on conveyor system dataset

Since the Conveyor system dataset consists of movement of packages through specific paths, we expect the good patterns to be subpaths of these paths. The results on the three Conveyor
Discovering compressing serial episodes from event sequences

Fig. 3 Package sorter topology

system datasets are given in Table 4. In all the three data sequences, our algorithm retrieves episodes corresponding to the subpaths of package flow (see Table 3 for the actual paths). These patterns actually correspond to subpaths where the packages move along the segments and turns at a fixed speed, without any congestion. The patterns returned by SQS and GoKrimp did not have any particular significance with respect to the topologies. Even though SQS gave some long patterns, those did not give any information regarding the underlying system of transportation of packages.

5.1.3 Effectiveness and efficiency of the algorithms

In this section, we compare the effectiveness and efficiency of the methods in terms of compression achieved, the number of patterns returned and the run time on the conveyor system dataset. Since the other methods, GoKrimp and SQS, employ a bit-level coding scheme, for comparison purpose, we also calculate the exact bits needed for our encoding scheme, even though it does not affect the encoding efficiency as discussed earlier. The encoding scheme remains exactly the same except for the specification of the alphabet size \( M \) in the beginning of the encoding. Since we only deal with positive integers, we assume that all the positive integers are encoded using the Elias codes [15]. For a positive integer, \( n \), the Elias code length for encoding \( n \) is \( 2\lfloor \log_2(n) \rfloor + 1 \). Each event type is encoded using a fixed code size of \( \lfloor \log M \rfloor + 1 \) bits. It is easy to see that this bit-level encoding scheme can also be easily decoded as discussed earlier.

For each method, the compression achieved is calculated as the ratio of size of encoded data using singleton patterns (which are 1-node episodes) to the size of encoded data using the subset of patterns returned by the methods. Table 5 shows the compression ratios achieved by the three algorithms on the conveyor system datasets. Here, the column CSC-2 lists compression ratio in terms of memory units, while the last column CSC-2 (Bitwise) gives the...
Table 4 Patterns discovered by various algorithm in the conveyor system datasets

| Method  | Topology | Top discovered patterns |
|---------|----------|-------------------------|
| GOKRIMP | 2I-2O    | [ S8 T4 S4 S2 T1 ], [ T3 I2 S3 S6 ], [ I1 O2 S8 ], [ S6 S4 S7 S5 ], [ S2 T1 S1 ], [ S8 T4 S4 ] |
|         | 3I-3O    | [ T5 T9 ], [ S10 I1 S14 ], [ S2 T1 ], [ T3 T6 S18 ], [ S16 S12 ], [ S8 S6 ] |
| PS      |          | [ T2 S4 ], [ S27 O2 ], [ S10 S12 ], [ S17 S18 ], [ S19 T10 ], [ S34 S35 ] |
| SQS     | 2I-2O    | [ S3 S8 T2 S7 T4 S3 I2 O2 T4 ], [ O1 S8 T4 ], [ O1 S8 T2 T4 ], [ S3 S8 T2 T4 T3 S6 I2 O1 O2 ], [ T4 T3 S6 ][ S5 S7 T4 T3 ] |
|         | 3I-3O    | [ T6 T9 ], [ O2 S18 ], [ O1 S11 ], [ O3 S9 ], [ T8 T7 S7 ], [ T7 T3 ] |
| PS      |          | [ T16 S24 ], [ S15 S36 ], [ T11 S36 ], [ T5 S24 ], [ T10 T15 ], [ S21 S36 ], [ T12 S38 ] |
| CSC-2   | 2I-2O    | [ S6 T3 S7 T4 S5 T2 S3 O1 ], [ S1 T1 S4 T3 S7 T4 S8 O2 ], [ S4 T3 S7 T4 S8 O2 ], [ I1 I2 ], [ T3 S7 T4 ], [ T1 I2 ] |
|         | 3I-3O    | [ S15 T7 S16 T8 S17 T9 S14 T6 S11 O2 ], [ S6 T3 S7 T4 S5 T2 S3 T5 S9 O1 ], [ S12 T7 S16 T8 S17 T9 S18 O3 ], [ I1 S1 T1 T3 T7 S16 T8 S17 T9 ], [ T7 S16 T8 S17 T9 ] |
| PS      |          | [ T3 S5 S6 S7 S8 S9 T4 S10 T5 ], [ S12 T6 S17 T7 T8 S19 T9 S23 S26 T15 S36 S37 S38 T16 S40 O3 ], [ S39 T13 S32 S33 S34 T14 S28 S25 T8 S19 T9 S20 T10 T11 S24 T12 S27 O2 ], [ T15 S36 S37 S38 T16 S40 O3 ], [ T13 S32 S33 S34 T14 ], [ S16 T6 S17 T7 T3 S5 S6 S7 S8 S9 T4 S10 T5 ], [ S1 T1 S3 T2 T6 S17 T7 ] |

The patterns for the CSC-2 are shown without the inter-event gap. PS stands for package sorter.

Table 5 Compression achieved by various algorithms on conveyor system data sequences

| Datasets        | SQS | GoKrimp | CSC-2 | CSC-2 (Bitwise) |
|-----------------|-----|---------|-------|-----------------|
| 2I-2O           | 1.42 | 1.36    | 4.56  | 4.79            |
| 3I-3O           | 1.13 | 1.11    | 4.63  | 4.89            |
| Package sorter  | 1.04 | 1.015   | 3.34  | 3.6             |

The best values of the results are in bold.

Table 6 Number of patterns returned by various algorithms on conveyor system data sequences

| Datasets        | SQS | GoKrimp | CSC-2 |
|-----------------|-----|---------|-------|
| 2I-2O           | 65  | 20      | 13    |
| 3I-3O           | 114 | 22      | 27    |
| Package sorter  | 140 | 9       | 77    |

bit-level compression ratio for CSC-2. As can be seen, our compression ratio is better by a factor of 3. As mentioned earlier, remote visualization and monitoring of such composable conveyor system is a potential application area of temporal data mining where the compression achieved is also important. Our method achieves better compression and also returns a very relevant set of episodes.
Table 7 shows the number of patterns returned by the three methods. Ideally for datasets with some inherent regularity in the sequence, a few significant patterns should explain the dataset well. In such cases, a good method should return a few relevant patterns. For the conveyor system datasets, where there is inherent regularity in the form of packet flows, CSC-2 returns a few highly relevant patterns, which happen to be subpaths of packet flows. GoKrimp also returns a small number of patterns, though as seen in Table 4, they are not relevant episodes. The package sorter topology is a complex topology with lots of intersecting paths. Hence, CSC-2 captures more number of patterns here than from other topologies. But for the same topology, GoKrimp gives the minimum number of patterns and none of these patterns capture good regularities in the underlying system. GoKrimp algorithm fails to detect any inherent regularity in the datasets (which is also seen by the low compression achieved). SQS returns the highest number of patterns in all cases, which are not at all relevant to these topologies. Table 7 shows run times of the three methods. As can be seen from the table, both GoKrimp and CSC-2 are very efficient in terms of run time, while SQS is almost 10 times slower than CSC-2.

5.2 Results on other datasets

We now discuss the experimental results on the datasets given in Table 2. In Sect. 5.2.1, we discuss the interpretability of the patterns output by different methods on the JMLR data. In Sect. 5.2.2, we compare the methods for efficiency in terms of compression achieved, runtime and the number of patterns. In Sect. 5.2.3, the effectiveness of the selected patterns from different methods is analyzed using the patterns as features for classification.

5.2.1 Interpretability of selected patterns

Of the six real-world datasets, only the outputs from the JMLR data could be analyzed for interpretability. For the JMLR dataset, we expect key phrases relevant to Machine Learning Research to pop up. We ran the CSC-2 algorithm with $K = 20$ to find the top 20 selected patterns. For the other two algorithms, we just selected the top 20 of the outputted patterns. Table 8 gives the patterns obtained on JMLR text data. As can be seen, the patterns returned by all three of the methods are relevant and almost identical. Also we could not see any redundancy in any of the pattern sets.

5.2.2 Efficiency of the algorithms

In this section, we analyze the efficiency of the methods in terms of compression achieved, the number of patterns returned and the run time on the datasets listed in Table 2. Table 9 presents the comparison of compression ratio for different methods. Again, the column named CSC-2 lists compression ratio in terms of memory units and the last column, CSC-2 (Bitwise), gives the bit-level compression ratio for CSC-2. As seen in Table 9, CSC-2 offers better compression on almost all the datasets.

| Datasets      | SQS | GoKrimp | CSC-2 |
|---------------|-----|---------|-------|
| 2I-2O         | 6   | 1       | 1     |
| 3I-3O         | 14  | 1       | 1     |
| Package sorter| 20  | 1       | 2     |
Table 8  Patterns discovered by various algorithm in the JMLR data

| Method     | Top discovered patterns |
|------------|-------------------------|
| GOKRIMP    | Support vector machin   |
| Real world | High dimension          |
| Machin learn | Reproduc hilbert space |
| Dataset    | Larg scale              |
| Bayesian network | Independ compon analysi |
| SQS        | Support vector machin   |
| Machin learn | Nearest neighbor        |
| State art  | Decis tree              |
| Dataset    | Neural network          |
| Bayesian network | Cross valid          |
| CSC-2      | Support vector machin   |
| Dataset    | Real world              |
| Learn algorithm | Reproduc kernel Hilbert space |
| Machin learn | Model select           |
| Bayesian network | High dimension      |

Table 9  Compression achieved by various algorithms

| Datasets  | SQS   | GoKrimp | CSC-2  | CSC-2 (bitwise) |
|-----------|-------|---------|--------|-----------------|
| jmlr      | 1.039 | 1.008   | 1.07   | 1.117           |
| aslbu     | 1.155 | 1.123   | 1.17   | 1.24            |
| aslgt     | 1.308 | 1.156   | 1.98   | 1.99            |
| auslan2   | 1.571 | 1.428   | 1.88   | 1.96            |
| context   | 2.7   | 1.7     | 1.95   | 1.98            |
| pioneer   | 1.3   | 1.17    | 1.58   | 1.74            |

The best values of the results are in bold

Table 10  Number of patterns returned by various algorithms

| Datasets  | SQS   | GoKrimp | CSC-2  |
|-----------|-------|---------|--------|
| jmlr      | 580   | 20      | 765    |
| aslbu     | 195   | 67      | 453    |
| aslgt     | 1095  | 68      | 105    |
| auslan2   | 13    | 4       | 7      |
| context   | 138   | 33      | 39     |
| pioneer   | 143   | 49      | 134    |

Table 8 shows the number of patterns returned by the three methods on real data. For the JMLR data, GoKrimp extracted only 20 patterns, which, as shown in Table 8, are all relevant. For SQS and CSC-2, the number of selected patterns were quite large. The initial phrases
were all relevant to machine learning, as seen in Table 8. For the rest of the datasets, the size of the subset of patterns identified by CSC-2 is in between that of the GoKrimp and SQS methods. We would like to point that the size of the selected set as such without considering the relevance of the selected patterns does not have any particular significance. In Sect. 5.2.3, we discuss this again in relation to classification accuracy.

Table 11 compares the run times. Except for the JMLR text data, CSC-2 took significantly less time compared to other methods, for all the other datasets. For the JMLR data, the number of event types $M$ is high (see Table 2) and hence the candidate set $C$ for CSC-2 is high, which results in more time for calculating the $OM$ matrix. On the dataset aslgt, both SQS and GoKrimp take a lot of time because the number of events in the sequence is very large. However, this does not affect our method much.

### 5.2.3 Classification results

Here we show the classification results on the five labeled datasets in Table 2 (the JMLR data have no class labels). For each method, we select class specific patterns using the respective methods and merge them along with the set of all singletons (which are 1-node episodes) to form features/attributes for classification. Thus, for classification, each sequence is represented by a feature vector consisting of the number of occurrences of each of the (selected) patterns together with the frequencies of the singletons in the sequence. We also show the results, when only singletons are used as features. We call this method as Singletons in the tables. In [15], GoKrimp was compared with closed episode mining method, BIDE [30], and one of their two step (mining and then subset selection) methods, SEQKrimp [15,16]. They used different classifiers and found that the linear SVM classifier was giving the best results for almost all datasets. And for all the datasets, they showed that the top patterns returned by the SeqKrimp and GoKrimp were giving better classification accuracy than the top patterns returned by the BIDE algorithm. For our study, we thus use the linear SVM as the classifier using the LIBSVM package [8]. Since SeqKrimp and GoKrimp were giving similar results, we only use GoKrimp for this comparison study as SeqKrimp is computationally very intensive.

The fixed interval serial episode patterns returned by CSC-2 are used as features in two different ways. In the first method, features are the frequencies of the selected fixed interval serial episodes along with the frequencies of the singletons. In the second approach, we drop the fixed inter-event constraints from the episodes and treat them as normal serial episodes (and care is taken to avoid multiple representations of the same serial episode). The frequencies of these serial episodes along with the singleton frequencies would be the feature representation of the sequences. The first method is called as CSC-2 and the second method is called CSC-2* in the results table.
Table 12 Percentage classification accuracy achieved by various algorithms on 20 runs

| Datasets | SQS | GoKrimp | CSC-2 | CSC-2* | Singletons |
|----------|-----|---------|-------|--------|------------|
|          | Mean | σ      | Mean  | σ     | Mean       | σ    |
| aslbu    | 70.08| 0.68   | 69.3  | 1.27  | 70.27      | 0.98 |
| aslgt    | **86.12** | 0.18 | 85.55 | 0.23  | 85.9       | 0.15 |
| auslan2  | 35   | 1.23   | 32.7  | 1.6   | **35.18**  | 1.2  |
| context  | **95.33** | 0.63 | 93.98 | 0.58  | 93.75      | 1.08 |
| pioneer  | 100  | 0      | 100   | 0     | 100        | 0    |

Table 13 Confusion matrix for the context dataset with SQS

|     | 1   | 2   | 3   | 4   | 5   |
|-----|-----|-----|-----|-----|-----|
| 1   | 770 | 110 | 0   | 0   | 0   |
| 2   | 74  | 886 | 0   | 0   | 0   |
| 3   | 0   | 4   | 954 | 2   | 0   |
| 4   | 4   | 0   | 491 | 1   |     |
| 5   | 0   | 20  | 3   | 4   | 973 |

Each (i, j) entry denotes how many of class i has been classified as class j

Table 14 Confusion matrix for the context dataset with CSC-2*

|     | 1   | 2   | 3   | 4   | 5   |
|-----|-----|-----|-----|-----|-----|
| 1   | 739 | 141 | 0   | 0   | 0   |
| 2   | 92  | 868 | 0   | 0   | 0   |
| 3   | 1   | 0   | 943 | 16  | 0   |
| 4   | 0   | 0   | 2   | 997 | 1   |
| 5   | 0   | 20  | 4   | 8   | 968 |

For each dataset, the results were obtained by averaging 20 repetitions of 10-fold cross-validation. The results are shown in Table 12. The table gives the mean and standard deviation (σ) of the classification accuracy. We see that SQS marginally outperforms the other methods for the aslgt and context datasets and the two CSC-2 methods have higher accuracies for the aslbu and auslan2 datasets, respectively. In all the datasets, however, the accuracies are similar for SQS and CSC-2. In comparison, the accuracies of the GoKrimp method are slightly lower than both the SQS and CSC-2 methods, except for the pioneer data, where none of the methods seem to misclassify. We also show the confusion matrix for one of the datasets, namely the context dataset, for the three methods in Tables 13, 14 and 15. (Each (i, j)th entry in the matrix denotes how many samples of class i has been classified as class j.)

Table 16 shows the number of patterns selected by each method as features, and we see that the number of features selected by SQS is far higher than other methods. Although not shown here, we noticed that, even though the performance of SQS was only slightly better than CSC-2, the run time for the experiments was at least five times longer than that of CSC-2. On the other hand, GoKrimp selects comparatively the lowest number of patterns (except...
Table 15  Confusion matrix for the context dataset with GoKrimp

|       | 1  | 2  | 3  | 4  | 5  |
|-------|----|----|----|----|----|
| 1     | 756| 124| 0  | 0  | 0  |
| 2     | 70 | 890| 0  | 0  | 0  |
| 3     | 0  | 2  | 941| 17 | 0  |
| 4     | 0  | 0  | 2  | 984| 14 |
| 5     | 0  | 19 | 11 | 30 | 940|

Each \((i, j)\) entry denotes how many of class \(i\) has been classified as class \(j\).

Table 16  The average number of features per dataset for each method

| Datasets | SQS | GoKrimp | CSC-2 | Singletons |
|----------|-----|---------|-------|------------|
| aslbu    | 70  | 17      | 23    | 263        |
| aslg    | 1154| 418     | 923   | 94         |
| auslan2  | 15  | 0       | 6     | 23         |
| context  | 150 | 64      | 42    | 107        |
| pioneer  | 132 | 24      | 121   | 184        |

The number (except for the Singletons) is the number of non-singleton extracted patterns which represent the feature set along with the singletons.

for the context dataset) and also has the lowest accuracy among the three methods. For the auslan2 dataset, GoKrimp could not extract any pattern and hence the classification accuracy is similar to Singletons (the slight difference is due to difference in the cross-validation splits for different runs). It is also interesting to see that the Singletons method, which consists of only the 1-node frequencies, is always close to the best results. But nevertheless, the table shows that the selected patterns from different methods have contributed to the increase in accuracy.

5.3 Discussion

The results presented here show that CSC-2 is a good method for finding a subset of patterns that achieve good compression. It is also seen that the patterns that achieve good compression are also highly relevant for the problem. For the conveyor system datasets, our method was shown to perform extremely well in unearthing patterns representing the stable flow of items and achieving good compression. Both the aspects are of importance for remote monitoring of such systems as discussed earlier. On these datasets, the other algorithms, namely SQS and GoKrimp, failed to find patterns that capture the package flow, and they also could not achieve much data compression.

We have also tested our method on some real-world datasets. The CSC-2 algorithm discovered subsets of episodes that result in better data compression compared with the other methods, and our algorithm also seems faster than other algorithms for most of the datasets. The subset of patterns identified by CSC-2 is also seen to be very effective in classification scenarios.

Recall that the patterns used by CSC-2 are fixed interval serial episodes. Such episodes, as we have seen, suited the conveyor system datasets, where sequential occurrences of events follow such a fixed gap mechanism. For the other sequential datasets used for classification, such constraints may not be really relevant. But even on these datasets, CSC-2 identifies a subset of patterns that result in both data compression as well as better performance in classification. This shows that our pattern structure is not particularly restrictive and it is useful on a variety of datasets.
6 Conclusion

Frequent episodes discovered from sequential data are supposed to give us good insights into the characteristics of the data source. However, in practice, most mining algorithms output a large number of highly redundant episodes. Isolating a small subset of episodes that succinctly characterize the data is a challenging problem. In this paper, we presented an MDL-based approach for this problem. Using the interesting class of fixed interval serial episodes and a novel data encoding scheme, we presented a method to discover a subset of highly relevant episodes. In contrast to methods in Lam et al. [15, 16] and Tatti and Vreeken [27], our method achieves good data compression, while being able to work with event sequences with time stamps.

We compared our method with SQS and GoKrimp on text data and also on a number of real-world datasets which were used earlier in temporal data mining. On all these datasets, our method is good in comparison with others, in terms of both compression and run time. For the classification scenario, our method was only slightly less effective than SQS but better than GoKrimp. But we achieved it with far fewer patterns and very low run times.

In this paper, we also briefly discussed a novel application area for sequential pattern mining. This is the composable conveyor system. We presented empirical comparison of our method with that of others on three datasets from this problem domain to demonstrate both the effectiveness and efficiency of our method.

In this paper, we have not attempted any statistical analysis of our method so that we can relate the data compression to some measure of statistical significance of the pattern subset isolated by our method. This is an interesting and challenging direction to extend the work presented here. We would be exploring this in our future work.

In this paper, we assumed that the entire data sequence is available in the memory. There are also interesting applications, which involve data streams where one cannot go back to earlier data and the whole data sequence is never available at a time. Temporal data mining algorithms need to be redesigned to be effective with data streams [12, 13, 20]. There have been a few methods using MDL-based summarization schemes for streaming data as well. An extension of the Krimp method for streaming transaction data is considered in Van Leeuwen and Siebes [28], where they try to detect changes in the streams. MDL-based summarization schemes for temporal streaming sequences are proposed in Lam et al. [14]. We hope to extend the methods presented here to streaming sequences in our future work.

References

1. Achar A, Ibrahim A, Sastry PS (2013) Pattern-growth based frequent serial episode discovery. Data Knowl Eng 87:91–108
2. Achar A, Laxman S, Sastry PS (2012) A unified view of the apriori-based algorithms for frequent episode discovery. Knowl Inf Syst 31(2):223–250
3. Archer B, Shivakumar S, Rowe A, Rajkumar R (2009) Profiling primitives of networked embedded automation. In: IEEE international conference on automation science and engineering, 2009. CASE 2009. IEEE, pp 531–536
4. Burdick D, Calimlim M, Flannick J, Gehrke J, Yiu T (2005) Mafia: a maximal frequent itemset algorithm. IEEE Trans Knowl Data Eng 17(11):1490–1504
5. Calders T, Goethals B (2002) Mining all non-derivable frequent itemsets. In: Elomaa T, Mannila H, Toivonen H (eds) Principles of data mining and knowledge discovery, vol 2431 of Lecture Notes in Computer Science. Springer, Berlin, pp 74–86
6. Casas-Garriga G (2005) Summarizing sequential data with closed partial orders. In: Proceedings of the 2005 SIAM international conference on data mining, SDM 2005. SIAM, pp 380–391
7. Chandola V, Kumar V (2007) Summarization-compressing data into an informative representation. Knowl Inf Syst 12(3):355–378
8. Chang C-C, Lin C-J (2011) LIBSVM: a library for support vector machines. ACM Trans Intell Syst Technol 2:27:1–27:27. http://www.csie.ntu.edu.tw/~cjlin/libsvm
9. Geerts F, Goethals B, Mielikäinen T (2004) Tiling databases. In: Suzuki E, Arikawa S (eds) Discovery science, vol 3245, of Lecture Notes in Computer Science. Springer, Berlin, pp 278–289
10. Grünwald PD (2007) The minimum description length principle, vol 1. The MIT Press, Cambridge
11. Han J, Cheng H, Xin D, Yan X (2007) Frequent pattern mining: current status and future directions. Data Min Knowl Discov 15(1):55–86
12. Han J, Ding B (2009) Stream mining. In: Liu L, Özsu M (eds) Encyclopedia of database systems. Springer, Berlin, pp 2831–2834
13. Koper A, Nguyen H (2011) Sequential pattern mining from stream data. In: Tang J, King I, Chen L, Wang J (eds) Advanced data mining and applications, vol 7121, Lecture Notes in Computer Science. Springer, Berlin, pp 278–291
14. Lam HT, Calders T, Yang J, Mörcchen F, Fradkin D (2013) Zips: mining compressing sequential patterns in streams. In: Proceedings of the ACM SIGKDD workshop on interactive data exploration and analytics, IDEA ’13. ACM, New York, pp 54–62
15. Lam HT, Mörcchen F, Fradkin D, Calders T (2014) Mining compressing sequential patterns. Stat Anal Data Min 7(1):34–52
16. Lam HT, Mörcchen F, Fradkin D, Calders T (2012) Mining compressing sequential patterns. In: Proceedings of the 2012 SIAM international conference on data mining. SIAM, pp 319–330
17. Laxman S, Sastry PS, Unnikrishnan KP (2007) A fast algorithm for finding frequent episodes in event streams. In: Proceedings of the 13th ACM SIGKDD international conference on knowledge discovery and data mining. ACM, pp 410–419
18. Lin D-I, Kedem ZM (2002) Pincer-search: an efficient algorithm for discovering the maximum frequent set. IEEE Trans Knowl Data Eng 14(3):553–566
19. Mannila H, Toivonen H, Verkamo AI (1997) Discovery of frequent episodes in event sequences. Data Min Knowl Discov 1(3):259–289
20. Mendes LF, Ding B, Han J (2008) Stream sequential pattern mining with precise error bounds. In: Eighth IEEE international conference on data mining, 2008. ICDM’08. IEEE, pp 941–946
21. Méger N, Rigotti C (2004) Constraint-based mining of episode rules and optimal window sizes. In: Boulicaut J-F, Esposito F, Giannotti F, Pedreschi D (eds) Knowledge discovery in databases: PKDD 2004, vol 3202, Lecture Notes in Computer Science. Springer, Berlin, pp 313–324
22. Moerchen F, Fradkin D (2010) Robust mining of time intervals with semi-interval partial order patterns. In: Proceedings of the 2010 SIAM international conference on data mining. SIAM, pp 315–326
23. Pasquier N, Bastide Y, Taouil R, Lakhal L (1999) Discovering frequent closed itemsets for association rules. In: Beeri C, Buneman P (eds) Database theory—CDT’99’, vol 1540, Lecture Notes in Computer Science. Springer, Berlin, pp 398–416
24. Rissanen J (1983) A universal prior for integers and estimation by minimum description length. Ann Stat 11:416–431
25. Shivakumar S (2006) Sensor-actuator systems for automation. In: Work in progress session, IEEE real-time systems symposium. IEEE
26. Siebes A, Vreeken J, van Leeuwen M (2006) Item sets that compress. In: Proceedings of the 2006 SIAM international conference on data mining. SIAM, pp 395–406
27. Tatti N, Vreeken J (2012) The long and the short of it: summarising event sequences with serial episodes. In: Proceedings of the 18th ACM SIGKDD international conference on knowledge discovery and data mining. ACM, pp 462–470
28. Van Leeuwen M, Siebes A (2008) Streamkrimp: detecting change in data streams. In: Machine learning and knowledge discovery in databases. Springer, pp 672–687
29. Vreeken J, Van Leeuwen M, Siebes A (2011) Krimp: mining itemsets that compress. Data Min Knowl Discov 23(1):169–214
30. Wang J, Han J (2004) Bide: efficient mining of frequent closed sequences. In: Proceedings of the 20th international conference on data engineering, 2004. IEEE, pp 79–90
31. Wang J, Han J, Pei J (2003) Close++: searching for the best strategies for mining frequent closed itemsets. In: Proceedings of the ninth ACM SIGKDD international conference on knowledge discovery and data mining. ACM, pp 236–245
32. Wang J, Karypis G (2006) On efficiently summarizing categorical databases. Knowl Inf Syst 9(1):19–37
33. Witten IH, Moffat A, Bell TC (1999) Managing gigabytes: compressing and indexing documents and images, 2nd edn. Morgan and Kaufmann, San Francisco
34. Xiang Y., Jin R., Fuhry D., Dragan FF (2008) Succinct summarization of transactional databases: an overlapped hyperrectangle scheme. In: Proceedings of the 14th ACM SIGKDD international conference on knowledge discovery and data mining. ACM, pp 758–766
35. Yan X., Han J., Afshar R. (2003) Clospan: mining closed sequential patterns in large datasets. In: Proceedings of SIAM international conference on data mining. SIAM, pp 166–177

A. Ibrahim is a Ph.D. student under the guidance of Prof. P. S. Sastry in Electrical Engineering Department, Indian Institute of Science, Bangalore. He did his B.Tech. (Computer Science and Engineering) from National Institute of Technology, Calicut, and M.Tech. (Computer Science and Engineering) from Indian Institute of Technology, Kanpur. His research interests are broadly in the areas of Machine Learning and Data mining.

Shivakumar Sastry is an Associate Professor with the Department of Electrical and Computer Engineering, The University of Akron. He received his Ph.D. degree in Computer Engineering and Science from Case Western Reserve University and holds Masters Degrees in Computer Science from University of Central Florida and in Electrical Engineering from the Indian Institute of Science. His research interests are in Networked Embedded Systems, Real-time systems and Graph algorithms. Prior to joining Akron, he was a Senior Research Scientist with Rockwell Automation. He is currently a Senior Member of IEEE and serves as an Associate Editor for the IEEE Transactions on Automation Science and Engineering.

P. S. Sastry received B.Sc.(Hons.) in Physics from Indian Institute of Technology, Kharagpur, and B.E. in Electrical Communications Engineering and Ph.D. from Department of Electrical Engineering, both from Indian Institute of Science, Bangalore. Since 1986, he has been a faculty member in the Department of Electrical Engineering, Indian Institute of Science, Bangalore, where currently he is the ABB Chair Professor. He has held visiting positions at University of Massachusetts, Amherst, University of Michigan, Ann Arbor, and General Motors Research Labs, Warren, USA. He is a recipient of Sir C. V. Raman Young Scientist award, from Govt. of Karnataka, Hari Om Ashram Dr. Vikram Sarabhai research award from PRL, Ahmedabad, and Most Valued Colleague award from General Motors Corporation, USA. He is a Fellow of Indian National Academy of Engineering. Currently, he is an associate editor of IEEE Transactions on Cybernetics. His research interests include machine learning, pattern recognition, data mining and computational neuroscience.