HKU CSIS Tech Report TR-2002-11

Mining Emerging Substrings*

Sarah Chan Ben Kao C.L. Yip Michael Tang

Department of Computer Science and Information Systems
The University of Hong Kong
{*wyschan, kao, clyip, fmtang*}*@csis.hku.hk*

Abstract. We introduce a new type of KDD patterns called *emerging substrings*. In a sequence database, an emerging substring (ES) of a data class is a substring which occurs more frequently in that class rather than in other classes. ESs are important to sequence classification as they capture significant contrasts between data classes and provide insights for the construction of sequence classifiers. We propose a suffix tree-based framework for mining ESs, and study the effectiveness of applying one or more pruning techniques in different stages of our ES mining algorithm. Experimental results show that if the target class is of a small population with respect to the whole database, which is the normal scenario in single-class ES mining, most of the pruning techniques would achieve considerable performance gain.

Keywords: emerging substring, sequence mining, merged suffix tree, pruning

1 Introduction

More and more companies do business on a global sense in a multitude of languages. They would like to quickly identify the language of foreign callers and route their calls to operators who can speak the language. They are also interested in efficiently directing potential customers' e-mails to members of staff who are literate on the language.

In the above example, a spoken word can be regarded as a *sequence* of phonetic sounds while a written word can be viewed as a sequence of characters. Due to the different building blocks and rules used by the languages, some consecutive sub-elements, or *substrings*, of these sequences may only be present in one particular language, or they may tend to be more easily found in one language rather than in any others. Identifying frequent substrings specific to a particular language is of great advantage since it does not only allow us to distinguish one language from another, but also provides a good indicator of the language, or *class*, to which a given word belongs [17, 6].

In this paper we discuss the idea of *emerging substrings* for *Knowledge Discovery in Databases* (KDD) [10] and propose efficient algorithms for extracting them. Given a sequence database that is partitioned into a number of data classes, an emerging substring (ES) of a target class is a substring

*This work is supported by HK RGC under the grant HKU 7040/02E.

which occurs *more frequently* in that class rather than in the others. This requires that the ratio of an ES's support (or occurrence frequency) in that class and its support in other classes, or the growth rate, be at least equal to a certain value. To be a significant representative of a target class, the support of an ES in that class must also reach a specified threshold. ESs are important features in the field of sequence classification due to their intrinsic ability to capture significant contrasts between classes of sequences, and their potential usefulness for providing knowledge for the construction of sequence classifiers.

Jumping emerging substrings (JESs) is a specialization of emerging substrings — JESs are ESs having infinity growth rate. That is, they are substrings which can only be found in one class but not in any others. JESs capture sharper changes between data classes, however, they are more sensitive to noise than ESs having finite growth rate.

Besides language identification, the idea of emerging substrings can also be applied to many everyday life applications. For example, the order of products or services purchased by customers helps marketing executives identify customer needs, classify customers according to their buying behaviors, and set up marketing strategies [5]. Discovering specific orders of movements in stock prices provides knowledge for financial analysts to forecast stock performance [19]. Identifying frequent sequences of note duration [25] in MIDI [16] tracks helps distinguish melody tracks from non-melody ones [22, 24]. Knowing the associations between illnesses and repeated occurrences of certain genes eases the discovery of the origins of illnesses and early discovery of them [21]. Besides these, the mining of ESs is also potentially useful for protein classification, gene-coding region identification, web-log mining, content-based e-mail processing/forwarding systems, etc. The applications span a wide range of industries and are virtually endless.

The most straightforward approach of mining ESs is to enumerate all possible substrings existing in the database and count their occurrences in each class. However, a huge sequence database could contain millions of sequences and the number of substrings included in a sequence increases quadratically with the sequence length. For example, GenBank [4] had 15 million sequences in 2001 and a typical human genome has 3 billion characters. The mentioned approach is apparently impractical as it is computationally too expensive both in terms of time and working memory. This makes the mining of ESs a difficult task. Cleverer algorithms which can generate *distinct* ES candidates and filter out some of them efficiently are therefore needed.

We propose to use a *suffix tree-based framework* for mining ESs. In the suffix tree representation of a sequence, each suffix of the sequence can be viewed as a path from the root node to a terminal node in the tree, and the size of the tree in terms of the number of nodes in it is linear to the length of the sequence. Merging the suffix trees of all the sequences leads to a data structure that can effectively store all distinct substrings contained within a class of sequences in space linear to the sum of the sequence lengths. This merged suffix tree-based candidate generation approach is capable of maintaining the support count of each substring in each class, and eliminating the work of counting the occurrences of substrings which are not present in any sequence.

A sequence database often contains two or more data classes. In this paper we place our emphasis on the problem of *single-class ES mining*, i.e., to mine ESs belonging to *one* distinct class of the database. Our strategy goes like this. First, we construct a merged suffix tree from the sequences belonging to the class we target at, marking the number of occurrences of the substrings (repeated occurrences within a sequence are only counted once) in that class by associating each node with a counter. Next, we update the tree using sequences of all other classes, recording the number of occurrences of substrings using another set of counters. Then, we perform a tree traversal to extract all ESs of the target class based on the values of the support threshold, the growth rate

threshold and the support counters.

Besides recommending an ES mining model, this paper also aims at studying the effectiveness of applying various *pruning* techniques in different stages of our ES mining algorithm. We consider three basic techniques, which are namely *pruning with the support threshold*, *pruning with the growth rate threshold*, and *pruning with the length threshold*. The combined effects of these pruning methods are also covered. Experimental results show that if the target class is of a small population with respect to the whole database, which is the normal scenario in single-class ES mining, most of the pruning techniques would achieve considerable performance gain. On the other hand, if the size of the target class is close to that of the whole database, the performance gain brought about by some pruning methods may become less significant.

The scope of this work is limited to frequently occurring substrings (consecutively ordered events) within sequences. Frequently occurring sub-sequences (non-consecutively ordered events) may provide valuable insights in certain application domains (e.g., in the analysis of customers' purchase behavior [5] and web-logs [18], non-consecutive purchases or pages viewed could be useful) but these features often do not carry much sense otherwise (e.g., separate characters in a word no longer denote a word; discrete notes are no longer representative for a musical track). Moreover, mining these features would involve many more candidate sequences resulting in higher complexity as well as computational cost. Dedicated solutions have yet to be developed.

The rest of this paper is organized as follows. In Section 2, we introduce ESs, explain some basic terminologies and state a formal definition of the ES mining problem. In Section 3, we review some related work. In Section 4, we introduce the ideas of suffix trees and their merged variants, and explain their roles in our ES mining algorithms. In Section 5, we describe our ES mining algorithms and pruning techniques in details. In Section 6, we present our experimental setup, results and evaluation. Finally, we give some concluding remarks in Section 7.

2 Problem Definition

In this section we will first explain some notations used throughout this paper. Then we will define emerging substrings and jumping emerging substrings. Finally, a formal statement of the emerging substring mining problem will be given.

2.1 Basic definitions

Let $\Sigma = \{a_1, a_2, \ldots, a_m\}$ be the set of all *symbols*, or the alphabet. A *sequence* is a non-empty string (or ordered list of symbols) with finite length over Σ. The length of a sequence is the number of symbols contained in it. For example, $\mathcal{T} = \langle t_1, t_2, \ldots, t_l \rangle$ where $t_i \in \Sigma \ \ \forall i \in 1 \ldots l$, is a sequence of length l.

Given a length-k string $s = \langle s_1, s_2, \ldots, s_k \rangle$ and a length-l sequence $\mathcal{T} = \langle t_1, t_2, \ldots, t_l \rangle$. We say that s is a *substring* of $\mathcal{T}$, denoted as $s \sqsubseteq \mathcal{T}$, if $\exists i \in 1 \ldots (l-k+1)$ such that $\langle s_1, s_2, \ldots, s_k \rangle = \langle t_i, t_{i+1}, \ldots, t_{i+k-1} \rangle$. If $s \neq T$, s is a proper substring of $\mathcal{T}$, indicated as $s \sqsubset \mathcal{T}$. Furthermore, if the equality holds when $i = 1$, s is a prefix of $\mathcal{T}$. We denote this as $s \prec \mathcal{T}$. If the equality holds when $i = l - k + 1$, s is a suffix of $\mathcal{T}$. We denote this as $s \succ \mathcal{T}$.

In a training sequence database $\mathcal{D}$, each sequence $\mathcal{T}_i$ has exactly one class label $\mathcal{C}_{\mathcal{T}_i}$. Let $\mathcal{C} = \{\mathcal{C}_j : j = 1 \ldots m\}$ be the set of all class labels. For any sequence $\mathcal{T}$ not belonging to class $\mathcal{C}_k$, we say $\mathcal{T}$

belongs to $\overline{\mathcal{C}_k}$, the *complementary class* of $\mathcal{C}_k$. A database $\mathcal{D}$ with m distinct class labels can be partitioned into m disjoint datasets, such that each dataset contains sequences sharing the same class label. These datasets are represented as $\mathcal{D}_1, \mathcal{D}_2, \ldots, \mathcal{D}_m$, where $\mathcal{D}_k = \{\mathcal{T}_i : \mathcal{C}_{\mathcal{T}_i} = \mathcal{C}_k\}$. It follows that $\mathcal{D} = \{\mathcal{T}_i\} = \bigcup_{j=1}^{m} \mathcal{D}_j$. For each $\mathcal{D}_k \subset \mathcal{D}$, its *complementary dataset* $\overline{\mathcal{D}_k}$ is defined as $\mathcal{D} - \mathcal{D}_k$, and its size $|\mathcal{D}_k|$ is the number of sequences in it.

The *support count* (or simply *count*) and *support* of a non-empty string s in a dataset $\mathcal{D}_k$, denoted as $count_{\mathcal{D}_k}(s)$ and $supp_{\mathcal{D}_k}(s)$, respectively, are defined as:

$$count_{\mathcal{D}_k}(s) = |\{\mathcal{T} : \mathcal{T} \in \mathcal{D}_k \wedge s \sqsubseteq \mathcal{T}\}|$$
$$supp_{\mathcal{D}_k}(s) = \frac{count_{\mathcal{D}_k}(s)}{|\mathcal{D}_k|}$$

The *growth rate* of a non-empty string s from a dataset $\mathcal{D}_i$ to a dataset $\mathcal{D}_j$ (where $i \neq j$) is a measurement of the *change* in support between the datasets, which is defined as:

$$growthRate_{\mathcal{D}_i \to \mathcal{D}_j}(s) = \begin{cases} 0, & \text{if } supp_{\mathcal{D}_i}(s) = 0 \text{ and } supp_{\mathcal{D}_j}(s) = 0, \\ \infty, & \text{if } supp_{\mathcal{D}_i}(s) = 0 \text{ and } supp_{\mathcal{D}_j}(s) > 0, \\ \frac{supp_{\mathcal{D}_j}(s)}{supp_{\mathcal{D}_i}(s)}, & \text{otherwise.} \end{cases}$$

2.2 ES and JES

Given a support threshold $\rho_s \in [0, 1]$ and a growth rate threshold $\rho_g \in (1, +\infty)$, a non-empty string s is a *ρ_g-emerging substring* (ρ_g-ES), or simply an *emerging substring* (ES) if ρ_g is understood, of a class $\mathcal{C}_k$ if the following two conditions hold:

$$\begin{array}{ll} \textit{support condition}: & supp_{\mathcal{D}_k}(s) \geq \rho_s, \\ \textit{growth rate condition}: & growthRate_{\overline{\mathcal{D}_k} \to \mathcal{D}_k}(s) \geq \rho_g. \end{array}$$

An ES of $\mathcal{C}_k$ is also called an ES from $\overline{\mathcal{D}_k}$ to $\mathcal{D}_k$. We say that a class $\mathcal{C}_k$ is the *favorable class* of a string s, and s *favors* $\mathcal{C}_k$, if and only if s is an ES of $\mathcal{C}_k$.

A jumping emerging substring (JES) of a class $\mathcal{C}_k$ is a string s that can be found in $\mathcal{D}_k$ but not in $\overline{\mathcal{D}_k}$. That is, $supp_{\mathcal{D}_k}(s) > 0$ and $supp_{\overline{\mathcal{D}_k}}(s) = 0$. Using our notation, a JES is an ∞-ES. An ES with finite growth rate is a non-jumping ES.

To illustrate the idea of ESs and JESs, let us consider an example. Table 1(a) shows a database with two classes, each having four sequences. Table 1(b) lists out all possible substrings, their supports with respect to each class, and the growth rates of all ESs. If we set $\rho_s = 2/4$ and $\rho_g = 1.5$, there will be six ESs: Class 1 has one non-jumping ES (A) and three JESs (ABC, BCD and ABCD), whereas Class 2 has one non-jumping ES (B) and one JES (ABD).

2.3 The ES mining problem

Given a sequence database $\mathcal{D}$, the set $\mathcal{C}$ of all class labels associated with the sequences in $\mathcal{D}$, a support threshold ρ_s and a growth rate threshold ρ_g, the *ES mining problem* is to discover the set of all ρ_g-ESs present in $\mathcal{D}_j$ for each class $\mathcal{C}_j \in \mathcal{C}$.

Class 1	Class 2
ABCD	ABD
BD	BC
A	CD
C	B

(a) Datasets

Substring	Support		Growth rate of ES	
	Class 1	Class 2	Class 1	Class 2
A	2/4	1/4	2	
B	2/4	3/4		1.5
C	2/4	2/4		
D	2/4	2/4		
AB	1/4	1/4		
BC	1/4	1/4		
BD	1/4	1/4		
CD	1/4	1/4		
ABC	1/4	0/4	∞	
ABD	0/4	1/4		∞
BCD	1/4	0/4	∞	
ABCD	1/4	0/4	∞	

(b) Discovering emerging substrings (for $\rho_s = 2/4$ and $\rho_g = 1.5$)

Table 1: Example on ESs and JESs

The focus of our study is on the *single-class ES mining problem*. The definition of this sub-problem is similar to above, except that a *target class* $\mathcal{C}_k \in \mathcal{C}$ has to be specified and our goal is to discover the set of all ρ_g-ESs present in $\mathcal{D}_k \subset \mathcal{D}$. In such a case, $\overline{\mathcal{C}_k}$ is referred to as the *opponent class*.

3 Related Work

Our work on emerging substrings (ESs) is motivated by the concept of *emerging patterns* (EPs). EPs were proposed by Dong and Li [7] to capture useful contrasts between data classes, or emerging trends in timestamped databases. Under their model, a structural database consists of a collection of objects, each being a set of items (or an itemset). Each object in the database belongs to one of the classes, and it is labeled according to its class. The database can be partitioned into datasets, such that all objects in a dataset share the same class label. An itemset I is said to be contained in an object O if $I \subseteq O$, and the support of I in a dataset is defined as the fraction of objects in the dataset that contain I. Loosely speaking, an itemset is an EP if its support increases significantly from one dataset to another (the change in support is defined as the growth rate). With this property, EPs capture multi-attribute contrasts between data classes.

The EP mining problem is, for a given growth rate threshold ρ, to find all EPs with growth rate at least equal to ρ. Dong and Li are convinced that EPs with low to medium supports, such as $1-20\%$, give new insights to domain experts in many applications. However, the efficient mining of EPs with low supports is a challenging problem since (i) the Apriori property [1] no longer holds for EPs, and (ii) there are often too many EP candidates, especially for high dimensional databases. Naive algorithms are thus too costly. The authors illustrated this by an example: as there are over 350 itemsets in the PUMS dataset, the naive algorithm would have to find the supports of 2^{350} itemsets in two datasets, compute their growth rate, and determine whether they are EPs. This is apparently an impossible task.

Borders were proposed to concisely describe frequent collections of itemsets, and they play a major role in discovering EPs. A border is an ordered pair $\langle \mathcal{L}, \mathcal{R} \rangle$, where $\mathcal{L}$ and $\mathcal{R}$ are called the left-hand bound and right-hand bound, respectively. $\mathcal{L}$ is defined as the set of minimal itemsets in the collection, and $\mathcal{R}$ is defined as the set of maximal ones. The set interval of, or the collection of sets represented by, $\langle \mathcal{L}, \mathcal{R} \rangle$, is denoted as $[\mathcal{L}, \mathcal{R}] = \{Y |\ \exists X \in \mathcal{L}, \exists Z \in \mathcal{R}$ such that $X \subseteq Y \subseteq Z\}$.

It was proved that any interval-closed collection of itemsets has a unique border. In the mining of EPs, Bayardo's Max-Miner [3] is used to efficiently discover the border of the collection $\mathcal{S}$ of all frequent itemsets, for each dataset, with respect to any fixed support threshold δ in the dataset. The resultant borders are then manipulated by border-based algorithms, to mine EPs, which are in turn represented by a set of borders. The use of borders has not only eliminated the need of enumerating the elements of $\mathcal{S}$, and counting their supports in each dataset, but also provided a compact way of representing and storing the discovered EPs.

The main algorithm for discovering EPs is called MBD-LLBORDER. It takes in a special pair of borders as input, and derives a target class's EPs satisfying a support threshold θ_{min} in that class but being lower than a support threshold δ_{min} in the opponent class. The algorithm makes use of a sub-routine called BORDER-DIFF to derive the differential between a pair of borders. The EPs discovered by MBD-LLBORDER are concisely represented as the union of up to n intervals of all the borders derived by BORDER-DIFF, where n is the number of itemsets in the right-hand bound of the target class' border.

Several variants of EPs have been introduced, with Jumping EPs (JEPs) [8] being the most important one. JEPs are EPs with infinity growth rate. Since a JEP can only be found in one distinct class in the database, it is a good indicator of that class. MBD-LLBORDER can be used to mine JEPs, using two *horizontal borders* derived by HORIZON-MINER. Other variants of EPs include strong EPs [11] (which are EPs with all subsets being EPs) and the Most Expressive JEPs (MEJEPs) [13] (which are the minimal JEPs).

EPs have been used to build powerful classifiers, which are reported to outperform C4.5 [20] and CBA [14], for many datasets. CAEP (Classification by Aggregating EPs) [9] is the first proposed EP-based classifier. It adopts an eager learning approach and uses the combined power of a set of EPs to make a classification decision. Given a testing instance s, it computes a score of it for each class $\mathcal{C}_k$, which is defined as[1]:

$$
\begin{aligned}
score(s, \mathcal{C}_k) &= \sum_{\substack{e \subseteq s \\ e \in EP(\mathcal{C}_k)}} \frac{count_{\mathcal{D}_k}(e)}{count_{\mathcal{D}}(e)} \cdot supp_{\mathcal{D}_k}(e) \\
&= \sum_{\substack{e \subseteq s \\ e \in EP(\mathcal{C}_k)}} \frac{growthRate_{\overline{\mathcal{D}_k} \to \mathcal{D}_k}(e) \cdot \frac{|\mathcal{D}_k|}{|\overline{\mathcal{D}_k}|}}{growthRate_{\overline{\mathcal{D}_k} \to \mathcal{D}_k}(e) \cdot \frac{|\mathcal{D}_k|}{|\overline{\mathcal{D}_k}|} + 1} \cdot supp_{\mathcal{D}_k}(e)
\end{aligned}
$$

A JEP-classifier [13] is similar to a CAEP classifier except that only MEJEPs are used, which strengthens its resistance to noise in the training set and reduces its complexity. Unlike the CAEP and JEP classifiers, DeEPs (Decision-making by EPs) [12] is a lazy learning, instance-based classifier. It classifies an unseen object by compactly summarizing the supports of the discovered patterns in the light of maximizing the discriminating power of the patterns, and it is said to have achieved higher accuracy and efficiency than its eager learning counterparts.

Emerging substrings (ESs) in sequence databases are analogous with EPs in itemset databases. As in the case of EPs, the Apriori property does not hold for ESs either. However, techniques for extracting EPs cannot be easily modified to extract ESs, since (i) there are many more unique substrings in a sequence compared with unique subsets in an itemset, hence it is impractical to

[1] This definition is a modified version from the one presented in [9]. In the original definition, it is assumed that the populations of $\mathcal{D}_k$ and its complement $\overline{\mathcal{D}_k}$ are roughly the same. We modified the scoring function so as to relax the assumption.

transform substrings into itemsets, and find ESs by discovering single-attribute EPs in the transformed dataset, and (ii) the border approach (with modifications) can only be used to mine jumping emerging substrings but not non-jumping ones, as there are no existing algorithms that find the borders of frequent substrings. This makes the efficient mining of ESs a bigger challenge, and imposes a need for algorithms dedicated to the discovery of ESs.

We propose a suffix tree-based framework for the mining of ESs. Under this model, we only have to consider the *suffixes*, rather than substrings, of each sequence. A length-n sequence has n distinct suffixes and $O(n^2)$ substrings. Manipulating suffixes has a clear computational edge over manipulating substrings. We add the suffixes of the sequences to a *merged suffix tree*, which effectively stores all the substrings of a collection of sequences. Furthermore, by associating each node in the tree with a set of support counters and performing basic tree operations (e.g., node insertion or deletion), we can maintain each substring's count with respect to each class just by completely or partially scanning all the symbols in each suffix of each sequence once.

Like EPs, emerging substrings are good representatives of a class due to their intrinsic ability to capture distinguishing characteristics of data classes, or trends over time. ESs and Jumping ESs (JESs) would give useful knowledge for the construction of powerful sequence classifiers. Due to the gigantic number of substrings (many are redundant) involved in a sequence, it is infeasible to compute the similarity of sequences at classification time. Hence the eager learning approach seems to be a better solution than its lazy-learning counterpart. This leads us to research on the problem of efficient mining of ESs from a given training database.

4 Merged Suffix Trees

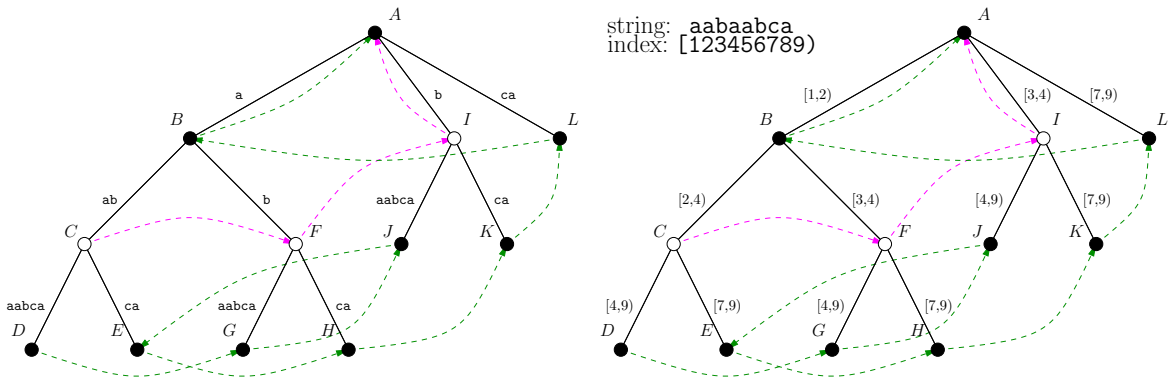
Before presenting our ES mining algorithms, let us first describe the suffix tree and merged suffix tree structures. As a convention, we use Greek letters (e.g., ω, α) to denote strings, lowercase letters (e.g., A, B) in typewriter font to denote the symbols in strings, and upper case letters (e.g., A, B) to refer to the nodes in a suffix tree or a merged suffix tree.

4.1 Suffix Trees

A *suffix tree* of a string ω is a level-compressed dictionary trie of all suffixes in ω. Given a length-n sequence, a suffix tree can be constructed using $O(n)$ time and space with online [23] or offline [15] algorithms. A suffix tree is a compact way of representing all the substrings of a given sequence.

To illustrate, let ω be the sequence AABAABCA. Figure 1(a) shows the suffix tree of ω, denoted as $ST(\omega)$. In the figure, node A is the *root* of the tree. An undirected solid edge connects a child node to its parent, and it is associated with a substring of ω. For example, the edge connecting nodes I and J in the figure is labeled AABCA, where AABCA $\sqsubset \omega$. Each node in $ST(\omega)$ is also associated with a substring of ω. We use the notation S_X to denote the substring associated with a node X. Given a node X, if we traverse the tree from the root to X and concatenate all the substrings that are associated with the edges on the traversal path, we obtain S_X. For example, with respect to Figure 1(a), S_H = ABCA, which is the concatenation of the substrings represented by edges AB, BF and FH. The root of a suffix tree is associated with the empty string, ε.

A node X in a suffix tree is called a *terminal node* if S_X is a *suffix* of ω. Otherwise, X is called a *non-terminal node*. In our example, K is a terminal node as S_K = BCA $\succ$ AABAABCA $= \omega$. A



(a) Edges represented by symbols

(b) Edges represented by index ranges

Figure 1: Suffix tree of AABAABCA

length-n sequence has $n+1$ distinct suffixes (including ε) and hence it has $n+1$ terminal nodes. In Figure 1(a), terminal nodes and non-terminal nodes are colored black and white, respectively. It can be shown that if a substring α is the longest common prefix of two suffixes of ω, there will be a non-terminal node X in the tree such that $S_X = \alpha$ [23]. To illustrate, let us consider the substring AAB which is the longest common prefix of the suffixes AABAABCA and AABCA. We can see from Figure 1(a) that there is a non-terminal node, namely node C, associating with AAB. In addition, for each internal (non-leaf) node in a suffix tree, the edges linking it to its children are associated with substrings starting with different symbols.

Each node in a suffix tree has a pointer called the *suffix link*. Suffix links are indicated by dotted arrows in the figure. The suffix link of a node X points to a node Y if and only if $S_X = xS_Y$ for some symbol x. For instance, the suffix link of node J points to node E. It can be easily checked that S_J = BAABCA = BS_E.

Any substring of a sequence ω must be a prefix of some suffix of ω. Since each non-empty suffix of ω is represented by a terminal node in $ST(\omega)$, any non-empty substring of ω can be extracted by a partial traversal of the tree, starting from the root to a terminal node. For example, the terminal node J is associated with the suffix BAABCA. The string BAAB can be obtained by traversing the tree along edge AI (which gives B) and then partially traversing along edge IJ (i.e., using only AAB in AABCA that is associated with edge IJ). In fact, one can imagine adding 4 nodes, namely X_1, X_2, X_3 and X_4, along edge IJ such that each newly formed edge corresponds to a single symbol. Figure 2 illustrates this. In our discussion, it is useful to consider these extra nodes. Since these nodes do not exist *explicitly* in the implementation of a suffix tree, we call them *implicit* nodes. Similar to an explicit node, each implicit node in $ST(\omega)$ is associated with a substring of ω.

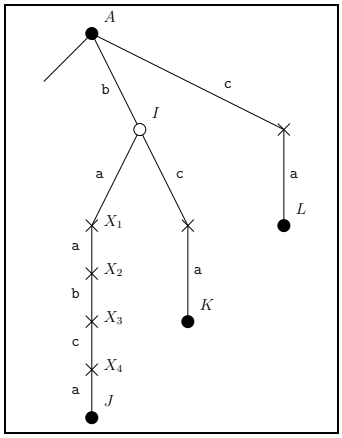


Figure 2: Part of the suffix tree of AABAABCA with implicit nodes shown as crosses

In Figure 1(a), each edge in $ST(\omega)$ is labeled with a substring of ω. Instead of recording the symbols directly, we use a left-closed right-open range $[i_{start}, i_{end})$ to represent a substring of ω for efficient use of space, as shown in Figure 1(b). Variables i_{start} and i_{end} are indices to ω. The first symbol of a sequence is given the index value 1. $\omega[i_{start}, i_{end})$ represents the substring in ω starting at position i_{start} and ending before position i_{end}. In our example, $\omega[7, 9)$ denotes the substring CA.

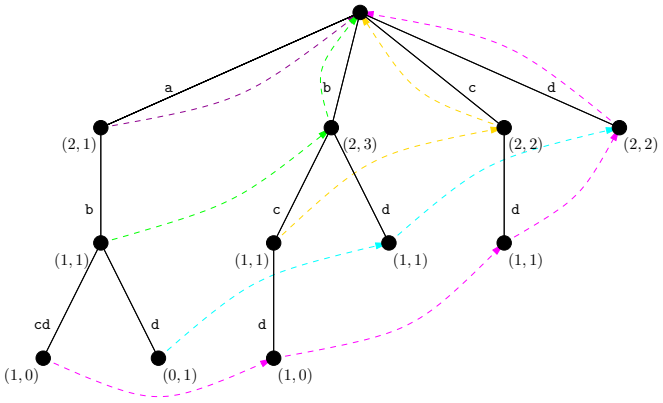
Although a length-n sequence has $O(n^2)$ substrings, there are at most $O(n)$ nodes in its suffix tree. With the index range representation, the tree can be stored in $O(n)$ space and traversed in $O(n)$ time. Hence, the suffix tree-based structure allows us to store and extract unique substrings in a compact and efficient manner. There are standard algorithms for constructing suffix trees. Interested readers are referred to [15, 23].

4.2 Merging Suffix Trees

As described above, substrings in a sequence can be uniquely represented by a suffix tree. Extending this idea a little bit, we can uniquely store all substrings present in a dataset $\mathcal{D}$ by making use of a *merged suffix tree*, or simply a *merged tree*, represented as $MT(\mathcal{D})$.

Class 1	Class 2
ABCD	ABD
BD	BC
A	CD
C	B

(a) Datasets (same as in Table 1(a))



(b) The corresponding merged suffix tree. (x, y) in each node denotes the number of occurrences of the associated substring in Class 1 and Class 2, respectively.

Figure 3: An example of merged suffix tree

Our implementation of a merged suffix tree is similar to a suffix tree except that each node is associated with *support counters*. In the simplest case where there are only two classes of sequences, each node is associated with two counters. If a node X has the counter values (c_1, c_2), it means the support count of S_X with respect to the two classes are c_1 and c_2, respectively. For example, Figure 3(a) shows a dataset with two classes, each comprising four sequences. Figure 3(b) shows the result of merging the eight suffix trees formed from the eight sequences. The counter values of the node associated with substring A are (2, 1). It means A is contained in two sequences (ABCD and A) in Class 1, and one sequence (ABD) in Class 2.

If a dataset $\mathcal{D}$ is associated with m classes, where $m > 2$, we can assign m counters to each node X in $MT(\mathcal{D})$. Each counter c_i is responsible for recording the number of occurrences of S_X in $\mathcal{D}_i$. The merged tree is therefore capable of keeping track of the support counts of each substring present in $\mathcal{D}$ in each of the m classes. In *single-class ES mining*, we are only interested in discovering ESs from a target class $\mathcal{C}_k$. Thus, only two sets of counters will be necessary: one (i.e., c_1) for the support counts in $\mathcal{D}_k$, and the other (i.e., c_2) for the support counts in $\overline{\mathcal{D}_k}$. In fact, the suffix tree $ST(\omega_i)$ for a sequence ω_i can be thought of as being a *single-sequence merged tree*, $MT(\{\omega_i\})$. If the class label of ω_i is $\mathcal{C}_k$, the k-th counter at each node in $MT(\{\omega_i\})$ would have the value 1, and all other counters would have the value 0.

We mentioned that each node in a suffix tree has a suffix link. As discussed in [23, 2], suffix links

are used to facilitate string matching, i.e., they help us to determine whether a given string is contained in the sequence represented by the suffix tree. Since our goal is to discover all ESs in a target class, such string matching tools are not required. This explains why suffix links are not maintained in our implementation of merged suffix trees.

The symbols associated with an edge are denoted by a pair of indices $[i_{start}, i_{end})$ to a vector $\mathcal{S}$, which is the concatenation of all the sequences in the database, D (i.e., $\mathcal{S} = \omega_1\omega_2 \ldots \omega_{|\mathcal{D}_k|}$). If the set $\mathcal{E}$ of substrings associating with a node Y are found to be ESs, they are represented collectively by three indices to $\mathcal{S}$, indicated as $\langle i_0, i_{start}, i_{end}\rangle$, where i_0 is the common starting position of these substrings in $\mathcal{S}$, and i_{start} and i_{end} are indices representing the incoming edge of Y. All the ESs in $\mathcal{E}$ share the same support counters (those associated with Y) and growth rate. By using index representation, all ESs can be stored concisely and retrieved conveniently.

The resultant merged tree for the $|\mathcal{D}_k|$ length-$O(n)$ sequences in a dataset $\mathcal{D}_k$ has $O(n|\mathcal{D}_k|)$ nodes. Its space complexity is therefore roughly proportional to the number of sequences and the average length of the sequences contained in $\mathcal{D}_k$. As there are $O(n|\mathcal{D}_k|)$ edges in the merged tree, all ESs found can be represented in $O(n|\mathcal{D}_k|)$ space. There are various ways of building a merged tree. Among them are the depth-first and breadth-first approaches of merging individual suffix trees. A merged tree can also be constructed directly by adopting an *Ukkonen*-like algorithm. Our ES mining algorithms build merged trees by appending suffixes of the sequences to the tree. We will describe this in details in the next section.

5 ES Mining Algorithms

In this section we give several algorithms for mining emerging substrings. After presenting a BASELINE algorithm, we will describe three basic pruning techniques which aim at boosting its performance. We will then explore the possibility of combining the power of these pruning techniques.

5.1 The BASELINE algorithm

Our BASELINE algorithm extracts the emerging substrings of a target class $\mathcal{C}_k$ from a dataset $\mathcal{D}$ in three phases:

1. **Construction Phase (C-Phase).** In this phase, a merged tree $MT(\mathcal{D}_k)$ is built from all the sequences of the target class, $\mathcal{C}_k$.

   Initially, $MT(\mathcal{D}_k)$ only contains the root node, which has counter values $(c_1, c_2) = (0, 0)$. Then, for each sequence $\omega_i \in \mathcal{D}_k$, we add each of its $|\omega_i|$ suffixes (excluding the empty string, ε) to $MT(\mathcal{D}_k)$ by path matching — we match the symbols in a suffix s_j with the symbols represented in the edges of the tree, starting from the root until either the whole suffix is matched [case 1], a leaf node is encountered [case 2], or a mismatch occurs [case 3]. During the traversal, we give an increment of 1 to the c_1 counter of each visited node unless it has previously been traversed by another suffix of ω_i.

   In case 1 [suffix s_j is fully matched], if s_j is identical to the substring represented by an *implicit* node I (i.e., $S_I = s_j$), we *explicitize* I, duplicate the counter values of I's (explicit) child node C for it, and then increase the value of I's c_1 counter by one.

   In case 2 [suffix s_j has not been fully matched and a leaf node T is met], S_T is a proper prefix

of s_j. We give T a new child node N (with counter values $(1, 0)$), and associate edge TN with the unmatched part of s_j.

In case 3 [a mismatch occurs], if the node (say, node Y) associated with the longest prefix of s_j being matched is implicit, we explicitize Y, duplicate the counter values of Y's (explicit) child node C for it, and increase the value of Y's c_1 counter by one. No matter Y is implicit or explicit, we give Y a new child node N (with counter values $(1, 0)$), and associate edge YN with the unmatched part of s_j.

Note that in case 1, we do not explicitize node I if its child node C has previously been visited by another suffix of the same sequence as s_j (so as to avoid repeated contribution of a sequence to the same counter). Similarly, in case 3, we do not increase the value of node Y's c_1 counter if the same situation arises.

Since we only deal with $\mathcal{C}_k$ sequences in this phase, all the c_2 counters in $MT(\mathcal{D}_k)$ have the value 0. In $MT(\mathcal{D}_k)$, any substring represented by an explicit node has support counts identical to those registered by the support counters of that node; any substring represented by an implicit node I has support counts identical to those registered by the support counters of the child of I. This suggests that each explicit node X in a merged tree can be thought of as *relating* not only to the substring associated with it, but also to those substrings associated with the implicit nodes present within the edge connecting it to its parent. We call all these substrings the *related substrings* of node X. In this way, most explicit nodes in a merged tree relate to multiple substrings. This explains why such data structure can compactly and uniquely store all substrings present in a dataset, and support the efficient extraction of them.

$MT(\mathcal{D}_k)$ now contains all the unique substrings present in dataset $\mathcal{D}_k$, with support counters with respect to $\mathcal{D}_k$ maintained. ■

2. **Update Phase (U-Phase).** In this phase, $MT(\mathcal{D}_k)$ is updated with all the sequences of the opponent class, ($\overline{\mathcal{C}_k}$). We denote the resultant tree by $MT'(\mathcal{D}_k)$.

   We define the *update* of a merged tree as "update the support counters of the substrings in the merged tree". This means we will increase (when necessary) the support counts of substrings which are *already* present in $MT(\mathcal{D}_k)$, but *not* introduce any substring α which can only be found in $\overline{\mathcal{D}_k}$ (i.e., $\alpha \in \overline{\mathcal{D}_k} \wedge \alpha \notin \mathcal{D}_k$) into the tree.

   The way we update $MT(\mathcal{D}_k)$ in this phase is similar to how we construct the tree in the C-Phase, except that this time we update the value of the c_2 counter (instead of c_1), and we do *not* add the child node N to the tree in cases 2 and 3 described previously. Cases 2 and 3 now become:

   In case 2 [suffix s_j has not been fully matched and a leaf node T is met], S_T is a proper prefix of s_j. No tree operation is needed.

   In case 3 [a mismatch occurs], if the node (say, node Y) associated with the longest prefix of s_j being matched is implicit, we explicitize Y, duplicate the counter values of Y's (explicit) child node C for it, and increase the value of Y's c_1 counter by one. If Y is already explicit, no tree operation is needed.

   The resultant merged tree, $MT'(\mathcal{D}_k)$, contains all the unique substrings present in dataset $\mathcal{D}_k$, with support counters with respect to both datasets $\mathcal{D}_k$ and $\overline{\mathcal{D}_k}$ maintained. ■

3. **eXtraction Phase (X-Phase).** In this phase, all ESs of $\mathcal{C}_k$ are extracted by a pre-order tree traversal on $MT'(\mathcal{D}_k)$.

   Recall that we stated in Subsection 2.2 that an ES of class $\mathcal{C}_k$ must have its support in $\mathcal{D}_k$ at least equal to the *support threshold* (ρ_s) and growth rate (from $\overline{\mathcal{D}_k}$ to $\mathcal{D}_k$) at least equal to

the *growth rate threshold* (ρ_g). In this phase, we traverse $MT'(\mathcal{D}_k)$ starting from the root in a pre-order manner. At each node X, we check its counter values to determine whether its related substrings can satisfy both the support condition (i.e., $supp_{\mathcal{D}_k}(S_X) = (c_1/|\mathcal{D}_k|) \geq \rho_s$) and the growth rate condition (i.e., $(c_1/|\mathcal{D}_k|)/(c_2/|\overline{\mathcal{D}_k}|) \geq \rho_g$).

We observe that if a sequence ω_i contains a substring μ, it will also contain any prefix ν of μ (i.e., $(\mu \sqsubseteq \omega_i) \wedge (\nu \prec \mu) \Rightarrow \nu \sqsubseteq \omega_i$). Thus, the value of each support counter of a node must be at least equal to that of the corresponding counter of any child node of it. It follows that if the substrings related to a node X are infrequent in $\mathcal{D}_k$ (i.e., $supp_{\mathcal{D}_k}(S_X) < \rho_s$), all the substrings related to any child node of X will also be infrequent in $\mathcal{D}_k$. Therefore, if we encounter a node related to substrings being infrequent in $\mathcal{D}_k$, we can ignore the subtree rooted at this node since we know we will not be able to derive any ESs from it.

With the properties of the merged tree structure, all ESs of $\mathcal{C}_k$ are mined in a complete or partial tree walk on $MT'(\mathcal{D}_k)$. ■

In short, we summarize the BASELINE algorithm as comprising the C-U-X-Phases.

5.2 With *support threshold* pruning

We observe that in the BASELINE algorithm, the c_2 counter of each substring α in $MT(\mathcal{D}_k)$ (constructed in the C-Phase) would be updated in the U-Phase as long as it is contained in some sequence in $\overline{\mathcal{D}_k}$. If α is infrequent with respect to $\mathcal{D}_k$, it will not be qualified to be an ES of $\mathcal{C}_k$ and all its descendent nodes will not even be visited in the X-Phase. This inspires us to consider adding a ρ_s-*Pruning Phase*, in which we prune all infrequent substrings (with respect to $\mathcal{D}_k$) in the merged tree right after it has been constructed. In this way, we may save some efforts and time on updating the counters of some nodes and explicitizing others, that are associated with substrings infrequent in $\mathcal{D}_k$.

ρ_s-Pruning Phase (P_s-Phase). In this phase, all substrings being infrequent in $\mathcal{D}_k$ are pruned from $MT(\mathcal{D}_k)$ by a pre-order traversal on the tree. We denote the resultant tree by $MT^s(\mathcal{D}_k)$ which now becomes the input to the U-Phase.

We traverse $MT(\mathcal{D}_k)$ starting from the root in a pre-order manner. At each node X, we check the value of its c_1 counter to determine whether its related substrings can satisfy the support condition (i.e., $supp_{\mathcal{D}_k}(S_X) = (c_1/|\mathcal{D}_k|) \geq \rho_s$) of being ESs of $\mathcal{C}_k$. If the condition fails, we remove the pointer that links X's parent node to it, and ignore all the descendent nodes of X. The subtree rooted at X is then effectively detached from $MT(\mathcal{D}_k)$. ■

We name the algorithm that comprises the C-P_s-U-X-Phases as the s-PRUNING algorithm. Its fundamental deviation from the BASELINE algorithm lies on its *earlier* use of ρ_s (from the X-Phase to the new P_s-Phase) to prune infrequent substrings in $\mathcal{D}_k$.

5.3 With *growth rate threshold* pruning

In the U-Phase of the BASELINE algorithm, as more and more sequences in $\overline{\mathcal{D}_k}$ are added to $MT(\mathcal{D}_k)$, the value of the c_2 counter of a node in the tree would get larger and larger. This implies that the support of the node's related substrings in $\overline{\mathcal{D}_k}$ is monotonically increasing, and thus the ratio of the support of the substrings in $\mathcal{D}_k$ to that in $\overline{\mathcal{D}_k}$ is monotonically decreasing. At some point, this ratio may become less than the growth rate threshold, ρ_g. When this happens, we know that the substrings have actually lost their candidature for being ESs in $\mathcal{C}_k$. This makes us

consider pruning the substrings in $MT(\mathcal{D}_k)$ as soon as they are found to be failing the growth rate requirement for ESs. By doing so, we may reduce the size of the tree and speed up the update of the tree. The idea of the ρ_g-*Update Phase* is described below.

> **ρ_g-Update Phase (U_g-Phase).** In this phase, $MT(\mathcal{D}_k)$ is updated with all the sequences of the opponent class, $\overline{\mathcal{C}_k}$. Substrings which fail to satisfy the growth rate condition are pruned from the tree. We denote the resultant tree by $MT'(\mathcal{D}_k)$.
>
> As mentioned before, an index range $[i_{start}, i_{end})$ is used to denote the symbols in an edge. For example, consider a string $\omega =$ ABCDE and a node X connected by the edge $\omega[1,5)$ to its child node Y. We have $S_Y = S_X$ ABCD (the substring associated with Y is the concatenation of the substrings associated with X and edge XY). With *growth rate threshold pruning*, we add one more entry to the index range, resulting in a $[i_{start}, i_q, i_{end})$ representation of an edge. i_q is called the *qualification point*, meaning that $[i_{start}, i_q)$ is the *disqualified region* and $[i_q, i_{end})$ is the *qualified region*, of the edge. For example, consider a node X connected by the edge $\omega[1,3,5)$ to its child node Y. Since substrings $S_X\ \omega[1,2)$ and $S_X\ \omega[1,3)$ end in the disqualified region (i.e., $[1,3)$) of XY, they are no longer related to Y. But since $S_X\ \omega[1,4)$ and $S_X\ \omega[1,5)$ end in the qualified region (i.e., $[3,5)$) of XY, they are the only substrings related to Y. We say that substrings $S_X\ \omega[1,2)$ and $S_X\ \omega[1,3)$ are effectively pruned from the tree, and they will be ignored in the X-Phase subsequently.
>
> Initially, each i_q would have the same value as the corresponding i_{start}. When the support count of a substring in $\overline{\mathcal{D}_k}$ increases, we check if it can still satisfy the growth rate condition. If not, we update the value of i_q for the associated edge accordingly. If a node has no related substrings (i.e., $i_q = i_{end}$ in the incoming edge of X) and it has no child (i.e., it is a leaf), we can simply remove it. If a node X has no related substrings and it has only one child node C, X can be pruned at the cost of updating the value of i_{start} for C and the pointer from X's parent node to its child. ■

We name the algorithm that comprises the C-U_g-X-Phases as the g-PRUNING algorithm. Its fundamental deviation from the BASELINE algorithm lies on its *earlier* use of ρ_g (from the X-Phase to the U_g-Phase) to prune substrings with low growth rate.

5.4 With *length threshold* pruning

We notice that in a dataset, longer substrings often have lower support than shorter ones. This makes them less likely to fulfill the support condition for ESs. Instead of appending these longer substrings to the tree in the U-Phase and subsequently pruning them in the P_s-Phase (in *s-pruning*), an alternative way of reducing the number of ES candidates is to limit the length of the substrings to be added to $MT(\mathcal{D}_k)$ during tree construction. We therefore introduce the ρ_l-*Construction Phase*, in which we only match *length-limited* suffixes against the tree. If a suffix consists of more than ρ_l symbols, we ignore its (ρ_l+1)th symbol up to its last symbol. The size of $MT(\mathcal{D}_k)$ built in the C_l-Phase is therefore smaller than that built in the C-Phase.

> **ρ_l-Construction Phase (C_l-Phase).** In this phase, a merged tree $MT(\mathcal{D}_k)$ is built from all the length-limited suffixes of the sequences of the target class, $\mathcal{C}_k$.
>
> The way we construct $MT(\mathcal{D}_k)$ here is similar to what we have in the C-Phase, except that instead of matching all the symbols of a suffix s_j of a sequence ω_i against the substrings in the tree, we only consider its first $min(|s_j|, \rho_l)$ symbols. ■

We name the algorithm that comprises the C_l-U-X-Phases as the l-PRUNING algorithm. Its fundamental deviation from the BASELINE algorithm lies on its *addition* of a pruning parameter, ρ_l, in the C_l-Phase, to prune relatively long substrings existing in $\mathcal{D}_k$.

5.5 Combining the power of the pruning techniques

We summarize the phases of our BASELINE algorithm and its three variants, and the algorithms' relationship, in Table 2.

Algorithm	Construction	Update	Extraction
BASELINE	$\mathcal{D}_k \rightarrow MT(\mathcal{D}_k)$	$MT(\mathcal{D}_k) + \overline{\mathcal{D}_k} \rightarrow MT'(\mathcal{D}_k)$	$MT'(\mathcal{D}_k) + \rho_s + \rho_g \rightarrow$ ESs of $\mathcal{C}_k$
s-PRUNING	$\mathcal{D}_k \rightarrow MT(\mathcal{D}_k)$ $MT(\mathcal{D}_k) + \rho_s \rightarrow MT^s(\mathcal{D}_k)$	$MT^s(\mathcal{D}_k) + \overline{\mathcal{D}_k} \rightarrow MT'(\mathcal{D}_k)$	$MT'(\mathcal{D}_k) + \rho_g \rightarrow$ ESs of $\mathcal{C}_k$
g-PRUNING	$\mathcal{D}_k \rightarrow MT(\mathcal{D}_k)$	$MT(\mathcal{D}_k) + \overline{\mathcal{D}_k} + \rho_g \rightarrow MT'(\mathcal{D}_k)$	$MT'(\mathcal{D}_k) + \rho_s \rightarrow$ ESs of $\mathcal{C}_k$
l-PRUNING	$\mathcal{D}_k + \rho_l \rightarrow MT(\mathcal{D}_k)$	$MT(\mathcal{D}_k) + \overline{\mathcal{D}_k} \rightarrow MT'(\mathcal{D}_k)$	$MT'(\mathcal{D}_k) + \rho_s + \rho_g \rightarrow$ ESs of $\mathcal{C}_k$

Table 2: Summary of phases of algorithms

As described previously, the essence of the three pruning algorithms lie on their earlier use of ρ_s or ρ_g, or introduction of ρ_l. In fact, the three pruning techniques can be coupled together easily, as they are applied in different phases. Hence, four variants of pruning algorithms are resulted: the sg-PRUNING algorithm combines s-PRUNING with g-PRUNING; the ls-PRUNING algorithm combines l-PRUNING with s-PRUNING; the lg-PRUNING algorithm combines l-PRUNING with g-PRUNING; and the lsg-PRUNING algorithm combines l-PRUNING with s-PRUNING and g-PRUNING. We will explore the effects of all these combined techniques in the next section.

6 Performance Evaluation

We performed a number of experiments comparing the performance of our ES mining algorithms. Our goal is to assess the effectiveness of each of the three proposed pruning techniques, as well as the combined applications of two or all of them. In this section we present the major findings of our experiments.

We used "CI3" [25] as our test database. Each CI3 (Coarse Interval, mapping 3 intervals to 1) sequence consists of an ordered list of numbers denoting the interval sizes between consecutive notes within a musical track. From 750 MIDI [16] files, we manually extracted melody and non-melody tracks and converted the tracks into CI3 sequences for ES mining. 843 of the non-empty CI3 sequences belong to melody tracks, and the rest (6,742 sequences) of them are from non-melody tracks. The characteristics of the two datasets are tabulated in Table 3. In our evaluation, melody (the minority class) was the target class and non-melody (the majority class) was the opponent class. Our task was to efficiency discover the ESs of the target class that satisfy both the support threshold, ρ_s, and the growth rate threshold, ρ_g.

Dataset	No. of sequences	Avg. sequence length	Max. sequence length	No. of distinct symbols
Melody	843 (11.1%)	331.0	1,085	29
Non-melody	6,742 (88.9%)	274.9	2,891	61

Table 3: Datasets

In our experiments, we used different values of ρ_s (i.e., 0.1%, 0.5%, 1.0%, 1.5% and 2.0%) and ρ_g (i.e., 2, 3, 4, 5 and ∞). Before examining the runtimes of the algorithms, let us take a look at the number of ESs with different combinations of values of these two thresholds. Table 4(a) shows the number of non-jumping ESs and Table 4(b) shows the number of JESs. Observe that the number of non-jumping ESs decreases when the value of either ρ_s or ρ_g increases, while the number of JESs decreases when the value of ρ_s increases but it is not influenced by the choice of ρ_g. The column

min_{count} in Table 4(b) denotes the minimum number of occurrences for a substring to be frequent in the melody dataset, with respect to a certain value of ρ_s. It is worth noting that when $\rho_s = 0.1\%$, a substring is frequent as long as it exists in the melody dataset since $min_{count} = \lceil 843 \times 0.1\% \rceil = 1$. This accounts for the enormous number of both kinds of ESs when ρ_s attains this value.

$\rho_s \setminus \rho_g$	2	3	4	5
0.1%	10,588,746	10,503,828	9,336,484	9,331,844
0.5%	17,619	13,850	9,931	7,264
1.0%	8,195	6,013	3,961	2,535
1.5%	5,249	3,701	2,294	1,321
2.0%	3,799	2,573	1,552	819

(a) Non-jumping ESs

ρ_s	min_{count}	Any value of ρ_g
0.1%	1	34,531,736
0.5%	5	89
1.0%	9	1
1.5%	13	0
2.0%	17	0

(b) JESs

Table 4: Number of non-jumping ESs and JESs

Table 5 states the size (in terms of the number of nodes) of the merged tree after different phases of various algorithms, with different values for the thresholds. Though not covering all values of the thresholds used in the experiments, the table gives a useful overview of the relative tree sizes and extent of substring pruning resulted from the algorithms.

Algorithm	$MT(\mathcal{D}_k)$	$MT^s(\mathcal{D}_k)$	$MT'(\mathcal{D}_k)$
BASELINE	416,151	/	542,094
g-PRUNING ($\rho_g = 2$)	416,151	/	510,764
g-PRUNING ($\rho_g = \infty$)	416,151	/	330,865
s-PRUNING ($\rho_s = 1.0\%$)	416,151	10,945	10,963
sg-PRUNING ($\rho_s = 1.0\%, \rho_g = 2$)	416,151	10,945	8,599
sg-PRUNING ($\rho_s = 1.0\%, \rho_g = \infty$)	416,151	10,945	2
l-PRUNING ($\rho_l = 500$)	416,151	/	542,080
l-PRUNING ($\rho_l = 100$)	389,671	/	506,378
l-PRUNING ($\rho_l = 50$)	261,839	/	357,327

Table 5: Number of nodes in the merged tree after different phases

The merged tree $MT(\mathcal{D}_k)$ constructed in the C-Phase for the algorithms without the "l" parameter is the same, and possesses 416,151 nodes. The BASELINE algorithm expands the tree to the size of 542,094 nodes after the U-Phase. With node pruning in the U_g-Phase, the g-PRUNING algorithm limits $|MT'(\mathcal{D}_k)|$ to 510,764 (-5.8%) when $\rho_g = 2$, and to 330,865 (-39.0%) when $\rho_g = \infty$. With s-PRUNING and $\rho_s = 1.0\%$, the sizes of the input and output trees of the U-Phase shrink to 10,945 (-97.4%) and 10,963 (-98.0%), respectively. sg-PRUNING shares the same size of $MT^s(\mathcal{D}_k)$ (10,963) produced by s-PRUNING, and further reduces $|MT'(\mathcal{D}_k)|$ to 8,599 (-98.4%) when $\rho_g = 2$, and a distinguishing 2 (-100.0%) when $\rho_g = \infty$. With l-PRUNING, the sizes of both $MT(\mathcal{D}_k)$ and $MT'(\mathcal{D}_k)$ are smaller than those in the BASELINE algorithm, and the reduction is greater when the value of ρ_l is tuned smaller. We will relate the performance of the algorithms to their ability to remove ES candidates from the merged tree.

In the rest of this section, we will first discuss the performance of the algorithms that make use of a single pruning technique, and then evaluate those using multiple pruning techniques. All the runtimes were taken on a Pentium III Xeon 700 machine with 4 GB RAM.

6.1 Performance of the BASELINE algorithm

Figure 4 shows the runtimes of the BASELINE algorithm for different values of ρ_s and ρ_g. For each value of ρ_s, the mining time decreases slightly with an increase in the value of ρ_g, due to

the reduction in the number of ESs and hence in the time spent in extracting and storing ESs in the X-Phase. The difference is very small since most of the mining time is spent in constructing $MT(\mathcal{D}_k)$ with the melody sequences in the C-Phase ($13-14\%$) and updating the tree with the non-melody sequences in the U-Phase ($74-85\%$). When $\rho_s = 0.1\%$, the runtime for $\rho_g = \infty$ is much less than those for other values of ρ_g, since it only has to mine JESs while the latter cases have to spend much time on extracting and storing around ten million non-jumping ESs (as seen in Table 4(a)).

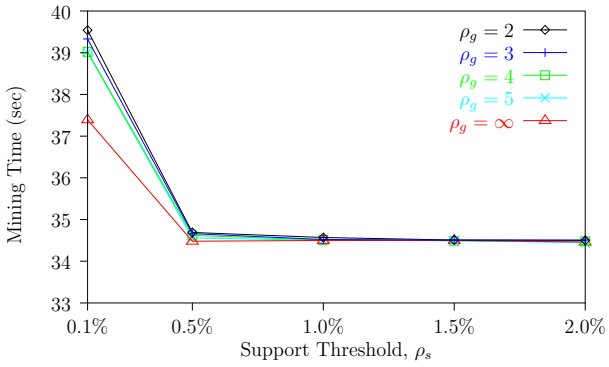


Figure 4: Runtimes of the BASELINE algorithm

For each value of ρ_g, the mining time decreases by a clear margin (from $37.4-39.5$ secs to $34.4-34.7$ secs) when the value of ρ_s is raised from 0.1% to 0.5%, and remains rather stable (in fact, decreases very mildly) when the value of ρ_s increases further. This is due to the sharp decrease in the number of ESs from the order of millions when $\rho_s = 0.1\%$ to the order of thousands (for non-jumping ESs) or tens (for JESs) when $\rho_s \geq 0.5\%$.

6.2 Performance of the *s*-PRUNING algorithm

Figure 5 gives the runtimes of the s-PRUNING algorithm for different values of ρ_s and ρ_g. For comparison, the curve for the BASELINE algorithm when $\rho_g = 2$ is also shown. The runtime of this algorithm is much shorter than the corresponding runtime of the BASELINE algorithm, except when $\rho_s = 0.1\%$.

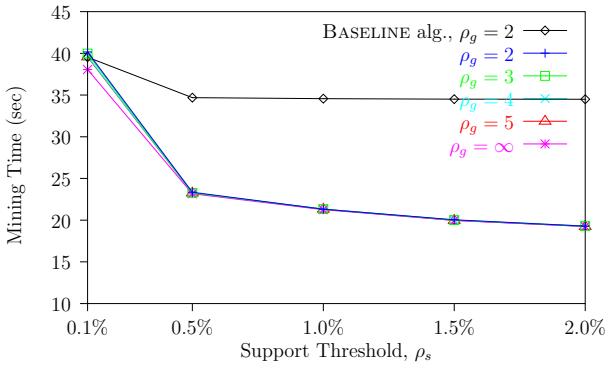


Figure 5: Runtimes of the s-PRUNING algorithm

As we discussed in Subsection 5.2, the s-PRUNING algorithm aims to reduce the mining time by shrinking the tree size by removing infrequent substrings (with respect to $\mathcal{D}_k$) in the extra P_s-Phase, and thus spending less time on tree update in the U-Phase. When $\rho_s = 0.1\%$, no substrings in

$MT(\mathcal{D}_k)$ are infrequent with respect to $\mathcal{D}_k$, hence no nodes can be removed from the tree and the time taken in the U-Phase remains the same. Besides, the algorithm also suffers the cost (about 0.64 secs) of performing a complete tree walk on $MT(\mathcal{D}_k)$ in the P_s-Phase, leading to a mild increase in the overall runtime.

However, when $\rho_s \geq 0.5\%$, the algorithm meets its objectives. A large proportion (e.g., 98.0% for $\rho_s = 1.0\%$, as seen in Table 4(a)) of the tree can be discarded in the P_s-Phase, and the U-Phase is speeded up to a large extent (by $39 - 52\%$). As the U-Phase is the most time-consuming phase, the total mining time is speeded up accordingly (by $33 - 44\%$). The greater the value of ρ_s, the greater the speedup. Similar to the BASELINE algorithm, the value of ρ_g almost does not affect the runtime of the s-PRUNING algorithm.

6.3 Performance of the g-PRUNING algorithm

Figure 6 gives the runtimes of the g-PRUNING algorithm for different values of ρ_s and ρ_g. Again, for comparison, the curve for the BASELINE algorithm when $\rho_g = 2$ is also shown. This algorithm is slightly slower than the BASELINE algorithm, except when $\rho_g = \infty$.

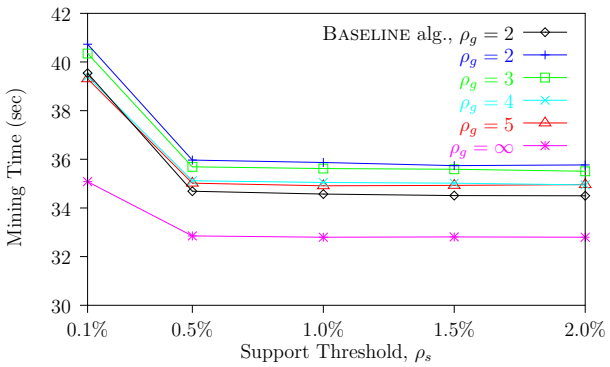


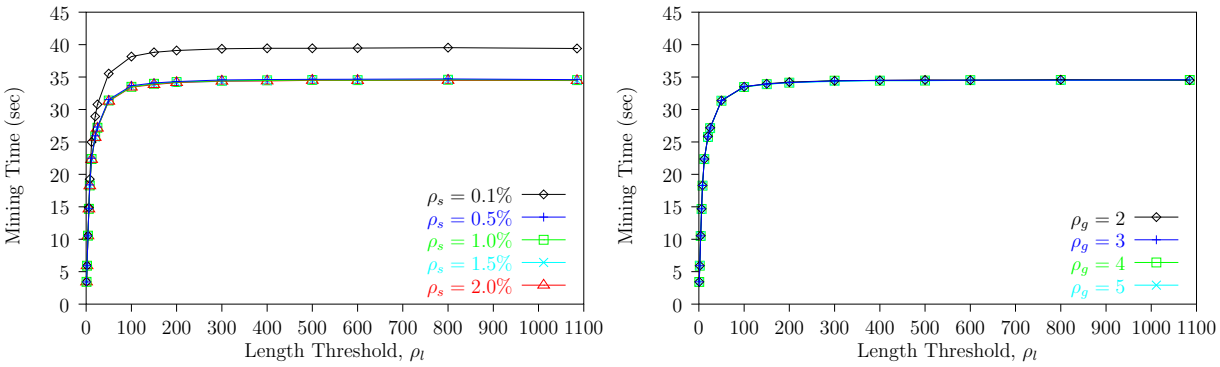
Figure 6: Runtimes of the g-PRUNING algorithm

The g-PRUNING algorithm aims to reduce the tree updating and overall mining time by computing the updated growth rate of ES candidates and removing the disqualified (with respect to ρ_g) ones from the merged tree in the U_g-Phase. Our experiments showed that when the value of ρ_g is finite, not many nodes can be pruned in the U_g-Phase. For example, as indicated in Table 5, only 5.8% of the nodes are removed when $\rho_g = 2$. Even when ρ_g increases to 5, less than 15% of the nodes can be pruned. As shown in our experiments, when $\rho_g = 2$ to 5, the time spent on growth rate checking and path compression cannot be compensated by the time saved by the reduction in tree size and tree updating efforts. Moreover, there is an extra node creation overhead for g-PRUNING as it requires an additional index for each edge of the tree. The overall runtime exceeds that of the BASELINE algorithm by about $1.3 - 3.7\%$. The greater the value of ρ_g, the smaller the slowdown.

When $\rho_g = \infty$, only JESs are to be discovered. If a substring is found to be present in any non-melody sequence, it can be pruned right away. Hence, candidature checking is simplified and more nodes can be removed from the tree. As we have seen in Table 5, 39.0% of the tree can be discarded when the value of ρ_g is infinite. This significantly speeds up tree update and contributes to the 4.9% drop in runtime compared with the BASELINE algorithm, when $\rho_s \geq 0.5\%$. Like the BASELINE algorithm, the value of ρ_s almost does not affect the runtime of the g-PRUNING algorithm, except when $\rho_s = 0.1\%$, when the runtime is significantly longer.

6.4 Performance of the l-PRUNING algorithm

Figure 7 shows the runtimes of the l-PRUNING algorithm with the value of the length threshold, ρ_l, ranging from 1 to 1085 (the length of the longest melody sequence). In Figure 7(a), ρ_g is fixed at 2 and the curves represent varying values of ρ_s. In Figure 7(b), ρ_s is fixed at 1.0% and the curves represent varying values of ρ_g. Like the BASELINE algorithm, different combinations of the values of ρ_s and ρ_g result in almost identical performance of the l-PRUNING algorithm at any value of ρ_l, except when $\rho_s = 1.0\%$. When $\rho_l = 1085$, this algorithm is same as the BASELINE algorithm.



(a) With fixed growth rate threshold ($\rho_g = 2$) (b) With fixed support threshold ($\rho_s = 1.0\%$)

Figure 7: Runtimes of the l-PRUNING algorithm

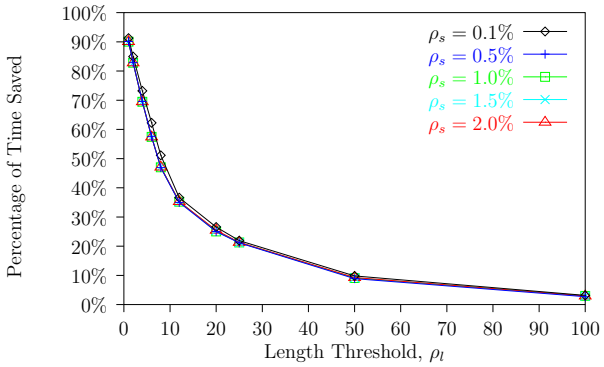


Figure 8: Percentage of time saved by l-PRUNING (with fixed growth rate threshold, i.e., $\rho_g = 2$)

The l-PRUNING algorithm was designed to build a smaller $MT(\mathcal{D}_k)$ by constructing the tree with length-limited suffixes in the C_l-Phase, thereby saving time in tree construction and update. Figure 8 shows the percentage of time saved by the algorithm with $\rho_g = 2$ and varying values of ρ_s. When $\rho_l > 100$, less than 3.0% of time can be saved. This is because the size of $MT(\mathcal{D}_k)$ built is not significantly smaller than that built in the C-Phase (of the BASELINE algorithm). We have seen from Table 5 that when $\rho_l = 100$, the tree $MT(\mathcal{D}_k)$ built in these two phases has 389,671 and 416,151 nodes, respectively. The difference in tree size is just 6.4%. As the value of ρ_l reduces further, the mining time reduces more sharply.

As discussed previously, applying l-PRUNING to remove ES candidates may lead to a loss of ESs. Figure 9 shows the percentage of ES loss with ρ_l ranging from 1 to 50. Except when $\rho_s = 0.1\%$, where a huge portion of ESs are lost even when $\rho_l = 50$, there is no ES loss unless ρ_l is less than 19 (for $\rho_s = 0.5\%$), 17 (for $\rho_s = 1.0\%$), 13 (for $\rho_s = 1.5\%$) or 11 (for $\rho_s = 2.0\%$). If we compare this figure with Figure 8, we can easily recognize the benefit of performing l-PRUNING. For example,

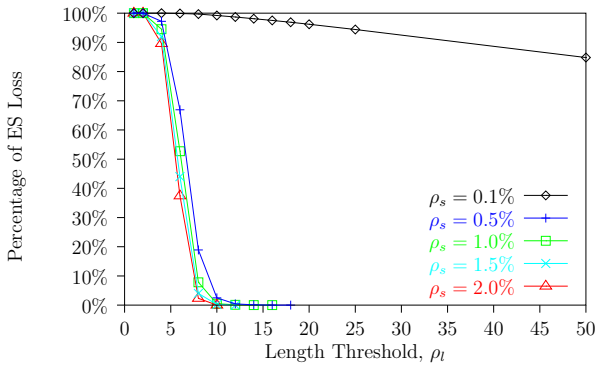


Figure 9: Percentage of ES loss due to l-PRUNING (with fixed growth rate threshold, i.e., $\rho_g = 2$, and only non-zero values shown)

taking $\rho_s = 0.5\%$, l-PRUNING can reduce the mining time by over 25% without causing any ES loss. The benefit is greater with larger values of ρ_s and smaller values of ρ_l. Several factors may influence the choice of ρ_l, such as the average sequence length and the maximum length among the ESs found in a previous or similar (in terms of application domain) ES mining activity. One may prefer to sacrifice a small proportion of ESs for performance speedup, hence picking a smaller value for the threshold. One may also determine a suitable value of ρ_l by taking a sample of each class' sequences and performing an ES mining exercise on the target class.

6.5 The combined pruning performance

Having examined the effectiveness of each individual pruning technique, let us look at the combined pruning performance.

Figure 10 presents the runtimes of the sg-PRUNING algorithm for different values of ρ_s and ρ_g. We have discussed in Subsection 6.2 that the s-PRUNING algorithm achieves a great reduction in runtime when $\rho_s \geq 0.5\%$, and in Subsection 6.3 that the g-PRUNING algorithm performs slightly worse than the BASELINE algorithm when the value of ρ_g is finite. Combining the above two pruning techniques, the sg-PRUNING algorithm is able to achieve lower runtimes than any of these three algorithms when $\rho_s \geq 0.5\%$ and for any magnitude of ρ_g. Its runtime decreases apparently when either the value of ρ_s or ρ_g increases.

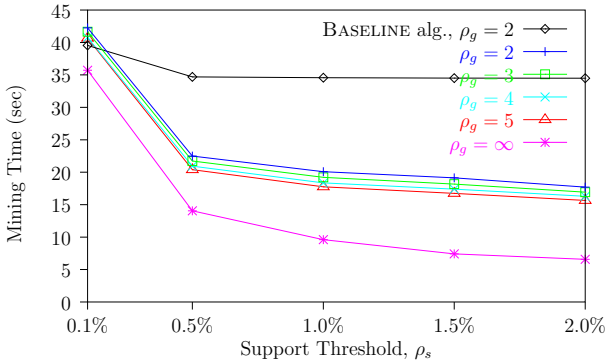


Figure 10: Runtimes of the sg-PRUNING algorithm

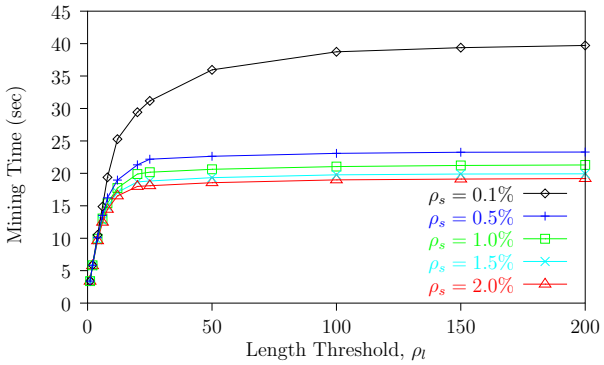


Figure 11: Runtimes of the ls-PRUNING algorithm (with fixed growth rate threshold, i.e., $\rho_g = 2$)

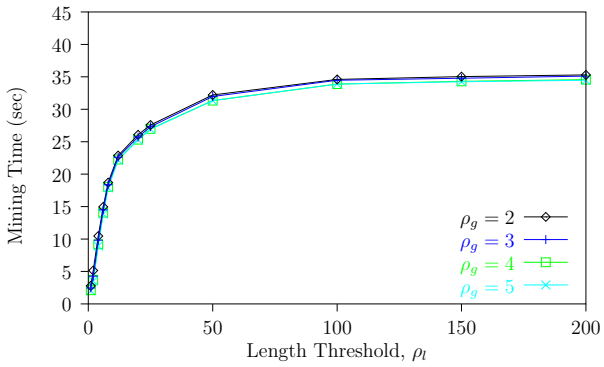


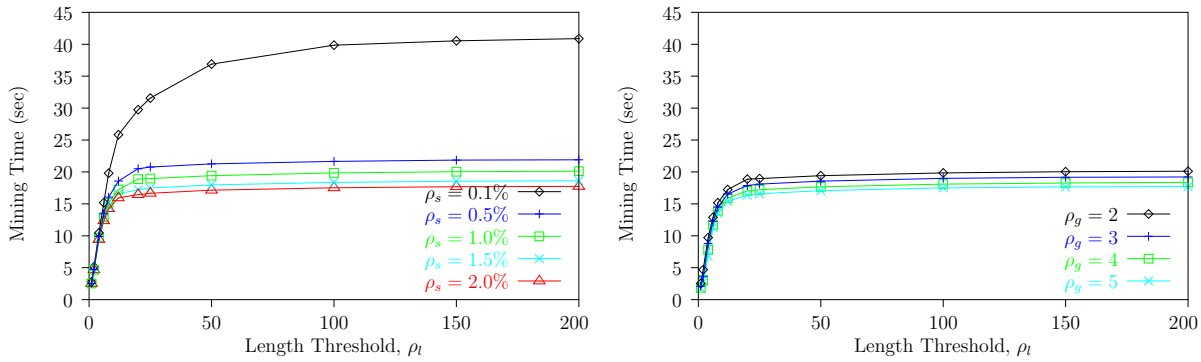
Figure 12: Runtimes of the lg-PRUNING algorithm (with fixed support threshold, i.e., $\rho_s = 1.0\%$)

Figure 11 presents the runtimes of the ls-PRUNING algorithm with $\rho_g = 2$ and varying values of ρ_s and ρ_l (only $\rho_l \leq 200$ is shown). In fact, when $\rho_l = 1085$, this algorithm is same as the s-PRUNING algorithm. Like the l-PRUNING algorithm, the runtime of the ls-PRUNING algorithm drops when the value of ρ_l becomes small. The drop is less sharp than the former since the time achieved by s-PRUNING (consider the level region of the curve) is already much shorter than that achieved by the BASELINE algorithm. Hence, applying l-PRUNING on top of the s-PRUNING algorithm is less advantageous than applying it on top of the BASELINE algorithm.

Figure 12 gives the runtimes of the lg-PRUNING algorithm with $\rho_s = 1.0\%$ and varying values of ρ_g and ρ_l (only $\rho_l \leq 200$ is shown). When $\rho_l = 1085$, this algorithm is indeed identical to the g-PRUNING algorithm. Like the l-PRUNING algorithm, the runtime of the lg-PRUNING algorithm drops when the value of ρ_l becomes small. We described in Subsection 6.3 that g-PRUNING slows down performance when the value of ρ_g is finite, but when l-PRUNING is coupled with it, its performance is comparable to the l-PRUNING algorithm for small values (e.g., 50) of ρ_g. This is mainly due to the decreased time overhead in node creation, and the observation is clearer as the value of ρ_g increases.

Figure 13 gives the runtimes of the lsg-PRUNING algorithm. In Figure 13(a), ρ_g is fixed at 2 and the curves represent varying values of ρ_s. In Figure 13(b), ρ_s is fixed at 1.0% and the curves represent varying values of ρ_g. When $\rho_l = 1085$, this algorithm is identical to the sg-PRUNING algorithm. For any values of ρ_l and ρ_g, and with $\rho_s \geq 1.0\%$, the lsg-PRUNING algorithm achieves the least runtime among all the algorithms described.

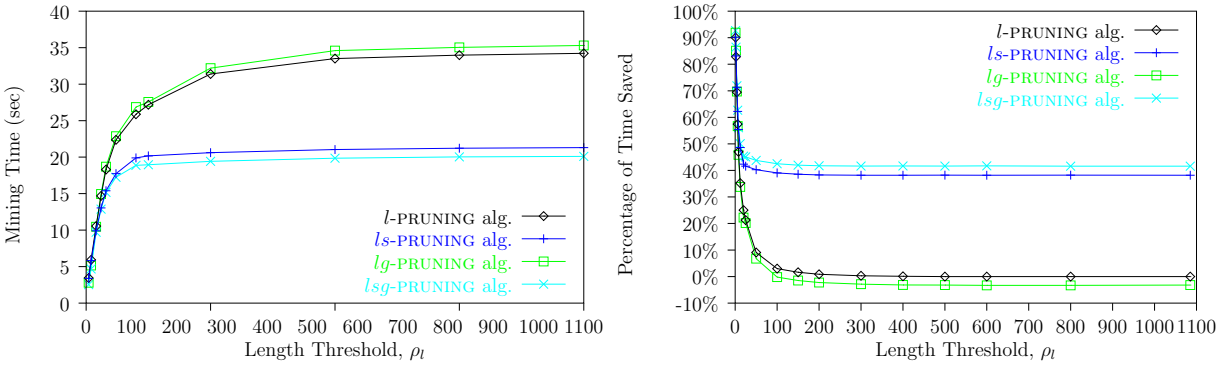
The performance of all our ES mining algorithms on the melody dataset, with the general setting



(a) With fixed growth rate threshold ($\rho_g = 2$)

(b) With fixed support threshold ($\rho_s = 1.0\%$)

Figure 13: Runtimes of the *lsg*-PRUNING algorithm



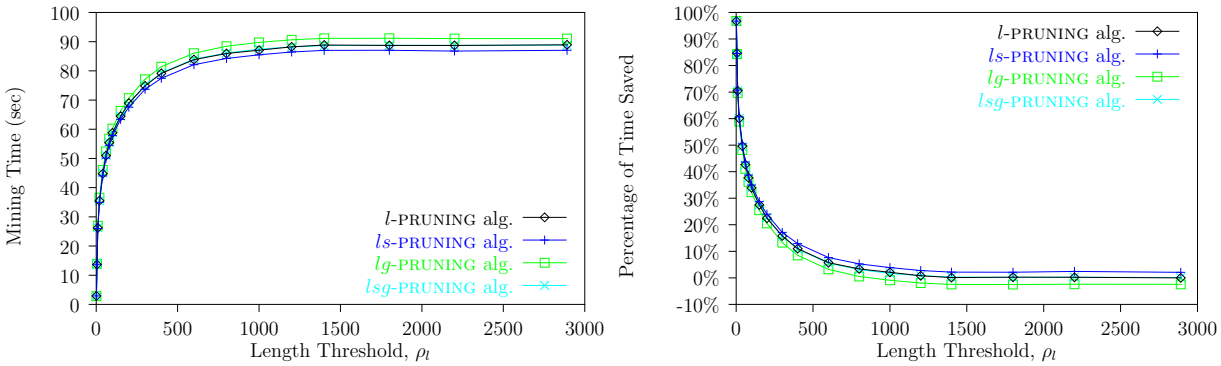
(a) Runtimes

(b) Percentage of time saved (compared with the BASELINE algorithm)

Figure 14: Runtimes and percentage of time saved for all the algorithms (with fixed support threshold, i.e., $\rho_s = 1.0\%$, and fixed growth rate threshold, i.e., $\rho_g = 2$) on the melody dataset. The rightmost point on the curve for an algorithm denotes the performance of the corresponding algorithm that does not have the "*l*" parameter.

of $\rho_s = 1.0\%$ and $\rho_g = 2$, is summarized in Figure 14. Figure 14(a) shows the algorithms' runtimes, while Figure 14(b) shows the percentage of time saved compared with the BASELINE algorithm (a negative value means being slower than the BASELINE algorithm). The rightmost point (at $\rho_l = 1085$) on the curve for a labeled algorithm denotes the performance of the corresponding algorithm upon which *l*-PRUNING has not been applied. Compared with the BASELINE algorithm, the *s*-PRUNING algorithm is faster by 38.2%; the *g*-PRUNING algorithm is slower by 3.3% but when it is coupled with *s*-PRUNING, a 41.6% reduction in time is registered. Taking $\rho_l = 18$ (no ES loss), *lg*-PRUNING, *l*-PRUNING, *ls*-PRUNING and *lsg*-PRUNING achieve an overall 22%, 25%, 42% and 45% reduction in mining time, respectively. It has been shown in the previous subsections that when the value of ρ_s is larger, the algorithms that involve *s*-PRUNING can cause more ES candidates to be pruned and hence further decrease the overall ES mining time. Similarly, when the value of ρ_g is larger, the performance of the algorithms that involve *g*-PRUNING will be boosted.

Besides speeding up the mining of ESs, another important contribution of the pruning techniques introduced in this paper is reducing the number of nodes in the merged tree both before and after tree update. This effectively limits the memory requirements of the tree structure.



(a) Runtimes

(b) Percentage of time saved (compared with the BASELINE algorithm)

Figure 15: Runtimes and percentage of time saved for all the algorithms (with fixed support threshold, i.e., $\rho_s = 1.0\%$, and fixed growth rate threshold, i.e., $\rho_g = 2$) on the non-melody dataset

All the experimental results described so far are for single-class ES mining on the minority (melody) class. To analyze the case of mining ESs of the majority (non-melody) class, we reversed the roles of the datasets (non-melody became the target class; melody became the opponent class) and conducted the same sets of experiments. Figure 15 presents the performance of the ES mining algorithms on the non-melody dataset, with $\rho_s = 1.0\%$ and $\rho_g = 2$. Again, the g-PRUNING algorithm is a bit (2.4%) slower than the BASELINE algorithm, while the s-PRUNING and sg-PRUNING algorithms are faster than that. Despite achieving a 64% (resp. 72%) cutdown in the time spent in the U-Phase (resp. U_g-Phase), the s-PRUNING (resp. sg-PRUNING) algorithm only results in a 4.7% (resp. 2.5%) reduction in the total runtime, compared with the BASELINE algorithm. This is because with the majority class being the target class, over 93% of the time is spent in tree construction in the C-Phase. Thus, the time saved in the U-Phase does not benefit the overall runtime much. Also notice that unlike the case of mining ESs of the melody class when there are not as many nodes, sg-PRUNING now performs worse than s-PRUNING due to the huge time overhead in creating one more index to represent each edge of the tree. With the specified values of ρ_s and ρ_g, the non-melody class has 2,958 non-jumping ESs and 1,360 JESs. Taking ρ_l as 382, the length of the longest ES, the maximum proportion of time saved by the l-PRUNING algorithm is about 12%. When $\rho_l = 382$, the reduction in the size of $MT(\mathcal{D}_k)$ is only 1.3%. Most of the time is saved in the C_l-Phase; the amount of time spent in the U-Phase and the X-Phase is almost unaffected.

In fact, the tree construction time can be shortened. Besides using the current method of suffix path matching, we also experimented with constructing the merged tree by building a suffix tree from each sequence and merging it to the merged tree by a depth-first-based merging approach. This speeds up the C-Phase of the BASELINE algorithm from 81.8 secs to 36.1 secs (i.e., by 55.9%). This merging approach can also be applied to the U-Phase, when sequences of the opponent class are used to update the tree. By speeding up tree construction and update, the amount of time saved by the s-PRUNING and g-PRUNING would become more significant with respect to the overall runtime. However, since the target class is the majority, the effect of l-PRUNING mostly acts on the C_l-Phase. It follows that if the tree construction time is shortened, the proportion of time saved by l-PRUNING would become less apparent.

7 Conclusions

In this paper we proposed emerging substrings (ESs) as a new type of knowledge patterns, with Jumping Emerging Substrings (JESs) being an important specialization of them. ESs in sequence databases are valuable because of their ability to capture distinguishing characteristics of data classes, and potential usefulness for the construction of powerful sequence classifiers. We introduced the ES mining problem and the sub-problem of single-class ES mining. The latter is the focus of this paper, and it refers to the discovery of ESs from a target class, among all the classes in the database. We proposed a suffix tree-based framework for efficient mining of ESs, and three basic pruning techniques for removing ES candidates, namely, *s*-PRUNING (*pruning with the support threshold*), *g*-PRUNING (*pruning with the growth rate threshold*) and *l*-PRUNING (*pruning with the length threshold*). We performed a series of experiments to evaluate the effectiveness of applying one or more pruning techniques in different stages of our ES mining algorithm. Experimental results showed that if the target class is of a small population with respect to the whole database, which is the normal scenario in single-class ES mining, most of the pruning techniques would achieve considerable performance gain. Furthermore, we demonstrated that *s*-PRUNING would be more effective when the value of ρ_s is larger; *g*-PRUNING would be more effective when the value of ρ_g is larger; *l*-PRUNING would be more effective when the value of ρ_l is smaller, but a loss of ESs may be resulted. Another benefit brought about by the pruning techniques is that by shrinking the size of the merged suffix tree, memory usage can be lessened.

On the other hand, if the target class is the majority of the sequence database, some of the pruning techniques may not be useful. If a merged suffix tree can be built efficiently from sequences of the target class, *s*-PRUNING and *sg*-PRUNING would be effective, while *l*-PRUNING may only bring a small cutdown on the total mining time. If tree construction is slow, *l*-PRUNING would be effective but the time reduction brought about by *s*-PRUNING and *sg*-PRUNING may not be obvious. One possible direction to speed up the mining of ESs of a majority class would be to discover more efficient methods to construct and update the merged suffix tree.

References

[1] Rakesh Agrawal and Ramakrishnan Srikant. Fast algorithms for mining association rules in large databases. In Jorge B. Bocca, Matthias Jarke, and Carlo Zaniolo, editors, *Proc. of the 20th International Conference on Very Large Data Bases (VLDB'94)*, pages 487–499, Santiago de Chile, Chile, 1994. Morgan Kaufmann.

[2] Alberto Apostolico. The myriad virtues of subword trees. In Alberto Apostolico and Zvi Galil, editors, *Combinatorial Algorithms on Words*, volume 12 of *NATO ASI series. Series F, Computer and System Sciences*. Springer-Verlag, 1984.

[3] Roberto J. Bayardo. Efficiently mining long patterns from databases. In *Proc. of the ACM SIGMOD International Conference on Management of Data (SIGMOD'98)*, pages 85–93, Seattle, WA USA, 1998. ACM.

[4] Dennis A. Benson, Ilene Karsch-Mizrachi, David J. Lipman, James Ostell, Barbara A. Rapp, and David L. Wheeler. GenBank. *Nucleic Acids Research*, 28(1):15–18, 2000.

[5] Michael J. A. Berry and Gordon S. Linoff. *Data Mining Techniques for Marketing, Sales and Customer Support*. John Wiley & Sons, 1997.

[6] W. B. Cavner and J. M. Trenkle. N-gram based text categorization. In *Proc. of the Third Annual Symposium on Document Analysis and Information Retrieval*, pages 161–169, 1994.

[7] Guozhu Dong and Jinyan Li. Efficient mining of emerging patterns: Discovering trends and differences. In Surajit Chaudhuri and David Madigan, editors, *Proc. of the 5th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 43–52, San Diego, California, USA, 1999. ACM.

[8] Guozhu Dong, Jinyan Li, and Xiuzhen Zhang. Discovering jumping emerging patterns and experiments on real datasets. In Joseph Fong, editor, *Proc. of the 9th International Database Conference on Heterogeneous and Internet Databases*, pages 155–168, Hong Kong, 1999. ACM Hong Kong Chapter, City University of Hong Kong Press.

[9] Guozhu Dong, Xiuzhen Zhang, Limsoon Wong, and Jinyan Li. CAEP: Classification by aggregating emerging patterns. In Setsuo Arikawa and Koichi Furukawa, editors, *Proc. of the 2nd International Conference on Discovery Science, (DS'99)*, volume 1721 of *Lecture Notes in Artificial Intelligence*, pages 30–42, Tokyo, Japan, 1999. Springer-Verlag.

[10] W. J. Frawley, G. Piatetsky-Shapiro, and C. Matheus. Knowledge Discovery In Databases: An Overview. In G. Piatetsky-Shapiro and W. J. Frawley, editors, *Knowledge Discovery In Databases*, pages 1-30, AAAI Press/MIT Press, Cambridge, MA., 1991.

[11] Jinyan Li. *Mining Emerging Patterns to Construct Accurate and Efficient Classifiers.* PhD thesis, Department of Computer Science & Software Engineering, The University of Melbourne, 2001.

[12] Jinyan Li, Guozhu Dong, and Kotagiri Ramamohanarao. Instance-based classification by emerging patterns. In Djamel A. Zighed, Jan Komorowski, and Jan Zytkow, editors, *Proc. of the 4th European Conference on Principles and Practice of Knowledge Discovery in Databases*, volume 1910 of *Lecture Notes in Artificial Intelligence*, Lyon, France, 2000. Springer-Verlag.

[13] Jinyan Li, Guozhu Dong, and Kotagiri Ramamohanarao. Making use of the most expressive jumping emerging patterns for classification. In Takao Terano, Huan Liu, and Arbee L. P. Chen, editors, *Proc. of the 4th Pacific-Asia Conference on Knowledge Discovery and Data Mining, Current Issues and New Applications (PAKDD'00)*, volume 1805 of *Lecture Notes in Artificial Intelligence*, pages 220–232, Kyoto, Japan, 2000. Springer-Verlag.

[14] Bing Liu, Wynne Hsu, and Yiming Ma. Integrating classification and association rule mining. In Rakesh Agrawal and Paul Stolorz, editors, *Proc. of the International Conference of Knowledge Discovery and Data Mining (KDD'98)*, pages 80–86, New York, 1998. AAAI, AAAI Press.

[15] Edward M. McCreight. A space-economical suffix tree construction algorithm. *Communication of the ACM*, 23(2):262–272, 1976.

[16] MIDI Manufacturers Association, MMA, PO Box 3173, La Habra, CA 90632-3173. *The Complete MIDI 1.0 Detailed Specification, document version 96.1*, 1996.

[17] Y. K. Muthusamy, E. Barnard, and R. A. Cole. Reviewing automatic language identification. *IEEE Signal Processing Magazine*, 11(4):33–41, 1994.

[18] J. Pei, J. Han, B. Mortazavi-Asl, and H. Zhu. Mining Access Patterns Efficiently from Web Logs. In *Proc. of the 4th Pacific-Asia Conference on Knowledge Discovery and Data Mining (PAKDD'00)*, pages 396–407, Kyoto, Japan, 2000.

[19] R. J. Povinelli. Identifying temporal patterns for characterization and prediction of financial time series events. In *Proc. of the International Workshop on Temporal, Spatial and Spatio-Temporal Data Mining (TSDM'00)*, pages 46–61, Lyon, France, 2000.

[20] John Ross Quinlan. *C4.5: Programs for Machine Learning.* Morgan Kaufmann Publishers, Inc., 1993.

[21] G. Stober, M. M. Nothen, P. Porzgen, M. Bruss, H. Bonisch, M. Knapp, H. Beckmann, and P. Propping. Systematic search for variation in the human norepinephrine transporter gene: identification of five naturally occurring missense mutations and study of association with major psychiatric disorders. In *Am J Med Genet*, 67(6):523–532, 1996.

[22] Michael Tang, Chi-Lap Yip, and Ben Kao. Selection of melody lines for music databases. In *Proc. of the 24th Annual International Computer Software and Applications Conference, 2000 (COMPSAC'00)*, pages 243–248, Taipei, Taiwan, 2000. IEEE.

[23] Esko Ukkonen. Constructing suffix trees on-line in linear time. *Information Processing*, 1:484–492, 1992.

[24] Rita S. Wolpert. Recognition of melody, harmonic accompaniment, and instrumentation: Musicians vs. nonmusicians. *Music Perception*, 8(1):95–106, 1990.

[25] Chi-Lap Yip and Ben Kao. A study on musical features for melody databases. In Trevor Bench-Capon, Giovanni Soda, and A. Min Tjoa, editors, *Proc. of the 10th International Conference and Workshop on Database and Expert Systems Applications (DEXA'99)*, volume 1677 of *Lecture Notes in Computer Science*, pages 724–733, Florence, Italy, 1999. Springer-Verlag.