

Efficiently Mining Closed Subsequences with Gap Constraints *

Chun Li[†], Jianyong Wang[‡]

Department of Computer Science and Technology

Tsinghua University, Beijing, 100084, China

[†]Socrates.lee@gmail.com, [‡]jianyong@tsinghua.edu.cn

Abstract

Mining frequent subsequence patterns from sequence databases is a typical data mining problem and various efficient sequential pattern mining algorithms have been proposed. In many problem domains (e.g, biology), the frequent subsequences confined by the predefined gap requirements are more meaningful than the general sequential patterns. In this paper we re-examine the closed sequential pattern mining problem by introducing the gap constraints. The most challenging parts in this task include the constrained pattern closure checking and unpromising search space pruning. Inspired by some state-of-the-art closed or constrained sequential pattern mining algorithms, we propose an efficient approach to finding the complete set of closed sequential patterns with gap constraints. The approach combines the newly devised constrained pattern closure checking scheme and pruning techniques with the pattern growth based subsequence enumeration framework. Our extensive performance study shows that our approach is very efficient in mining frequent closed subsequences with gap constraints.

Keywords. Constrained subsequence mining, frequent closed subsequence, gap-constraint.

1 Introduction

Since sequences reflect the temporal relationships among the real world objects (or events), sequential pattern mining has become an essential data mining task, which has shown broad applications, including frequent subsequence-based classification [6, 24], discover-

ing block correlations in storage systems and identifying copy-paste and related bugs in large-scale software code [13, 14], API specification mining and API usage mining from open source repositories [28, 15], sequence-based clustering [27, 1], web log analysis [5], pattern discovery in genome and protein sequences [22, 32, 33], and so on. Many efficient sequential pattern mining algorithms have been proposed for various problem formulations, such as the general sequential pattern mining [2, 17, 25, 9, 31, 19, 3], closed sequential pattern mining [29, 26], constraint-based sequential pattern mining [7, 20, 23, 12], frequent episode mining [16], cyclic association rule mining [18], partial periodic pattern mining [8], frequent partial order mining [4, 21] and long sequential pattern mining in noisy environment [30].

In recent years many studies focused on two typical problems of sequential pattern mining. The first problem is on mining sequential patterns with gap constraint [32, 12, 33]. Since limited gaps reflects the coherency requirements among the temporally correlated pattern elements (especially in some bio-data such as genome and protein sequences), it is necessary to devise some methods to mine sequential patterns with user specified gap constraints. Mining frequent subsequence patterns with gap-constraints helps discover sequential patterns of elements separated within a specified gap, thus can also reduce the size of result set. To perform this task the algorithm needs to record the occurrences of each sequence element in order to check if a sequential pattern meets the gap-constraint, which may consume a lot of memory space.

The second recent focus on sequential pattern mining is to mine the complete set of frequent closed subsequences [29, 26]. Recent studies have shown that mining closed patterns instead of all patterns would lead to a more compact yet complete result set and better efficiency. The Clospan algorithm [29] mines closed sequential patterns using a candidate maintenance-and-test paradigm, that is, it needs to maintain a candidate set of already mined patterns. Thus Clospan consumes much memory when the result set is huge. The BIDE

*This work was supported in part by National Basic Research Program of China under Grant No. 2006CB303103, Basic Research Foundation of Tsinghua National Laboratory for Information Science and Technology(TNLIST), Program for Selected Talents (i.e., "Gu Gan Ren Cai") in Tsinghua University, Program for New Century Excellent Talents in University under Grant No. NCET-07-0491, and the Scientific Research Foundation for the Returned Overseas Chinese Scholars, State Education Ministry of China.

algorithms [26] mines all closed patterns using a BI-Directional Closure Checking scheme without keeping the candidate pattern sets, therefore it needs no space to maintain the intermediate result set.

Can we find a way to mine closed sequential patterns with gap-constraints? In this paper we propose an efficient approach to finding the complete set of closed sequential patterns with gap constraints. The new algorithm, Gap-BIDE, combines the newly devised constrained pattern closure checking scheme and pruning techniques with the pattern growth based subsequence enumeration framework. As Gap-BIDE inherits the same design philosophy as BIDE algorithm, it shares the same merit, that is, it does not need to maintain a candidate pattern set, which saves space consumption. Our extensive performance study shows that Gap-BIDE is both runtime and space efficient in mining frequent closed subsequences with gap constraints.

The rest of this paper is organized as follows. In section 2 we present the problem definition of frequent closed gap-constrained sequential pattern mining and discuss the related work. Section 3 focuses on the Gap-BIDE algorithm, mainly introducing the gap-constrained BI-Directional Extension pattern closure checking scheme and the gap-constrained BackScan pruning method. In section 4 we present an extensive performance study. Finally, we conclude the study in section 5.

2 Preliminaries and Related Work

2.1 Preliminaries

To simplify our discussion, let us first introduce some preliminaries for gap-constrained closed sequential pattern mining.

Let a set of distinct items $I=\{i_1, i_2, \dots, i_n\}$ be the alphabet of sequences. A sequence S is an ordered list of events, denoted by $\langle e_1, e_2, \dots, e_m \rangle$, where e_j is an item, namely, $e_j \in I$ for $1 \leq j \leq m$. For brevity, a sequence of $\langle e_1, e_2, \dots, e_m \rangle$ is also written as $e_1 e_2 \dots e_m$. The number of events (i.e., instances of items) in a sequence is called the length of the sequence and a sequence of length l is also called an l -sequence. Take DNA Sequence for example, $I=\{A, T, C, G\}$, and ATTACAG is a 6-sequence.

A sequence $S_a=a_1 a_2 \dots a_n$ is said to be contained in another sequence $S_b=b_1 b_2 \dots b_m$, if $n \leq m$ and there exist integers $1 \leq j_1 < j_2 < \dots < j_n \leq m$ such that $a_1=b_{j_1}, a_2=b_{j_2}, \dots, a_n=b_{j_n}$. If sequence S_a is contained in sequence S_b , S_a is called a *subsequence* of S_b and S_b a *supersequence* of S_a , denoted by $S_a \sqsubseteq S_b$.

A wild-card (denoted by a single dot, ‘.’) is a special symbol that matches any item in the alphabet. A gap is a sequence of wild-cards. The size of a gap refers to the number of wild-cards in it. For example, the size

of ‘.....’ is 5. We use $g(N)$ to represent a gap of size N ; we use $g(M, N)$ to represent a gap whose size is within the range $[M, N]$. The range $[M, N]$ is called a *gap-constraint*.

A pattern P with length l and gap-constraint $[M, N]$ is an ordered list of l events (i.e., items) with a $g(M, N)$ gap between every two neighbor events, denoted by $P=\langle p_1, g(M, N), p_2, g(M, N), \dots, g(M, N), p_l \rangle$, or $P_{[M, N]}=p_1 p_2 \dots p_l$ for short, where p_j is an item, namely, $p_j \in I$ for $1 \leq j \leq l$. A pattern $P_{[M, N]}$ matches a sequence $S=e_1 e_2 \dots e_m$ w.r.t. gap constraint $[M, N]$, if there exist integers $1 \leq j_1 < j_2 < \dots < j_l \leq m$ such that $p_1=e_{j_1}, p_2=e_{j_2}, \dots, p_l=e_{j_l}$ and $M \leq j_k - j_{k-1} \leq N$, for $2 \leq k \leq l$. The subsequence $e_{j_1} e_{j_2} \dots e_{j_l}$ of S , is called an *appearance* of P in S . For example, gap-constrained pattern $P=\langle A, g(2, 3), T, g(2, 3), G \rangle$ matches sequence $S=\text{ACTTACAGTT}$, and ACTTACAG is an appearance of P in S , but P does not match sequence ACCCATATG.

A gap-constrained pattern $P_{a[M, N]}=a_1 a_2 \dots a_h$ is said to be contained in another gap-constrained pattern $P_{b[M, N]}=b_1 b_2 \dots b_l$, if $h \leq l$ and there exist integers $1 \leq j_1 < j_2 < \dots < j_h \leq l$ such that $a_1=b_{j_1}, a_2=b_{j_2}, \dots, a_h=b_{j_h}$, and P_a is said to be contiguously contained in P_b , if $i_k+1=i_{k+1}$, where $1 \leq k \leq h-1$. If pattern P_a is contained in pattern P_b , P_a is called a *subpattern* of P_b and P_b is called a *superpattern* of P_a . If pattern P_a is contiguously contained in pattern P_b , P_a is called a *contiguous subpattern* of P_b and P_b is called a *contiguous superpattern* of P_a . The *prefix pattern* P' of a gap-constrained pattern $P_{a[M, N]}=a_1 a_2 \dots a_h$ is a contiguous subpattern of P_a containing the first $h-1$ events of P_a , namely, $P'_{[M, N]}=a_1 a_2 \dots a_{h-1}$.

An input sequence database SDB is a set of tuples (sid, S), where sid is a sequence identifier, and S an input sequence. Table 1 shows the input sequence database SDB in our running example. The number of tuples in SDB is called the base size of SDB, denoted by $|SDB|$. Given a gap constraint $[M, N]$, a tuple (sid, S) is said to *contain* a gap-constrained pattern P , if P matches S w.r.t. gap constraint $[M, N]$. The *absolute sequence support* of a gap-constrained pattern P in a sequence database SDB is the number of tuples in SDB which contain P , denoted by $seq_sup^{SDB}(P)$, and the *relative sequence support* is the percentage of tuples in SDB that contain P (i.e., $seq_sup^{SDB}(P)/|SDB|$). The *universal support* of a gap-constrained pattern P in a sequence database SDB is the total number of appearances of P in SDB, denoted by $uni_sup^{SDB}(P)$. Obviously, we have $uni_sup^{SDB}(P) \geq seq_sup^{SDB}(P)$.

Given an absolute support threshold min_sup , a gap constraint $[M, N]$, and an input sequence database SDB, a subsequence P is called a *frequent gap-constrained subsequence* (or a gap-constrained sequen-

sid	Sequence
0	$\langle 0, 1, 2, 3, 1 \rangle$
1	$\langle 0, 2, 2, 1, 0, 3, 2 \rangle$
2	$\langle 0, 4, 1, 2, 3, 2 \rangle$

Table 1: An example sequence database SDB .

tial pattern) if $seq_sup^{SDB}(P) \geq min_sup$. The task of *gap-constrained sequential pattern mining* is to mine the complete set of frequent gap-constrained subsequences in database SDB .

EXAMPLE 1. *In the running example shown in Table 1, let the absolute sequence support threshold be 2 and the gap constraint be $[2,3]$, the set of gap-constrained sequential patterns contains 10 patterns in total, namely, $\langle 0 \rangle:3$, $\langle 1 \rangle:3$, $\langle 2 \rangle:3$, $\langle 3 \rangle:3$, $\langle 0,1 \rangle:2$, $\langle 0,2 \rangle:3$, $\langle 1,2 \rangle:2$, $\langle 1,3 \rangle:3$, $\langle 0,1,2 \rangle:2$, $\langle 0,1,3 \rangle:2$ (Note that the number after the colon is the absolute sequence support of the corresponding pattern). ■*

2.2 Related Work

The sequential pattern mining problem was first proposed by Agrawal and Srikant in [2], and the same authors also designed an efficient GSP algorithm [25] based on the well-known Apriori property. Since then, many sequential pattern mining algorithms have also been proposed for performance improvements [9, 31, 19, 3].

The Clospan [29] algorithm was proposed to mine closed sequential patterns. It mines a candidate set of closed patterns based on Prefixspan algorithm [19], and keeps the candidate patterns sets for post-pruning. The BIDE [26] algorithm which does not need to keep a candidate set of closed patterns also adopts the framework of Prefixspan to generate new patterns. Instead of post-pruning, BIDE checks and prunes a candidate pattern as soon as it is found.

There are also several algorithms focusing on the gap-constrained sequential pattern mining. The TEIRESIAS [22] algorithm mines genome sequence patterns with wildcards in fixed positions, such as "A..TG.C". It generates new patterns by convoluting two shorter already mined patterns. The MPPm [32] algorithm focuses on mining frequent gap-constrained sequential patterns in a single genome sequence. The work in [12] studies the problem of mining minimal distinguishing subsequence patterns with gap constraints. The MCPas [33] algorithm mines frequent gap-constrained sequential patterns in multiple sequences, and further improves the performance. These algorithms need to keep the position of every item of all appearances, so the memory consumption is big when the dataset is large and very dense.

3 Gap-BIDE: Mining Closed Gap-Constrained Sequential Patterns

In this section, we first introduce the formal problem formulation of closed gap-constrained sequential pattern mining, then present the Gap-BIDE algorithm in detail.

3.1 Closed Gap-Constrained Sequential Pattern Mining

Though the gap constraint remarkably reduces the size of the result set, mining all the frequent patterns of some dense datasets such as DNA datasets will get very huge result sets.

In general sequential pattern mining, a *closed pattern* is defined as a pattern which has no superpattern of the same sequence support. Mining closed patterns not only reduces the size of the result set, but also causes no information loss. Directly use the concept of closed pattern mining in the gap-constrained sequential pattern mining setting will induce some problems. According to the traditional definition of a closed sequential pattern, all its subpatterns must be sequential patterns too. However, as the downward closure property no longer holds in the gap-constrained sequential pattern mining setting, a subpattern of a gap-constrained sequential pattern may not form a sequential pattern. For example, if database SDB has only one input sequence ACCTTAGT, and the support threshold is 1, the pattern $P_a[2,3]=ATG$ is a closed pattern according to the traditional definition since there is no superpattern of P_a which has the same support. However, one of its subpattern, $P_b[2,3]=AG$, is not a gap-constrained sequential pattern, which violates the gap constraint. Although the downward closure property does not hold, we can easily derive the following property of a gap-constrained sequential pattern, and get that AT and TG are gap-constrained sequential patterns.

PROPERTY 3.1. *All contiguous subpatterns of a gap-constrained sequential pattern must be gap-constrained sequential patterns.*

Based on the above analysis, we adjust the definition of a closed gap-constrained sequential pattern in the following way.

DEFINITION 1. *A closed gap-constrained sequential pattern is such a pattern P_a , that there is no contiguous superpattern P_b of P_a , where $seq_sup^{SDB}(P_a) = seq_sup^{SDB}(P_b)$.*

Given an input sequence database SDB , an absolute support threshold min_sup , and a gap constraint $[M, N]$, our task is to mine *the complete set of closed*

gap-constrained sequential patterns from SDB with respect to the support threshold min_sup and gap the constraint $[M, N]$.

3.2 The Gap-BIDE Algorithms

In this section, we introduce the Gap-BIDE algorithm which shares the same design philosophy as BIDE algorithm [26]. First, we establish a framework to enumerate all the frequent gap-constrained patterns. Then we introduce a gap-constrained closed pattern checking scheme under this enumeration framework. At last, we derive a gap-constrained pruning technique to improve the efficiency of the algorithm.

3.2.1 Frequent Gap-Constrained Pattern Enumeration Framework

Many previous algorithms (e.g., the teiresias algorithm [22]) generate new gap-constrained patterns by convoluting shorter patterns, which means they compare the suffix of one pattern with the prefix of another pattern to see if the two could be joined together. This method needs to store all the position information of each appearance of any pattern, the memory consumption could be huge when the dataset is large and dense.

We basically adopt the framework of PrefixSpan [19] algorithm to enumerate all the frequent gap-constrained patterns. We recursively add new items to the shorter patterns, thus we do not need to store all the position information of all already mined patterns.

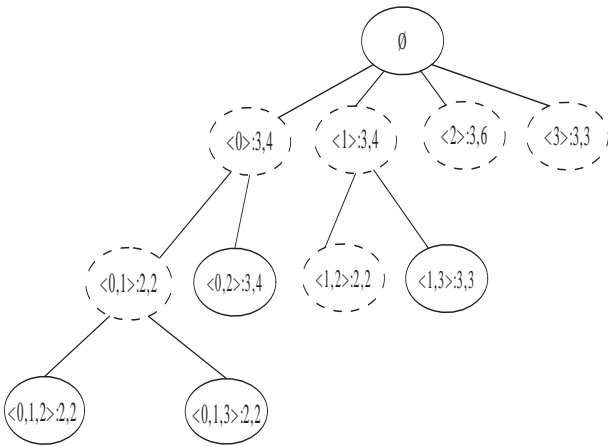


Figure 1: The lexicographic gap-constrained frequent closed sequence tree in our running example.

Given a lexicographic ordering of the items in alphabet I , the search space of the sequence mining forms a unique sequence tree. The root node is \emptyset ,

each other node of the tree represents a gap-constrained sequential pattern. By extending a node of a length- l pattern P , we can find all the length- $(l+1)$ patterns with prefix P according to the chosen lexicographic ordering. We search the sequence tree in depth-first order. Each non-root node is the prefix pattern of its child nodes, and the algorithm enumerates all the patterns recursively. The Figure 1 demonstrates the search tree of the running example shown in Table 1, where the absolute sequence support threshold is 2 and the gap constraint is $[2,3]$. The closed patterns and non-closed patterns are shown in the nodes with solid borders and dotted borders, respectively. In the figure each gap-constrained subsequence pattern is followed by a colon, its absolute sequence support, and its universal support.

ALGORITHM 1: **PatternsEnumeration**(SDB, min_sup, M, N)

INPUT: (1) SDB : An input sequence database, (2) min_sup : the minimum support threshold, (3) M and N : the parameters of a gap constraint.

OUTPUT: the set of gap-constrained sequential patterns.

01. find the set of all length-1 frequent items, $L1$;
 02. for each item i in $L1$
 03. call **PGrowth**(i);
 04. return.
-

SUBROUTINE 1: **PGrowth**(P)

INPUT: (1) P : A prefix sequence pattern.

OUTPUT: the set of gap-constrained sequential patterns with prefix P .

05. output pattern P ;
 06. search each forward space of all appearances of P , and find the set of all length-1 locally frequent items, L ;
 07. for each item i in L
 08. build new pattern $Pnew = P+i$;
 09. call **PGrowth**($Pnew$);
 10. return.
-

For each gap-constrained pattern in a given sequence database SDB , each appearance of the pattern is recorded by a triple $(sid, beginPos, endPos)$, where sid is the ID of the sequence where the appearance occurs, $beginPos$ and $endPos$ are the positions of the first item and last item of the pattern in the sequence.

DEFINITION 2. (*Backward Space*) Given an appearance of pattern $P[M, N]$ with triple $(sid, beginPos, endPos)$, the backward space of the appearance is part of the sequence sid which is of the range $[beginPos - N, beginPos - M] \cap [0, beginPos)$.

DEFINITION 3. (*Forward Space*) Given an appearance of pattern $P[M, N]$ with triple $(sid, beginPos, endPos)$, the forward space of appearance is part of the sequence sid which is of the range $[endPos + M, endPos + N] \cap [endPos, l)$, where l is the length of sequence sid .

EXAMPLE 2. In the running example shown in Table 1, if the sequence support threshold is 2 and gap constraint is $[2, 3]$, the backward spaces of the pattern $\langle 1 \rangle$ are $\langle 1, 2 \rangle$, $\langle 0, 2 \rangle$, $\langle 0 \rangle$; the forward spaces of the pattern $\langle 1 \rangle$ are $\langle 3, 1 \rangle$, $\langle 3, 2 \rangle$, $\langle 3, 2 \rangle$. ■

We can get the sequence support of every item by scanning the forward spaces of the appearances of a prefix pattern. The set of items whose support is no smaller than the minimal support threshold min_sup is the locally frequent items of a prefix pattern.

We can get the enumeration algorithm in ALGORITHM 1. As the algorithm is straightforward, we will not elaborate it here.

3.2.2 The Gap-Constrained Bi-Directional Closure Checking Scheme

The enumeration algorithm in previous section only mines all the frequent gap-constrained sequential patterns instead of only closed ones. When getting upon a frequent prefix pattern, we need to check if it is a closed one.

Most of the existing algorithms on mining closed itemsets or closed general sequential patterns need to maintain a candidate set of closed patterns for checking if newly discovered patterns are closed. But keeping candidate sets might lead to huge memory cost and low efficiency when the result set is very large. The BIDE algorithm uses a BI-Directional Closure Checking scheme to do closure checking without keeping a candidate pattern sets, thus achieves better efficiency in terms of both space and runtime. Inspired by the BIDE algorithm, we devise a gap-constrained sequential pattern closure checking scheme for the Gap-BIDE algorithm, and thus avoids keeping a large candidate set of closed frequent gap-constrained patterns in order for checking if a newly mined pattern is closed.

Assume $P_{a[M, N]} = p_1 p_2 \dots p_l$ is a gap-constrained sequential pattern in database SDB. Given another gap-constrained sequential pattern $P_{b[M, N]} = p_0 p_1 p_2 \dots p_l$, if $seq_sup^{SDB}(P_a) = seq_sup^{SDB}(P_b)$ holds, we say p_0 is a *backward extension item* of P_a . Let $P_{c[M, N]} = p_1 p_2 \dots p_l p_0$ be another gap-constrained sequential pattern, if $seq_sup^{SDB}(P_a) = seq_sup^{SDB}(P_c)$, we say p_0 is a *forward extension item* of P_a .

According to the above two definitions and the definition of a closed gap-constrained pattern, it is evident to get the following theorem.

THEOREM 3.1. A gap-constrained pattern P is closed, if and only if P has no backward extension item nor forward extension item.

The following two lemmas provide practical methods to find backward and forward extension items.

LEMMA 3.1. Search each item in each backward space of the appearances of a given gap-constrained pattern P in SDB, if there exists an item p_0 , whose local support within the backward spaces equals $seq_sup^{SDB}(P)$, p_0 must be a backward extension item of P .

Proof. Let $P_{[M, N]} = p_1 p_2 \dots p_l$. As item p_0 's local support within the backward spaces is equal to $seq_sup^{SDB}(P)$, we can get a new gap-constrained pattern $P'_{[M, N]} = p_0 p_1 p_2 \dots p_l$ by adding p_0 before P , and $seq_sup^{SDB}(P') = seq_sup^{SDB}(P)$. Therefore, p_0 is a backward extension item of P . ■

LEMMA 3.2. Search each item in each forward space of the appearances of a given gap-constrained pattern P in SDB, if there exists an item p_0 , whose local support within the forward spaces equals $seq_sup^{SDB}(P)$, p_0 is a forward extension item of P .

Proof. Assume that $P_{[M, N]} = p_1 p_2 \dots p_l$. As item p_0 's local support within the forward spaces is equal to $seq_sup^{SDB}(P)$, we can get a new gap-constrained pattern $P'_{[M, N]} = p_1 p_2 \dots p_l p_0$ by adding p_0 after P , and $seq_sup^{SDB}(P') = seq_sup^{SDB}(P)$. Therefore, p_0 is a forward extension item of P . ■

EXAMPLE 3. In the running example shown in Table 1, by scanning the backward spaces of pattern $\langle 2 \rangle$ we find item 0 with a support of 3, thus pattern $\langle 2 \rangle$ is not a closed one; and in the forward spaces of pattern $\langle 0 \rangle$ we find item 2 with a support of 3, we know that pattern $\langle 0 \rangle$ is not a closed one too. ■

3.2.3 The Gap-Constrained BackScan Pruning Method

Performing the pattern closure checking leads to a more compact result set, but it does not improve the algorithm efficiency. Inspired by the BIDE algorithm, we propose a gap-constrained BackScan optimization technique in order to prune the unpromising parts of the search space for closed gap-constrained sequential pattern mining.

The idea of the *gap-constrained BackScan pruning* method is based on the observation that a single sequence may contain multiple appearances of the same gap-constrained pattern, and can be stated as the following theorem.

THEOREM 3.2. *Scan each backward space of the appearances of a given gap-constrained pattern P in SDB , if there exists an item p_0 , whose local universal support within the backward spaces, $uni_sup^{SDB}(p_0)$, is equal to $uni_sup^{SDB}(P)$, we can safely prune pattern P .*

Proof. *Let $P_{[M,N]}=p_1p_2\dots p_l$, then we can get a new sequential pattern $P'_{[M,N]}=p_0 p_1 p_2 \dots p_l$. Assume we extend $P_{[M,N]}$ by a gap-constrained subsequence s to get a longer pattern Q , denoted by $Q=P \diamond s$, we can always construct another pattern $Q'=P' \diamond s$. Because $uni_sup^{SDB}(p_0) = uni_sup^{SDB}(P)$, we have $uni_sup^{SDB}(Q') = uni_sup^{SDB}(Q)$. Thus, we cannot use P as a prefix pattern to generate any closed gap-constrained sequential patterns. ■*

We can prune the unpromising parts of the search space easily using the above Theorem. The following is an illustrating example.

EXAMPLE 4. *In the running example shown in Table 1, in the forward spaces of pattern $\langle 3 \rangle$ we find item 1 with a universal support of 3, which is equal to the universal support of pattern $\langle 3 \rangle$, so pattern $\langle 3 \rangle$ can be safely pruned. ■*

ALGORITHM 2: Gap-BIDE(SDB, min_sup, M, N)

INPUT: (1) SDB : An input sequence database, (2) min_sup : the minimum support threshold, (3) M and N : the parameters of a gap constraint.

OUTPUT: the set of gap-constrained closed sequential patterns.

11. find the set of length-1 frequent sequential patterns, $L1$;
 12. for each item i in $L1$
 13. call **PatternGrowth**(i);
 14. return.
-

SUBROUTINE 2: PatternGrowth(P)

INPUT: (1) P : A prefix sequence pattern.

OUTPUT: the set of gap-constrained closed sequential patterns with prefix P .

15. *backward_check*($P, needPruning, hasBackwardExtension$);
 16. if (*needPruning*)
 17. return;
 18. *forward_check*($P, hasForwardExtension$);
 19. if !(*hasBackwardExtension* || *hasForwardExtension*)
 20. output pattern P
 21. search each forward space of all appearances of P , and find the set of all locally frequent items, L ;
 22. for each item i in L
 23. build new pattern $Pnew = P+i$;
 24. call **PatternGrowth**($Pnew$);
 25. return.
-

3.2.4 The Algorithm

Algorithm 2 describes the Gap-BIDE algorithm which incorporates the pattern closure checking scheme and pruning technique described in Sections 3.2.2 and 3.2.3 into the gap-constrained sequence enumeration framework introduced in Section 3.2.1.

The algorithm first scans the database, finds the set of all frequent items (i.e., length-1 sequential patterns) and calls the subroutine **PatternGrowth**(P) to mine the gap-constrained closed sequential patterns with pattern P as the prefix (Lines 11-14). The subroutine **PatternGrowth**(P) first scans the backward spaces of prefix pattern P (Line 15), uses the gap-constrained Backscan pruning method to prune search space (Lines 16-17), scans the forward spaces of prefix P (Line 18), and uses the gap-constrained pattern closure checking scheme to check if pattern P is closed (Line 19), if so, output P as a gap-constrained closed sequential pattern (Line 20). Finally, it scans each forward space of all appearances of pattern P and finds the set of all locally frequent items, L (Line 21), uses each item in L to extend P , and mines the gap-constrained closed sequential patterns for the new prefix $Pnew$ by calling subroutine **PatternGrowth**($Pnew$) itself (Lines 22-24).

4 Empirical Results

4.1 Test Environment and Datasets

All of our experiments were performed on a Lenovo ThinkPad T60 with Intel T2400 CPU, 1GB memory and Windows XP professional installed. To evaluate the performance of Gap-BIDE, we used three real datasets. The first dataset, Gazelle, is very sparse, but it contains some very long frequent closed sequences with low support thresholds. This dataset was originally provided by Blue Martini and has been popularly used in evaluating several sequential pattern mining algorithms [29, 26]. It contains a total of 29,369 customers' Web click-stream data. The second dataset, AX829174, is very dense, and can be downloaded from the National Center for Biotechnology Information website [11]. It is a Homo Sapiens (human) DNA sequence, and has been used in previous studies [32, 33]. We randomly sampled 1000 sequences to form the dataset for our practical experiments. The third dataset, ProgramTrace, is a dense program trace dataset. It contains a total of 10 program traces and 105 unique items with an average sequence length of 488.

To our best knowledge, there is no existing algorithm which mines closed sequential patterns with gap-constraints, we thus evaluate Gap-BIDE in the following way. We first show that compared with sequential pattern mining with gap constraints, closed

gap-constrained sequential pattern mining reduces the number of patterns remarkably. Second, we present the comparison results among Gap-BIDE, BIDE, and CloSpan to show that the gap constraint is very useful in improving the algorithm efficiency. Finally we introduce the evaluation results for Gap-BIDE, including efficiency test, evaluation of the effectiveness of the pruning method, and scalability test.

4.2 Effectiveness of Closed Pattern Mining

Previous studies have shown that mining closed sequential patterns can lead to more concise result set than mining all sequential patterns [29, 26]. Is it still true for gap-constrained closed sequential pattern mining? To validate it, we conducted a series of experiments with various support thresholds and gap constraints, found that the above statement still holds in the gap-constrained sequential pattern mining setting. For example, by setting the absolute support threshold at 3 and the gap constraint at [2,4], there are totally 1,539,313 all gap-constrained sequential patterns, while there are only 224,162 gap-constrained closed sequential patterns for dataset AX829174. Figure 2 illustrates the difference between the numbers of patterns returned by Gap-BIDE and that of all gap-constrained sequential patterns for dataset AX829174 with the gap constraint of [2,4].

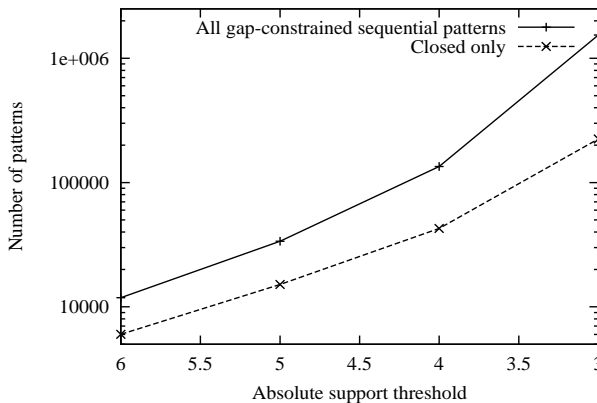


Figure 2: Closed patterns vs. All sequential patterns (# patterns, AX829174).

4.3 Effectiveness of Gap-Constrained Pattern Mining

We also conducted experiments to compare the Gap-BIDE algorithm with two state-of-the-art closed sequential pattern mining algorithms, BIDE and CloSpan, for dataset ProgramTrace in order to show that pushing the gap-constraints into closed sequential pattern min-

ing does make much sense in improving the algorithm efficiency. The experimental results show that mining gap-constrained patterns improves the algorithm efficiency significantly. For example, with absolute support threshold setting at 7, CloSpan crashed after 144,000 seconds' running, BIDE takes 288.56 seconds to finish, but Gap-BIDE only uses 0.422 seconds with the gap constraint of [4,8]. Figure 3 compares the runtime efficiency of CloSpan, BIDE, and Gap-BIDE with gap constraints of [4,8] and [4,6].

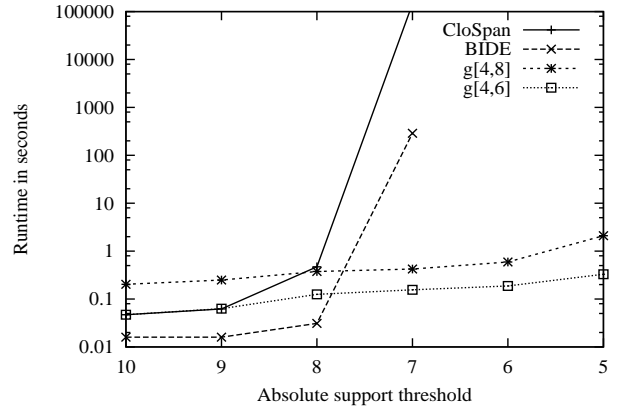


Figure 3: Gap-BIDE vs. BIDE and CloSpan (Runtime, ProgramTrace).

4.4 Evaluation of the Gap-BIDE Algorithm

Efficiency Test. We conducted a series of experiments to test the efficiency of Gap-BIDE using both AX829174 and Gazelle datasets. Here, we are interested in how Gap-BIDE algorithm acts when the minimum support threshold and gap constraint vary.

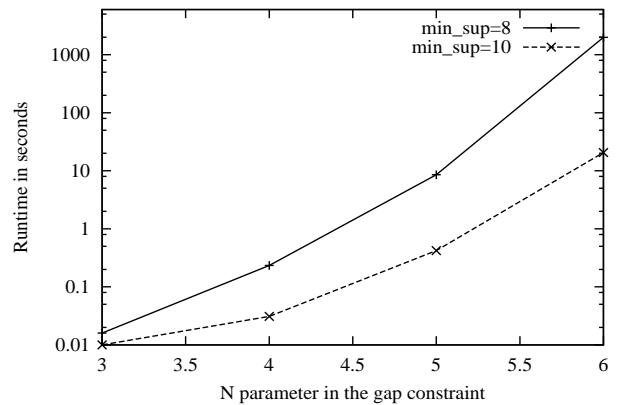


Figure 4: Varying the value of parameter N in the gap constraint (Runtime, AX829174, M=2).

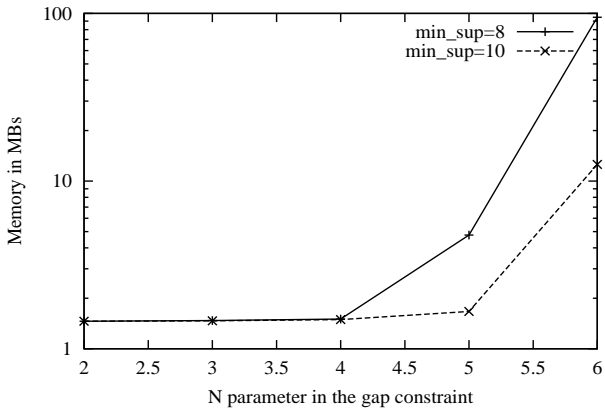


Figure 5: Varying the value of parameter N in the gap constraint (Memory, AX829174, M=2).

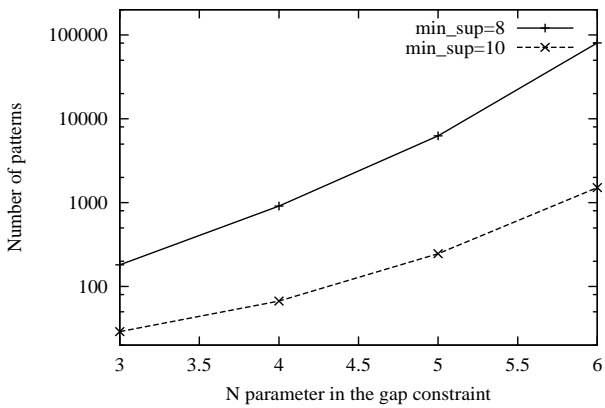


Figure 6: Varying the value of parameter N in the gap constraint (# patterns, AX829174, M=2).

We first fixed the minimum support at a certain value and changed the gap constraint to see how sensitive Gap-BIDE is to the gap constraint. Figures 4 and 5 show part of the results for dataset AX829174. In the experiments, we fixed the absolute minimum support at 8 and 10, respectively, and varied the gap constraint from [2,3] to [2,6]. We see that both the runtime and space usage increase exponentially with the linear increasing gap constraint. For example, at $min_sup=8$, the runtime of Gap-BIDE increases from 8.563 seconds to 1988 seconds when the gap constraint varies from [2,5] to [2,6]. This phenomena can be explained from two aspects. On the one hand, the search space increases dramatically when the gap constraint increases, on the other hand, the number of patterns increases as we release the gap constraint. Figure 6 shows the distribution of the number of mined patterns against the gap constraint when min_sup is set at 8 and 10 respectively. We see that the number of the discovered patterns also

increases exponentially with the linear increasing gap constraint.

From Figures 4, 5 and 6, we also see that for dense datasets like AX829174 the runtime, memory usage and number of mined patterns all increase significantly when we lower min_sup from 10 to 8. We then tested the efficiency of Gap-BIDE with varying minimum supports for sparse dataset Gazelle. In the experiments we fixed the gap constraint at [4,8] and [2,10] respectively, and varied the absolute minimum support from 45 to 15. Figures 7 and 8 show the runtime and memory consumption against the minimum support for sparse dataset Gazelle. We see that both the runtime and memory usage increase marginally with the decrease of minimum support. This phenomena can be partially explained by Figure 9, which shows that the number of mined patterns increases linearly with the increasing of minimum support.

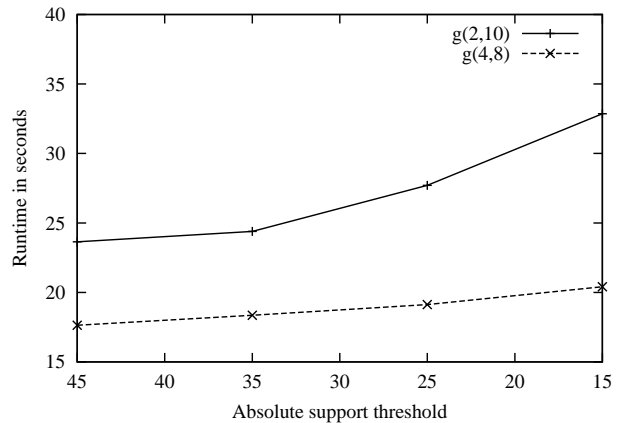


Figure 7: Varying the minimum support threshold with different gap constraints (Runtime, Gazelle).

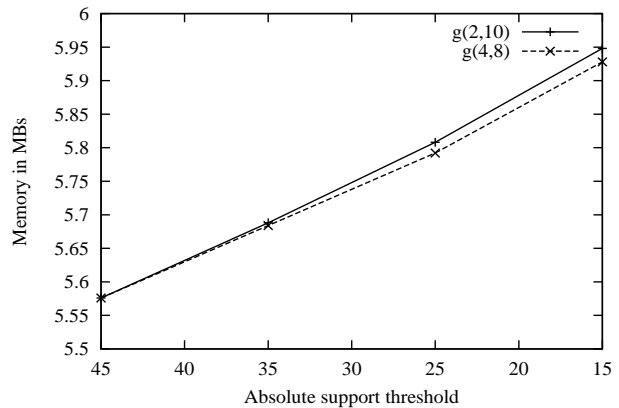


Figure 8: Varying the minimum support threshold with different gap constraints (Memory, Gazelle).

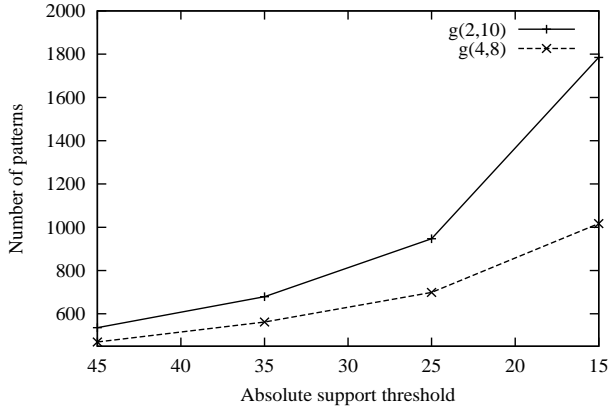


Figure 9: Varying the minimum support threshold with different gap constraints (# patterns, Gazelle).

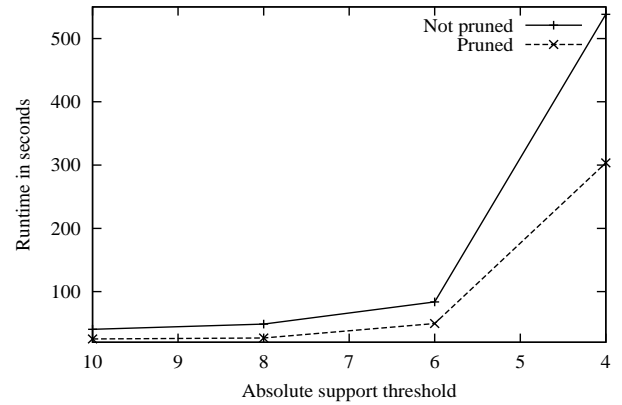


Figure 10: Pruning test on Gazelle (Runtime, $g(2,10)$).

Effectiveness of gap-constrained BackScan Pruning.

Fig. 10 shows the effectiveness of the gap-constrained BackScan pruning method adopted in Gap-BIDE algorithm for dataset Gazelle with the gap constraint of $[2,10]$. We see that the pruning method is relatively effective in pruning the unpromising parts of the search space, especially when support threshold is low. For example, with absolute support threshold setting at 4, the gap-constrained BackScan pruning method saves more than 200 seconds of runtime for dataset Gazelle with the gap constraint of $[2,10]$.

Scalability Test. We also evaluated the scalability of Gap-BIDE in terms of base size. In the experiments we used the dense dataset AX829174 and replicated it from 1 to 20 times. The gap constraint was set at $[2,3]$, and the minimum relative support threshold was set at 0.2, 0.4, and 0.8, respectively. Fig. 11 shows the scalability test of Gap-BIDE in terms of runtime efficiency, while Fig. 12 shows the scalability test of Gap-BIDE in terms of memory consumption. We see that both the runtime and space usage of Gap-BIDE increase linearly with the increasing number of input sequences, which implies that Gap-BIDE has good scalability in terms of base size.

5 Conclusions

This paper studies the problem of mining frequent closed sequential patterns with gap-constraints. We defined the concept of a ‘closed pattern’ in the gap-constrained sequential pattern mining setting, and devised the Gap-BIDE algorithm, which can efficiently mine gap-constrained closed sequential patterns. The Gap-BIDE algorithm combines the adjusted pattern growth based enumeration framework, the gap-constrained BI-Directional Closure Checking scheme,

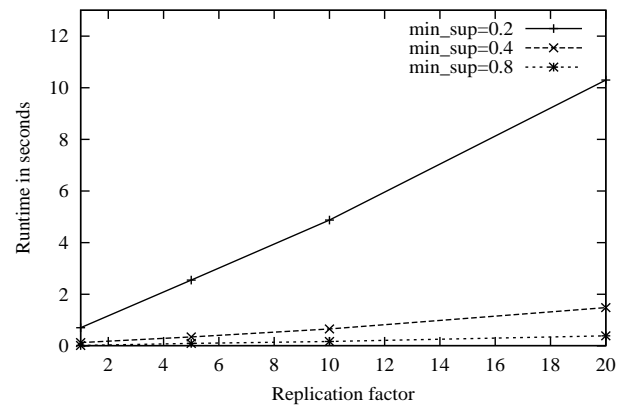


Figure 11: Scalability test of Gap-BIDE (Runtime, AX829174, $g(2,3)$).

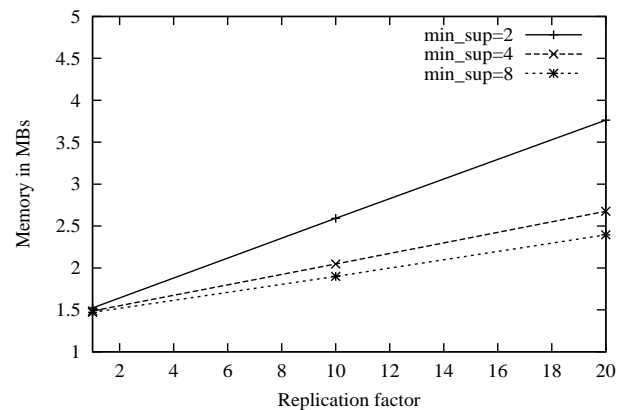


Figure 12: Scalability test of Gap-BIDE (Memory, AX829174, $g(2,3)$).

and the gap-constrained BackScan pruning method to meet the needs of our new problem formulation. Our experimental results show that the new algorithm is very

efficient in terms of both space and runtime, and scales well with respect to the database size. In addition, the mining of closed gap-constrained patterns reduces the number of discovered patterns and improves the algorithm efficiency significantly.

References

- [1] C.C. Aggarwal, N. Ta, J. Wang, J. Feng, M.J. Zaki. XProj: A Framework for Projected Structural Clustering of XML Documents. SIGKDD'07.
- [2] R. Agrawal, R. Srikant. *Mining Sequential Patterns*. ICDE'95.
- [3] J. Ayres, J. Gehrke, T. Yiu, J. Flannick. *Sequential Pattern Mining using a Bitmap Representation*. SIGKDD'02.
- [4] G. Casas-Garriga. *Summarizing Sequential Data with Closed Partial Orders*. SDM'05.
- [5] C. Chen, T. Cook. *Mining Contiguous Sequential Patterns from Web Logs*. WWW'07(poster).
- [6] M. Deshpande, G. Karypis. *Evaluation of Techniques for Classifying Biological Sequences*. PAKDD'02.
- [7] M. Garofalakis, R. Rastogi, K. Shim. *SPIRIT: Sequential Pattern Mining with Regular Expression Constraints*. VLDB'99.
- [8] J. Han, G. Dong, and Y. Yin. *Efficient mining of partial periodic patterns in time series database*. ICDE'99.
- [9] J. Han, J. Pei, B. Mortazavi-Asl, Q. Chen, U. Dayal, and M.C. Hsu. *FreeSpan: Frequent pattern-projected sequential pattern mining*. SIGKDD'00.
- [10] I. Jonassen, J.F. Collins, and D.G. Higgins. *Finding flexible patterns in unaligned protein sequences*. Protein Science, 4(8), 1995.
- [11] <http://www.ncbi.nlm.nih.gov>.
- [12] X. Ji, J. Bailey, G. Dong. *Mining Minimal Distinguishing Subsequence Patterns with Gap Constraints*. ICDM'05.
- [13] Z. Li, Z. Chen, S. Srinivasan, Y. Zhou. *C-Miner: Mining Block Correlations in Storage Systems*. USENIX FAST'04.
- [14] Z. Li, S. Lu, S. Myagmar, Y. Zhou. *CP-Miner: Finding Copy-Paste and Related Bugs in Large-Scale Software Code*. IEEE Transactions on Software Engineering, 32(3):176-192, 2006.
- [15] D. Lo, S.C. Khoo. *SMArTIC: Towards Building an Accurate, Robust and Scalable Specification Miner*. SIGSOFT FSE'06.
- [16] H. Mannila, H. Toivonen, A.I. Verkamo. *Discovering Frequent Episodes in Sequences*. SIGKDD'95.
- [17] F. Masegla, F. Cathala, P. Poncelet. *The psp Approach for Mining Sequential Patterns*. PKDD'98.
- [18] B. Ozden, S. Ramaswamy, A. Silberschatz. *Cyclic Association Rules*. ICDE'98.
- [19] J. Pei, J. Han, B. Mortazavi-Asl, Q. Chen, U. Dayal, M.C. Hsu. *PrefixSpan: Mining Sequential Patterns Efficiently by Prefix-Projected Pattern Growth*. ICDE'01.
- [20] J. Pei, J. Han, W. Wang. *Constraint-based Sequential Pattern Mining in Large Databases*. CIKM'02.
- [21] J. Pei, J. Liu, H. Wang, K. Wang, P.S. Yu, J. Wang. *Efficiently Mining Frequent Closed Partial Orders*. ICDM'05.
- [22] I. Rigoutsos, A. Floratos. *Combinatorial pattern discovery in biological sequences: the teiresias algorithm*. Bioinformatics, 14(1), 1998.
- [23] M. Seno, G. Karypis. *SLPMiner: An algorithm for Finding Frequent Sequential Patterns using Length-Decreasing Support Constraint*. ICDM'02.
- [24] R. She, F. Chen, K. Wang, M. Ester, et al. *Frequent-Subsequence-based Prediction of Outer Membrane Proteins*. SIGKDD'03.
- [25] R. Srikant, R. Agrawal. *Mining Sequential Patterns: Generalizations and Performance Improvements*. EDBT'96.
- [26] J. Wang, J. Han, C. Li. *Frequent Closed Sequence Mining without Candidate Maintenance*. IEEE Transactions on Knowledge and Data Engineering, 19(8):1042-1056.
- [27] J. Wang, Y. Zhang, L. Zhou, G. Karypis, C.C. Aggarwal. *Discriminating Subsequence Discovery for Sequence Clustering*. SDM'07.
- [28] T. Xie, J. Pei. *Data Mining for Software Engineering*. SIGKDD'06(tutorial).
- [29] X. Yan, J. Han, R. Afshar. *CloSpan: Mining Closed Sequential Patterns in Large Databases*. SDM'03.
- [30] J. Yang, P.S. Yu, W. Wang, J. Han. *Mining Long Sequential Patterns in a Noisy Environment*. SIGMOD'02.
- [31] M. Zaki. *SPADE: An Efficient Algorithm for Mining Frequent Sequences*. Machine Learning, 42:31-60, Kluwer Academic Publishers, 2001.
- [32] M. Zhang, B. Kao, D. Cheung, K. Yip. *Mining Frequent periodic patterns with gap requirement from sequences*. SIGMOD'05.
- [33] X. Zhu, X. Wu. *Mining Complex Patterns across Sequences with Gap Requirements*. IJCAI'07.