# GASP: Graph-based Approximate Sequential Pattern Mining

by

Wenqin Dong

Dr. Joyce C. Ho
Advisor

Department of Computer Science

Dr. Joyce C. Ho
Advisor

Dr. Davide Fossati
Committee Member

Dr. Roberto Franzosi
Committee Member

2020

# GASP: Graph-based Approximate Sequential Pattern Mining

by

Wenqin Dong

Dr. Joyce C. Ho
Advisor

An abstract of
a thesis submitted to the Faculty of Emory College of Arts and Sciences
of Emory University in partial fulfillment
of the requirements of the degree of
Bachelor of Science with Honors

Department of Computer Science

2020

## Abstract

GASP: Graph-based Approximate Sequential Pattern Mining

By Wenqin Dong

The rapid growth of data capturing sequential ordering information has led sequential pattern mining to become an essential data mining task. In sequential pattern mining, the goal is to discover frequent and useful patterns from sequences in the database. However, conventional algorithms (or exact sequential pattern mining algorithms) that discover all frequent sequential patterns generate a large number of patterns and incur a high computational and memory footprint. Thus, approximate sequential pattern mining techniques have been introduced. Yet, existing approximate methods fail to reflect the true frequent sequential patterns or only target singe-item event sequences. Multi-item event sequences are prominent in real-world applications. As an example, sequential pattern mining of electronic health records where a patient can have multiple interventions or diagnoses at the same visit. To alleviate these issues, we propose GASP, a graph-based approximate sequential pattern mining, that discovers frequent patterns not only on single-item event sequences but also multi-item event sequences. Our approach compresses the sequential information into a concise graph structure which results in a significantly smaller memory footprint. We assess the performance of GASP with both singe-item and multi-item event sequence datasets on computation time, memory usage, and recoverability of frequent patterns. The empirical results suggest that GASP outperforms existing approximate models by achieving better recoverability and outperforms existing exact models on computation time and memory usage.

# GASP: Graph-based Approximate Sequential Pattern Mining

By

Wenqin Dong

Dr. Joyce C. Ho
Advisor

A thesis submitted to the Faculty of Emory College of Arts and Sciences
of Emory University in partial fulfillment
of the requirements of the degree of
Bachelor of Science with Honors

Department of Computer Science

2020

Acknowledgements

First, I would like to thank my advisor, Dr. Joyce C. Ho for her guidance and support throughout the project. Dr. Ho not only teaches me how to proceed with honor thesis program, but also helps me get a more thorough understanding of computer science research, especially on the field of sequential pattern mining.

I would also like to thank all of my committee members for having supported, guided and mentored me throughout my career at Emory. I started computer science research journey in Dr. Davide Fossati's lab and was first introduced to pattern mining in his data mining class. Dr. Roberto Franzosi introduced me to the field of natural language processing and provided me with the opportunities on academic collaborations with other students/researchers.

I truly appreciate the opportunities to work with all three professors. Besides, I appreciate the opportunity to work with Dr. Ho's Ph.D student Eric Lee. He also gave me valuable suggestions and helped me on pre-processing data.

# Contents

# Chapter 1

# Introduction

An increasing amount of data is collected that contain the time or sequential ordering information. Such data is prominent in many applications including market basket analysis, text analysis, energy reduction of smarthomes, and clinical decision support. While sequential pattern mining has become a popular tool for these applications, such data ignores the sequential ordering of the events and consequently may fail to discover important or useful patterns [8]. As a motivating example, it is important for a healthcare provider to know the sequence of events (i.e., medical procedures or interventions) that may have resulted in an unfavorable outcome as shown in Figure 1.

Thus, mining sequential patterns or finding patterns that frequently occur in the data is important. To date, researchers have developed exact sequential pattern mining algorithms such as PrefixSpan [15], LAPIN [26], FAST [22], CM-SPADE [6], and CM-SPAM [6]. Unfortunately, there are two notable limitations that prevent the widespread usage of these algorithms on real-world applications: computational complexity (in terms of time and memory) and the generation of trivial frequent subsequences.

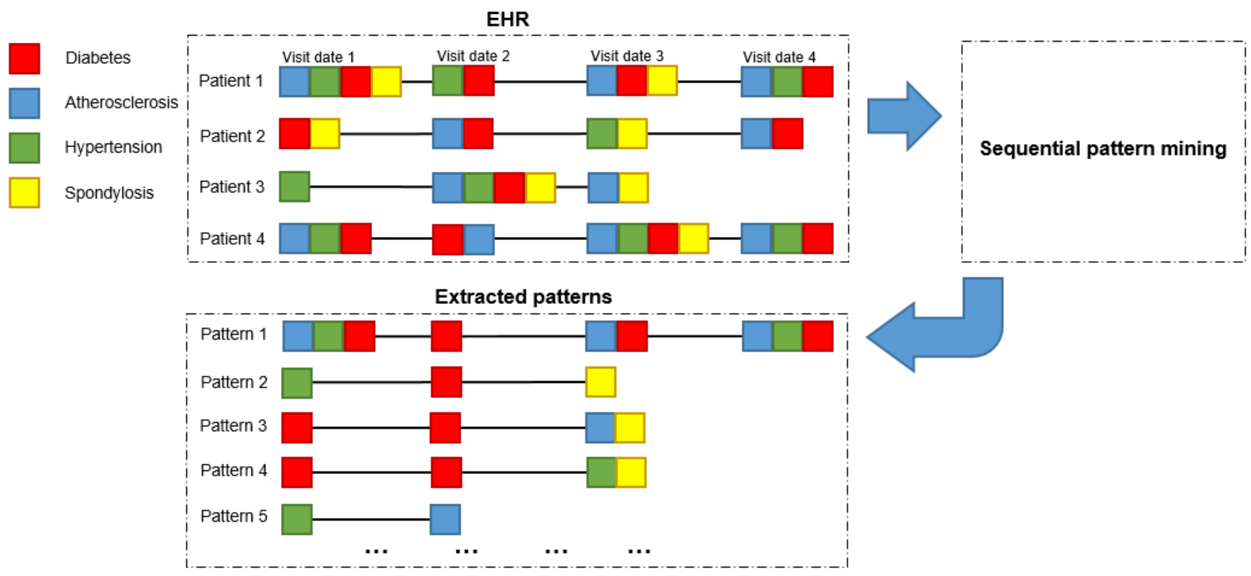

FIGURE 1: An illustrative example of sequential pattern mining in EHR data. Each patient has a sequence of visits and each visit can have multiple diagnoses. Using sequential pattern mining algorithms, frequent patterns can be extracted and only the partial results are shown in the figure.

While several existing works have been shown to be computationally efficient [22, 6], the experimental results are performed on considerably smaller datasets than the large-scale datasets like online data streams and medical datasets. Moreover, the datasets that are predominately used only contain a single item per event. When these exact sequential pattern mining algorithms are applied to electronic health record data, which contains multiple items per patient visit (i.e., event) and long patient sequences (i.e., lots of patient visits), the algorithms often fail to run due to the exponential increase in computational memory and time. Furthermore, the exact sequential pattern mining algorithms will generate trivial sequences with a single item or a series of subsequences which differ from one another only based on one item. The presence of such patterns can negatively impact the adoption of sequential pattern mining as an analysis method as the results can overburden the end-user. Therefore, exact sequential pattern mining may not always be desirable.

Approximate sequential pattern mining was first proposed by Kum *et al.* [16] by designing an algorithm called ApproxMap to mine consensus patterns, which are a subset of long and representative sequential patterns. ApproxMap clustered sequences together based on the similarity among sequences, and then mined consensus patterns from each cluster. Yet, ApproxMap can only mine a small subset of the exact sequential patterns and thus may fail to identify useful or important patterns. More recently, Zhu *et al.* [28] proposed a new algorithm to discover frequent approximate sequential patterns based on the Hamming distance model. They allowed some error as measured by the Hamming distance (equivalent to a small number of mismatches between two sequences with the same length) when mining frequent patterns. Similar to Kum *et al.*, Zhu *et al.* also classified similar sequences into groups. Then they mined out all globally repeating sequential approximate patterns in a local search fashion. Unfortunately, this model also fails to extract a large subset of the exact frequent sequential patterns. GraSeq [18] proposed to mine approximate sequential patterns by converting sequences into a weighted graph. Unfortunately, the algorithm only allows a single directed edge in the graph. This by nature excludes events that have multiple items in one event (or bucket). Therefore, GraSeq can only deal with single-item events. Also, GraSeq produces substantially less sequential patterns than the exact frequent sequential patterns. Although existing approximate sequential pattern mining offers more computationally efficient variants and avoids trivial subsequences, the extracted patterns may not be a reflection of the true sequential patterns.

To address the above limitations, we propose GASP, a novel **G**raph-based **A**pproximate **S**equential **P**attern mining algorithm. We introduce a new graph structure that transforms multi-item event sequences into a weighted graph with both directed and undirected edges. We also develop a new variant of the random walk algorithm to account for multi-item events and the variable length associated with the frequent sequences. Using the graph-based representation and the random walk algorithm, GASP improves the computational efficiency from both the memory and time perspective, while also offering reasonable accuracy with the true sequential patterns. We demonstrate GASP on both single-item and multi-item events and evaluated the performance.

The main contributions of this thesis are:

1. We present a new weighted graph structure to support sequential data that contains multiple items per event. The graph structure compresses the data while retaining much of the ordered sequence information.

2. We introduce a variant of the random walk algorithm to improve the efficiency and the scalability of our algorithm. The algorithm is able to produce close approximations to the true sequential patterns while having a small memory footprint.

3. We conduct experiments on sequential datasets with more than 1 item per event. Most of the approximate sequential pattern mining algorithms, including GraSeq, are designed for single-item events. Moreover, many exact sequential pattern mining algorithms, like CM-SPADE and CM-SPAM, were only tested on such databases. Our evaluation on three large datasets showcases that GASP requires 200% less memory and 5 times faster than CM-SPAM and requires 1000% less memory than CM-SPADE while achieving a similar time.

To begin with, Chapter 2 introduces definitions and notations that will be used throughout the paper. A literature review on both exact sequential pattern mining and approximate sequential pattern mining will also be done in Chapter 2. Chapter 3 elaborates on the proposed algorithm `GASP`. Chapter 4 lists the datasets that we did experiment on and the algorithms that we compared to. The evaluation metric we use to test the performance of `GASP` is also explained in Chapter 4. Chapter 5 reports the comparison results between `GASP` and other existing algorithms. Chapter 6 wraps up the major contributions we make to sequential pattern mining problems, and future paths of the research.

# Chapter 2

# Background

In this section, we present the definition of sequential pattern mining and related works on both exact and approximate sequential pattern mining. We follow the definition introduced by Fournier-Viger *et al.* [9].

## 2.1 Definition and Notation

Given a set of items $I = \{i_1, i_2, ..., i_m\}$, an itemset $X$ is a set of items such that $X \subseteq I$. A sequence $s$ is an ordered list of itemsets such that $s = \langle X_1, X_2, ..., X_n \rangle$. A sequence database $SDB$ is a list of sequences such that $SDB = \langle s_1, s_2, ..., s_p \rangle$ with unique sequence identifiers $1, 2, ..., p$. Table 1 is an example of $SDB$ which contains four sequences. All items in $\{\}$ are one event, and all events in $\langle\rangle$ is one sequence. Sequential pattern mining problem aims at finding frequent subsequences in $SDB$, and subsequence is defined as below.

**Definition 1.** *(**Subsequence**). A sequence $s_1 = \langle a_1, a_2, ..., a_m \rangle$ is a subsequence of another sequence $s_2 = \langle b_1, b_2, ..., b_n \rangle$ if and only if there exist integers $i_1, i_2, ..., i_m$ such that* $1 \leq i_1 \leq i_2 \leq$

Table 1: An example of a sequence Database (SDB).

| SID | Sequences |
|---|---|
| 1 | $\langle\{53,98\},\{58,98\}\rangle$ |
| 2 | $\langle\{257,53\},\{257,58\}\rangle$ |
| 3 | $\langle\{10,53\},\{257,259,58\},\{98\}\rangle$ |
| 4 | $\langle\{10\},\{259,53,58\}\rangle$ |

$... \leq i_m \leq n$ *and* $a_1 \subseteq b_{i_1}, a_2 \subseteq b_{i_2}, ..., a_m \subseteq b_{i_m}$.

By the definition, for example, from $\langle\{53,98\},\{58,98\}\rangle$, we can find a subsequence $\langle\{53\},\{98\}\rangle$. Conventionally, sequential pattern mining requires user-specified support to determine subsequences that satisfy this support (number of sequences that contain the pattern).

The absolute support of sequence $s$ in $SDB$ is the number of sequences that contain $s$. The relative support is the absolute support divided by the total number of sequences in $SDB$ [11]. If the relative support of sequence $s$ exceeds the threshold set by the user, $s$ is considered as a frequent subsequence of $SDB$.

## 2.2 Exact Sequential Pattern Mining

Most of the related studies of sequential pattern mining focus on getting the exact set of subsequences whose support counts are above the user-specified threshold. Since the computational complexity prevents the widespread usage of sequential pattern mining algorithms, researchers made great efforts to develop algorithms with time and space scalability to obtain the exact frequent patterns. A considerable amount of sequential pattern mining algorithms are based on Apriori[2], including AprioriAll [1], GSP [23], SPADE [27]. Another important category is Pattern-Growth-based approaches including PrefixSpan [15] and FreeSpan [14]. More recently, variants of sequential pattern mining techniques have been proposed to achieve better time and space scalability. Some examples of these variations are Closed Sequential Pattern Mining [11, 12, 24, 25], Maximal Sequential Pattern Mining [10, 13, 19, 20]. Closed sequential patterns are the set of sequential patterns that are not included in other frequent patterns with the same support count. Similar to closed sequential patterns, maximal sequential patterns are the set of sequential patterns that are not included in other frequent patterns. Both Maximal and Closed Pattern mining can alleviate the drawback of generating trivial sequential patterns.

In general, two factors are considered frequently in order to cut down the computational complexity. One is the data structure used to represent the original sequential database, and the other is the method to generate or store the next patterns to be developed in the search space. As an example, both SPAM [4] and bitSPADE [3] use IDList representation and encode IDList as bit vectors. IDList is a vertical representation of a sequence database. For each item $i$, it records the itemsets where $i$ appears. The major advantages of IDList is 1. It can be created with one scan of original sequential database; 2. Extending any pattern $s$ with item $i$ doesn't require another scan of the original database. Using such an efficient representation of the SDB avoids scanning database for multiple times. In this way, when generating new patterns, we can avoid maintaining a large number of candidate patterns in memory.

Although some exact sequential pattern mining algorithms are relatively efficient, still they meet difficulties when dealing with long sequences and multi-event sequences. Also, they may generate a huge number of short and trivial patterns. In order to alleviate these problems, researchers developed approximate sequential pattern mining algorithms.

## 2.3 Approximate Sequential Pattern Mining

The goal of approximate sequential pattern mining is to discover similar patterns to the true frequent sequential patterns but using less time and memory. This research area was first proposed by Kum *et al.* [16], where ApproxMap was developed to mine the consensus patterns shared by many sequences.

ApproxMap approached this problem by first clustering sequences based on similarity and then mining consensus patterns from each cluster through multiple alignment [16]. Since consensus patterns are a subset of actual frequent patterns, ApproxMap may fail to recover some important exact sequential patterns. Zhu *et al.* [28] proposed an algorithm for mining approximate sequential patterns under the Hamming Distance model using "break-down-and-build-up" methodology. However, they didn't compare the accuracy of their model with other methods. Chang *et al.* [5] proposed a mining method called eISeq over an online sequence data stream. In eISeq, a lexicographic tree structure is maintained for sequence insertion and frequent sequence selection. Li *et al.* [18] proposed a GraSeq, a graph-based approximate sequential pattern mining algorithm which transformed sequences into a directed weighted graph structure with only one scan of data. Then they introduced a non-recursive depth-first search algorithm to acquire approximate sequential patterns. Both eISeq and GraSeq were designed specifically for online data-stream. In other words, they didn't consider sequential datasets with multi-item event sequences. However, most of the real world datasets are actually multi-item event sequences.

# Chapter 3

# GASP: Graph-based Approximate Sequential Pattern Mining

Existing approximate sequential pattern mining algorithms [5, 16, 18] fail to recover some important exact sequential patterns or mine multi-item event sequences. To address these limitations, we propose GASP, a graph-based approximate sequential pattern mining model. We introduce a new graph structure that transforms multi-item event sequences into a weighted graph using both directed and undirected edges. The graph compresses the sequential information of all the individual sequences without requiring a significant memory footprint. The construction of the graph also captures information to generate variable length frequent sequences. We also extend the random walk algorithm to identify frequent sequential patterns from the SDB. GASP consists of three parts: 1) generate frequent subsequences to encode in the graph; 2) construct a weighted graph based on the frequent subsequences; and 3) use a random walk variant based on probabilities in the graph to identify the frequent sequences. Algorithm 1 presents an overview

**Algorithm 1** GASP

```
    Input: A sequential database T
        number of random walk iterations n
    Output: A list of sequences with weights R =
        {s: w | s is generated by random walk}
 1: $L_1, L_2, P_{start}, P_{end}, P_{length}, P_{trans}$ = GenSubseq(T)
 2: G = constructGraph($L_1, L_2, P_{end}, P_{trans}$)
 3: R = {}
 4: for $i$ = 0 to n do
 5:     $s, w$ = RandomWalk($G, P_{start}, P_{length}, P_{end}, P_{trans}$)
 6:     if $(s : w)$ exists in R then
 7:         increment value of $s$ by $w$
 8:     else
 9:         add $(s : w)$ to R
10:     end if
11: end for
```

of GASP. The following subsections describe the details of our method.

## 3.1 Subsequence Generation

To construct a graph from a SDB, it is necessary to determine the nodes and edges to capture the order or relation between each item. Our graph will encode a single item (hereby referred to as a 1-subsequence), and frequent two item sets (2-subsequence). Since the SDB can have multiple items per event, there are two types of 2-subsequences to distinguish between the scenario where the two items occur in the same event (type 1) and the case where the two items occur in a chronological order (type 2). Thus, our graph structure is a generalization of GraSeq to capture multi-item events.

We formally define the three subsequences as follows:

**Definition 2.** ***(1-subsequence)***.

*For a sequence $s = \langle X_1, X_2, \cdots, X_n \rangle$ where $X_1, X_2, ..., X_n$ are itemsets, a 1-subsequence (denoted*

Table 2: All subsequences generated from Table 1.

| $L_1$ | $L_2-1$ | $L_2-2$ |
|---|---|---|
| $\{58:4\}$, | $\{\langle 53,98\rangle:1\}$, | $\{\langle 53\rangle,\langle 58\rangle\}:3$, $\{\langle 10\rangle,\langle 259\rangle\}:2$ |
| $\{53:4\}$, | $\{\langle 257,58\rangle:2\}$, | $\{\langle 53\rangle,\langle 98\rangle\}:2$, $\{\langle 53\rangle,\langle 257\rangle\}:2$ |
| $\{98:3\}$, | $\{\langle 259,58\rangle:2\}$, | $\{\langle 10\rangle,\langle 58\rangle\}:2$, $\{\langle 98\rangle,\langle 58\rangle\}:1$ |
| $\{257:3\}$, | $\{\langle 58,98\rangle:1\}$, | $\{\langle 98\rangle,\langle 98\rangle\}:1$, $\{\langle 10\rangle,\langle 53\rangle\}:1$ |
| $\{259:2\}$, | $\{\langle 257,53\rangle:1\}$, | $\{\langle 259\rangle,\langle 98\rangle\}:1$, $\{\langle 257\rangle,\langle 257\rangle\}:1$ |
| $\{10:2\}$, | $\{\langle 10,53\rangle:1\}$, | $\{\langle 257\rangle,\langle 58\rangle\}:1$, $\{\langle 10\rangle,\langle 257\rangle\}:1$ |
| | $\{\langle 259,53\rangle:1\}$, | $\{\langle 58\rangle,\langle 98\rangle\}:1$, $\{\langle 10\rangle,\langle 98\rangle\}:1$ |
| | $\{\langle 257,259\rangle:1\}$, | $\{\langle 53\rangle,\langle 259\rangle\}:1$, $\{\langle 257\rangle,\langle 98\rangle\}:1$ |

*as $L_1$) is $L_1 = \langle i_k\rangle$ for all $i_k \in X_1 \cup X_2 \cup \ldots \cup X_n$.*

**Definition 3.** ***(2-subsequence-type-1)***.

*For a sequence $s = \langle X_1, X_2, \ldots, X_n\rangle$ where $X_1, X_2, \ldots, X_n$ are itemsets, a 2-subsequence-type-1 (denoted as $L_2-1$) is $L_2-1 = \langle i_k, i_j\rangle$ for all $i_k, i_j \in X_p$ such that $1 \le p \le n$. Note that $\langle i_k, i_j\rangle$ is equivalent to $\langle i_j, i_k\rangle$*

**Definition 4.** ***(2-subsequence-type-2)***.

*For a sequence $s = \langle X_1, X_2, \ldots, X_n\rangle$ where $X_1, X_2, \ldots, X_n$ are itemsets, a 2-subsequence-type-2 (denoted as $L_2-2$) is $L_2-2 = \langle\langle i_k\rangle,\langle i_j\rangle\rangle$ for all $i_k \in X_p$, $i_j \in X_q$ and $p < q$.*

All of the subsequences generated from Table 1 are shown in Table 2. Note that under our definition, $\langle 53,98\rangle$ is the same subsequence as $\langle 98,53\rangle$. However, $\langle\langle 53\rangle,\langle 58\rangle\rangle$ and $\langle\langle 58\rangle,\langle 53\rangle\rangle$ are considered as two different 2-subsequences (type 2). Furthermore, $\langle 53,98\rangle$ and $\langle\langle 53\rangle,\langle 98\rangle\rangle$ are two different types of 2-subsequences, the former is type 1 while the latter is type 2.

The generation of $L_1$, $L_2-1$, $L_2-2$ is listed as below. In particular, the entire subsequence generation process only requires a single scan through the SDB. The subsequence generation process is detailed in Algorithm 2.

**Algorithm** 2 GenSubseq

```
    Input: A sequential database T
    Output: 1-subsequences $L_1$
        2-subsequences $L_2$
 1: $L_1, L_2 = \{\}$
 2: for each sequence $t \in T$ do
 3:     preItem = []
 4:     k = number of items in $t$
 5:     for each itemset $X \in t$ do
 6:         $m$ = number of items in $X$
 7:         for $i = 0$ to m do
 8:             increment value of $L_1[X[i]]$ by 1
 9:             for $j = i$ to m do
10:                 $L_2$ = processL2(X[i], X[j], 1, $L_2$)
11:             end for
12:             for $l = 0$ to $length(preItem)$ do
13:                 $L_2$ = processL2(preItem[$l$], X[i] 2, $L_2$)
14:             end for
15:         end for
16:         add all items in $X$ to preItem
17:     end for
18: end for
```

$L_1$ **Generation**. The generation process scans all the sequences in the SDB to determine the unique items that occur in all of the sequences. During this process, it also stores the frequency of each subsequence. As a result, every item in the SDB is a member of $L_1$, and the frequency contains the occurrence of each item.

$L_2 - 1$ **Generation**. During the same scan of all the sequences, the generation process also identifies the $L_2 - 1$ subsequences. Since this subsequence focuses on items in the same events, for each sequence $s \in$ SDB, and each itemset $X \in s$, the algorithm generates a set $\{(a, b) \mid a, b \in X\}$. The algorithm also stores the number of occurrences of each item pair, $(a, b)$. The result is a list of all item pairs that occur in the same event and the frequency of these item pairs.

$L_2 - 2$ **Generation**. The generation process for the $L_2 - 2$ subsequences follows the same

**Algorithm 3** processL2

```
    Input: Two items X[i], X[j]
        type $\gamma$
        2-subsequences $L_2$
    Output: updated 2-subsequences $L_2$
1: if $(X[i], X[j], \gamma)$ exists in $L_2$ then
2:     increment value of $(X[i], X[j], \gamma)$ by 1
3: else
4:     add $\{(X[i], X[j], \gamma) : 1\}$ to $L_2$
5: end if
```

procedure as $L_2 - 1$. The main difference is that the two items must occur in two events, and the sequence itself must reflect the actual chronological order. In other words, the algorithm counts the number of occurrences of each item pair $\{(a, b) \mid a \in X_{i_1}, b \in X_{i_2}\}$ where $X_{i_1}$ must occur before $X_{i_2}$. Algorithm 3 details the generation process for the 2-subsequence for both types.

## 3.2 Graph Construction

After the sequence generation is complete, GASP constructs a mixed-type graph to store the sequence information. Each node in the graph represents an item. An edge between two nodes in the graph denotes the relation or ordering between two items.

**Node Representation**. A node in the graph is denoted using then tuple $\langle NID, P_{start}[NID]\rangle$. $NID$ is a unique identifier corresponding to each item in the 1-subsequences ($L_1$). $P_{start}[NID]$ is the probability a frequent pattern will start with this $NID$. Section 3.2.1 illustrates $P_{start}$ in detail.

**Edge Representation**. Each edge between two nodes is denoted using the tuple $\langle a, b, type, weight, P_{trans}, P_{end}\rangle$. $a, b$ are the two nodes (using $NID$) that are connected by the edge. To distinguish between items occurring in the same event or two chronological events, the graph consists of both directed and undirected edges. An undirected edge captures the type 1 subsequence, where two items occur in the same event. A directed edge denotes a type 2 edge, and the direction captures the sequential ordering of the items. The weight of the edge determines the likelihood of traversing to node $b$. $P_{trans}$ denotes the probability of picking a type-1 edge after traversing from $a$ to $b$. This is an essential addition to the graph, as it is important to determine when to transition to a new event. Section 3.2.2 provides additional motivation and details the computation of the transition probability. $P_{end}$ represents the probability of stopping the current pattern conditioned on traversing the edge from $a$ to $b$. Section 3.2.3 discusses the motivation and details of this computation.

### 3.2.1 Start Probability

While the start of a frequent sequential pattern can arise from any item in the sequence, we observe that in the true sequential patterns, items that occur more frequently in the SDB are more likely to serve as the beginning of such patterns. Thus, to simulate the exact sequential patterns, the start probability is set to reflect the likelihood of the item occurring in the SDB. Thus, $P_{start}$ is simply the normalized probability across all the items:

$$P_{start} = \frac{L1[i]}{\sum_j L1[j]} \tag{1}$$

To take the graph in Table 1 as an example, the frequency of each node is [58: 0.22, 53: 0.22, 98: 0.17, 257: 0.17, 259: 0.11, 10: 0.11]. Then we just randomly choose a vertex to start with based on this probability (i.e. 58 and 53 are more likely to be chosen, but 259 and 10 still get the chance to be selected).

### 3.2.2 Event Transition Probability

For multi-item events, it is important to distinguish between the probability of selecting an item in the same event or transitioning to a new event. Although the weight of an edge captures the overall probability of traversing to another node, in long event sequences the type-2 edges are likely to dominate in likelihood. In preliminary experiments using electronic health record data, the type-1 edges were dominated by the type-2 edge weights and thus yielded inaccurate single-item event patterns. To remedy this, each edge also contains an event transition probability, $P_{trans}$. The transition probability denotes the likelihood that the next item will be from the same event conditioned on taking the edge. In other words, it will determine whether to stop at the current itemset or continue to add to the current itemset.

The transition probability for the item pair, $(i_1, i_2)$, is calculated as follows:

$$P_{trans} = \begin{cases} \frac{numSameEvent-2}{numSameEvent-2+numFollowEvent} & (i_1, i_2) \text{ is type-1 edge} \\ \\ \frac{numSameEvent-1}{numSameEvent-1+numFollowEvent} & (i_1, i_2) \text{ is type-2 edge} \end{cases} \tag{2}$$

$numSameEvent$ captures the number of items in the current itemset. $numFollowEvent$ contains the number of items that are in the next (chronological) events. The formula is different for type-1 and type-2 because if we just picked a type-1 edge, both $i_1$ and $i_2$ are from the current event. That means there is only $numSameEvent - 2$ items that we can pick from the current itemset. If we just picked a type-2 edge, $i_2$ is in the current itemset while $i_1$ is from the previous itemsets. Thus, we only exclude one from $numSameEvent$. Again, the resulting $P_{trans}$ is the probability that the next item will be from the same event conditioned on taking $(i_1, i_2)$, and $1 - P_{trans}$ is the probability that the next item will be from the next events. The transition probability for all edges is calculated during the 2-subsequence generation process and does not require an additional pass of the data.

### 3.2.3 Edge-Based Ending Probability

We observed that in the true sequential patterns (i.e., patterns identified by exact sequential pattern mining algorithms), some items or item pairs tend to occur towards the end of the sequences while some other items or item pairs occur towards the beginning of the sequence. Since the start probability (Section 3.2.1) captures the latter, we introduce the edge-based ending probability, $P_{end}$, to capture the former. The idea of $P_{end}$ is to encapsulate the likelihood that this particular item or item-pair will terminate the pattern.

For the item pair, $(i_1, i_2)$, the edge-based ending probability is calculated as:

$$P_{end} = \begin{cases} 1 - \frac{numFollowEvent + numSameEvent - 2}{numItem} & (i_1, i_2) \text{ is type-1 edge} \\ 1 - \frac{numFollowEvent + numSameEvent - 1}{numItem} & (i_1, i_2) \text{ is type-2 edge} \end{cases} \tag{3}$$

Using the same definition of the variables as Equation (2), $numSameEvent$ and $numFollowEvent$ are the number of items in the same event and the number of items in the following event respectively. $numItem$ is the total number of items in the sequence. The resulting $P_{end}$ is the probability that $(i_1, i_2)$ will terminate the pattern, and $1 - P_{end}$ is the probability that we will continue to pick edges. Thus, if this item pair tends to occur towards the end of the sequence, the numerator of the second term will tend to be small compared to the total number of items in the sequence, resulting in a large probability of ending. Similar to the transition probability, the edge-ending probability is also calculated during the subsequence generation process. Therefore, each time the item pair occurs, we calculate the associated $P_{end}$ and take the average of the occurrences in all the subsequences.

### 3.2.4 Length-Based Ending Probability

Apriori-based sequential pattern mining algorithms are sensitive to the number of events(denoted as length for simplicity) in the candidate patterns. The first observation is that the memory complexity of the algorithms increases exponentially with an increase in the length of the patterns. Additionally, the size of the candidate patterns substantially shrinks as many of the candidates are pruned due to their support counts falling below the minimum support count. To ensure our graph structure encapsulates the shrinking of the patterns (i.e., less candidates with larger lengths), we introduce a length-based ending probability, $P_{length}$ based on the number of events. The length-based ending probability serves two purposes: (1) to minimize the occurrence of trivial patterns (1 or 2 event patterns) and (2) to minimize the occurrence of extremely long patterns. Thus, the length-based cumulative distribution function will approximately be similar to Figure 2.

As shown from the figure, the probability of ending the sequence with a short length

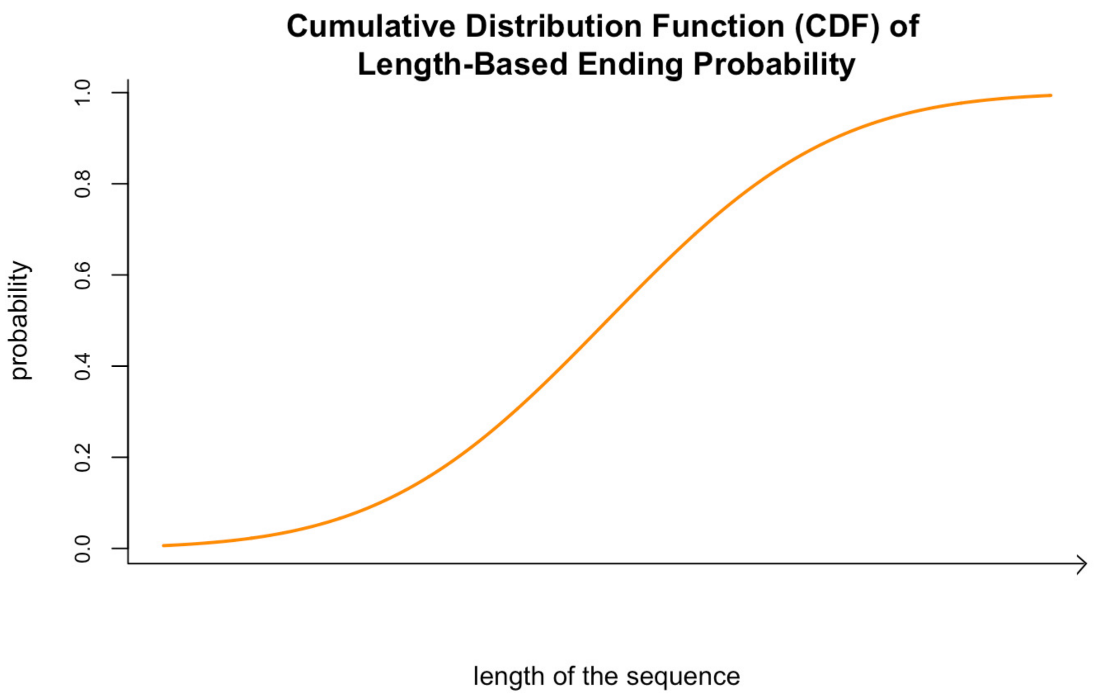

Figure 2: The cumulative distribution function of the length-based probability, $P_{length}$.

is small, whereas the probability of ending will be high for long sequences. This distribution is approximated by counting the total number of events in each sequence, and normalizing the counts such that the cumulative distribution ends in 1.

Again, for the SDB in Table 1, the resulting probability is {2: 0.75, 3: 1}.

### 3.2.5 Graph Construction Algorithm

Given the subsequence information and the different probabilities described above, the graph structure for the dataset can be constructed. First, the nodes are added based on the 1-subsequences ($L_1$), with the starting probability set to be the normalized frequency. Next, each item pair in the 2-subsequences ($L_2 - 1$ and $L_2 - 2$) is enumerated to add the appropriate edge, event transition probability, and edge-based ending prob-

**Algorithm 4** constructGraph

```
    Input: 1-subsequences $L_1$
        2 subsequences $L_2$
        Transition Probability $P_{trans}$
        Ending Probability $P_{end}$
    Output: Graph G
1: for $\{a : w\}$ in $L_1$ do
2:     G.addVertex(a, w)
3: end for
4: for $\{(a, b, type) : w\}$ in $L_2$ do
5:     G.addEdge(a, b, type, w)
6:     G.setTransProb(a, b, type, $P_{trans}[(a, b, type)]$)
7:     G.setEndProb(a, b, type, $P_{end}[(a, b, type)]$)
8: end for
```

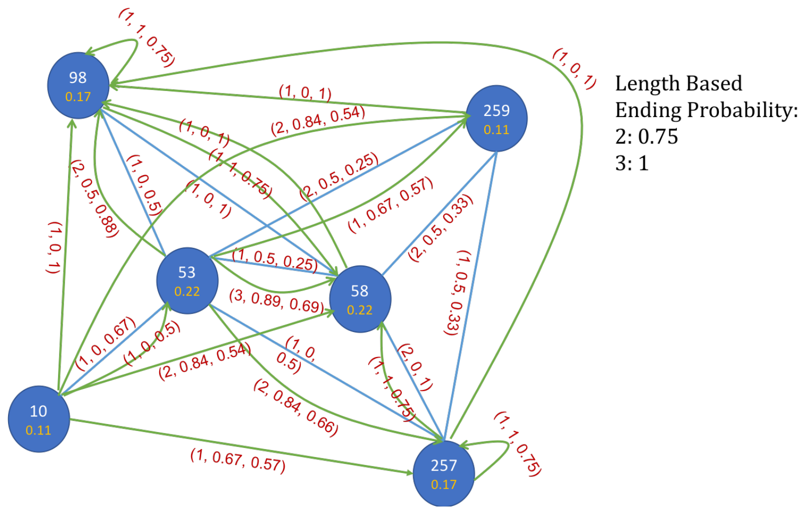

FIGURE 3: Constructed Graph from Table 2. Inside each node, the number in white is the $NID$ of each node, and the number in gold is $P_{start}[NID]$. Blue line denotes type-1 edge and green line with arrow denotes type-2 edge. The three numbers in dark red denote weight, $P_{trans}$, and $P_{end}$ respectively.

ability. Finally, the graph also stores the length-based ending probability as a separate variable. The graph construction algorithm is presented in Algorithm 4. Note that all four probabilities $P_{start}$, $P_{trans}$, $P_{end}$, $P_{length}$ can be calculated within one pass while generating all the subsequences. An example of the graph for Table 1 is shown in Figure 3.

## 3.3 Random Walk

Given the constructed graph, we introduce a random walk variant to traverse the graph and obtain the frequent sequential patterns. Random walk is a natural stochastic process on graphs and was introduced as a mechanism to replace enumerating all potential possibilities while providing a reasonable simulation of the likely paths through the graph [21]. In order to simulate the exact sequential patterns, edges that have higher weights should be traversed more often. Higher weights associated with the edge indicate that the item pair occurs more frequently in the original SDB, thus are more likely to be included in the frequent sequential patterns. GASP adapts the standard random walk algorithm to account for several important aspects of our constructed graph. Since GASP is designed to handle multi-item sequences, a standard random walk is not sufficient to differentiate between the two edge types. Moreover, the stopping criteria of random walk need to be adapted to reflect the two ending probabilities (edge-based and length-based). Therefore, in our customized random walk, the edge weight and the event transition probability, $P_{trans}$, are used to jointly determine the probability of traversing the next edge. Moreover, rather than randomly stopping the sequence, our random walk variant uses the two ending probabilities, $P_{end}$ and $P_{length}$, to determine an appropriate time to stop.

Our random walk proceeds using three basic steps:

1. Randomly pick a start node based on the $P_{start}$ and label it as current node.

2. Among all the neighbors of the current node (i.e., all the edges associated with the current node), randomly select one based on both the edge weight and $P_{trans}$ and

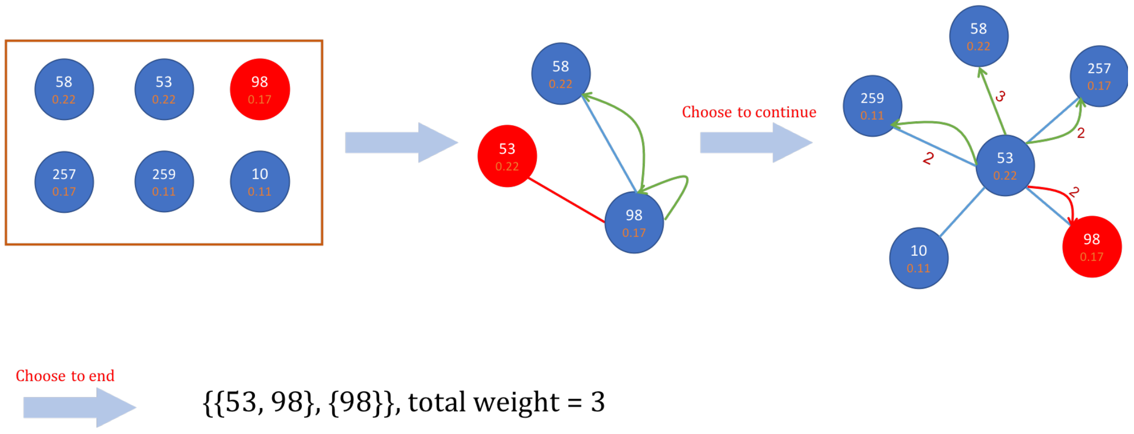

Figure 4: Example of one iteration of random walk.

traverse the selected edge.

3. Based on the $P_{end}$ of the traversed edge and $P_{length}$, randomly decide if the sequence is complete. If yes, store the result sequence and start a new sequence from step 1. If no, continue to step 2 and repeat.

Figure 4 provides an example of one iteration of random walk. First, the starting probability of each node is [58: 0.22, 53: 0.22, 98: 0.17, 257: 0.17, 259: 0.11, 10: 0.11]. If 98 is selected as the start node, we proceed to step 2. Among the 4 possible edges, random walk chooses a type-1 edge with 53, thus yielding the pattern $53, 98$. Based on the completion probability, random walk decides to continue, and thus chooses a type-2 edge of 98.

**Choosing an edge (Step 2)**. The choice of the edge between $a, b$ is dependent on two factors: (1) weight of the edge itself between the current node, $a$ and the next node, $b$ and (2) the $P_{trans}$. The likelihood of selecting the edge is calculated as the product between the weight of the edge and the associated transition probability ($P_{trans}$ for a type-1 edge and $1 - P_{trans}$ for a type-2 edge). Thus, the probability of each edge is the normalized likelihood across all possible edges connected to the current node. For each new iteration of the random walk (i.e, in step 1 of random walk), the $P_{trans}$ is initialized to 0.5, or an equal likelihood of choosing a type 1 edge or a type 2 edge. After selecting the edge, the transition probability is updated to reflect the $P_{trans}$ of the resulting edge.

Using our motivation example, if 98 is selected as the start node, $P_{trans} = 0.5$. The graph is queried to get the weight of the edges between 98 and all its possible neighbors. The weight of type-1 edges is multiplied by 0.5 and the weight of type-2 edges is also multiplied by 0.5, and the values are then normalized to produce a normalized probability. The resulting probabilities will be $[\langle 53,98\rangle : 0.25, \langle 58,98\rangle : 0.25, \langle\langle 98\rangle,\langle 98\rangle\rangle : 0.25, \langle\langle 98\rangle,\langle 58\rangle\rangle : 0.25]$. The probabilities are then used to randomly select which edge to traverse.

**Completion of a pattern (Step 3)**. The choice of determining the end of a random walk pattern is based on two factors: the latest edge selected and the number of events in the current pattern. The probability of a completed pattern is then calculated as the average of the two probabilities, $P_{length}$ and $P_{end}$. Based on this probability, the random walk decides whether the pattern is complete (and to start a new sequence) or if the pattern should continue (i.e., pick another edge to traverse).

Using the same example as above, where the starting node is 98, and random walk selects a type-1 edge with node 53, the edge-ending probability, $P_{end} = 0.5$ and $P_{length}[1] = 0$. Thus, the final completion probability is 0.25.

The detailed steps of our customized random walk are summarized in Algorithm 5.

**Accumulation of Patterns**. Upon the completion of a pattern (as defined by the random walk algorithm), the weights of all the edges that were traversed are summed up to yield the final weight of this particular sequence. As mentioned before, edges with higher weights occur more frequently in the original SDB and are more likely to be included in the frequent sequential patterns. Therefore, a pattern generated from a random walk with higher summed up weights is more likely to have higher support among the patterns generated by exact sequential pattern mining algorithms. Thus, the summed up weights are used for ranking.

If a pattern is generated multiple times by random walk, the weights are accumulated from all the different iterations. The generated pattern and the weight associated with the pattern are stored for quick querying.

Using the motivating example from Figure 4, the generated pattern is $\langle\langle 53, 98\rangle, \langle 98\rangle\rangle$. During the process, there were two edges that were traversed, $\langle 53, 98\rangle, \langle\langle 53\rangle, \langle 98\rangle\rangle$. Since the weight of the first edge is 1 and the weight of the second edge is 2, the final weight of the pattern is 3.

## 3.4 Implementation Details

We implement the algorithms using Python under version 3.6. For the purpose of maintaining computational efficiency and minimizing memory usage, a virtual graph object is not constructed. Instead, the Python dictionary is used to store the nodes, edges, and probabilities for faster queries and also to achieve a compact representation. Since the random walk algorithm can be easily parallelized to account for multiple runs, GASP utilizes multiple threads in our implementation to further reduce the running time on machines with multiple CPUs.

**Algorithm 5** RandomWalk

```
Input: Graph G
       Start Probability $P_{start}$
       Length Distribution $P_{length}$
       Transition Probability $P_{trans}$
       Ending Probability $P_{end}$
Output: A sequence $s$
        Sequence weight $w$
1: v = choose($P_{start}$)
2: $s$ = [v]
3: transP = [0.5, 0.5]
4: $w = 0$
5: numEvent = 1
6: while True do
7:     successors = G.getChild(v)
8:     $P_{next}$ = {}
9:     for each (c, type) ∈ successors do
10:        weight = G.getWeight(v, c, type) * transP[type - 1]
11:        add $\{(c, type) : weight\}$ to $P_{next}$
12:    end for
13:    normalize $P_{next}$
14:    child, type = choose($P_{next}$)
15:    $s = s.append(child)$
16:    $w$ += G.getWeight(v, child, type)
17:    if $type$ == 2 then
18:        increment numEvent by 1
19:    end if
20:    endP = $P_{end}[v, child, type]$
21:    lengthP = $P_{length}[numEvent]$
22:    stop_prob = $(endP + lengthP)/2$
23:    stop = choose($\{1 : stop\_prob, 0 : 1 - stop\_prob\}$)
24:    if stop then
25:        break
26:    end if
27:    transP = $P_{trans}[v, child, type]$
28:    v = child
29: end while
```

# Chapter 4

# Experiment Setup

## 4.1 Dataset

We employed two datasets, FIFA and CMS dataset, to assess the performance of `GASP`. Existing sequential pattern mining algorithms, both exact and approximate, predominantly perform experiments on SDBs with single-event sequences [18, 6] such as FIFA dataset. However, in real-world, we are usually dealing with SDBs with multi-event sequences. Thus, we also construct two different SDB variants of the CMS dataset.

The **FIFA** dataset is a real clickstream data from the website of FIFA World Cup 98. It is obtained directly from the SPMF library[1]. According to Fournier-Viger [6], the FIFA dataset is one of the most time-consuming datasets to extract frequent sequential patterns due to the size of the dataset.

The **CMS** dataset is a synthesized dataset that was adopted from a publicly accessible

[1]`https://www.philippe-fournier-viger.com/spmf/index.php?link=datasets.php`

dataset provided by the Centers for Medicare and Medicaid Services[2].

This dataset contains information about the patients' diagnosis on their visits between the period 2008 to 2009. While the FIFA dataset is employed to assess the performance on single-event sequences, the CMS dataset is used to evaluate on multi-event sequences. Unlike the FIFA dataset which is directly represented as an SDB, the CMS dataset requires an additional preprocessing step to construct the SDB. To construct a SDB, all the patient visits are sorted in chronological order and we focus on the International Classification of Diseases (ICD-9) billing diagnosis codes associated with each visit. It is important to note that each visit (or event) can contain more than one ICD-9 code. On average, a patient visit will contain 2.22 codes.

There can be different representations for the patient representation, such as the date representation, interval representation, etc. [17]. We follow the preprocessing steps detailed in [17] to construct the date representation and interval representation. Specifically, the date representation, the CMS (date) SDB, encodes each unique visit date as a single event. Under the interval representation, the CMS (interval) SDB, each event is constructed by merging visit dates that within a specified interval. For example, if the interval is set to 1 month, then all visits in the same month are merged into a single event. Since the CMS (date) SDB is a large dataset capturing more than 65,000 patients (or sequences) with an average length of 40.96 (or the average number of events), some exact sequential pattern mining algorithms were not able to run on CMS (date) because of the limitation of computational and memory footprint. To compare GASP with existing algo-

[2]https://www.cms.gov/research-statistics-data-and-systems/Downloadable-Public-Use-Files/SynPUFs/DE_Syn_PUF

Table 3: Characteristics of each SDB. The average sequence length refers to the average of number of events within each sequence.

| Dataset | # of sequences | avg. sequence length |
|---|---:|---:|
| CMS (date) | 68,185 | 40.96 |
| CMS (interval) | 68,185 | 16.74 |
| FIFA | 20,450 | 34.74 |

rithms, CMS (interval) is used as it contains the shorter length of sequences than CMS (date). The characteristics of the three SDBs are summarized in Table 3.

## 4.2 Experimental Design

All the experiments were run on a single machine, an Amazon EC2 r5.4xlarge instance, with 16 CPU cores and 128GB memory. To evaluate the performance of GASP, we compared against other sequential pattern mining algorithms from both approximate and exact sequential pattern mining as a baseline. Both GASP and GraSeq, an approximate sequential pattern mining algorithm, is implemented in Python. The SPMF library [7] is used for exact sequential pattern mining and implemented in Java.

### 4.2.1 Approximate Sequential Pattern Mining

As we construct a graph for sequential pattern mining, we compare GASP against GraSeq [18], a graph-based sequential pattern mining algorithm that transforms sequences into a directed weighted graph structure to discover the approximate frequent patterns. The GraSeq implementation is not released, thus, we implement GraSeq in Python. We note that the other approximate sequential pattern mining algorithms described in Section 2 are also not publicly released and thus are not available for comparison. However, GraSeq is designed to deal with only single-item event sequences as it only supports a directed edge in the graph. Thus, the graph in GraSeq only supports the directed edge (constructed using $L_2$−2). We generalized GraSeq to deal with multi-item event sequences by considering items in each event as a single "item" and transform the whole event to a node in the graph. Thus, in contrast to having only one item in each node, now the graph contains multiple items in each node. In this way, the graph can work with multi-item sequences with only directed edges. GraSeq extracts sequential patterns by applying depth-first search over the graph with every node serving as a starting node. A pattern is considered as frequent if (1) the frequency of every item in the pattern is above a user-specified threshold; (2) the frequency of every 2-subsequence in the pattern is also above the threshold; and (3) there is no repeated item in the pattern. In this way, GraSeq extracts only a few patterns than those generated by exact sequential pattern mining algorithms, we enhanced GraSeq to use our proposed random walk step to generate more patterns.

### 4.2.2 Exact Sequential Pattern Mining

Since GASP targets to extract approximate sequential patterns, we also evaluate it based on how many patterns are recovered compared to the exact sequential pattern mining algorithms. We compare the results with three exact sequential pattern mining algorithms.

- **FAST** [6]: An algorithm that improves SPAM [4] but introduced a concept of indexed sparse IDLists which can calculate the support of candidates more quickly.

- **CM-SPAM** [22]: A state-of-the-art sequential pattern mining algorithm that improves SPAM by introducing the concept of co-occurrence pruning using a co-occurrence map to reduce the number of candidate patterns.

- **CM-SPADE** [22]: Similar to CM-SPAM, this is also a state-of-the-art sequential pattern mining algorithm which uses co-occurrence pruning. However, the basic idea follows Spade [27] which uses a vertical database representation while SPAM uses horizontal database representation.

We use the implementation in the SPMF library [7] for CM-SPAM and CM-SPADE. More interesting is that the number of frequent patterns returned from each algorithm were different on the two CMS SDBs. We hypothesize that the discrepancy arises from the fact that these algorithms are designed to deal with singe-item event sequences while the CMS dataset is a multi-item event SDB. We also noticed that FAST returned the most frequent patterns from the SPMF library, and thus use it as the gold standard. However, FAST requires significant memory to extract frequent sequential patterns from CMS (date) and FIFA. Moreover, FAST only works with high user-specified support on our server. Consequently, the algorithm can only identify a relatively small set of sequential patterns. Due to memory limitations (even using a machine with 128 GB of RAM), we benchmark GASP, CM-SPADE, CM-SPAM against FAST on the CMS (interval) dataset. Since we are unable to run FAST on FIFA and CMS (date), we compare GASPto CM-SPADE on CMS (date) and FIFA dataset.

## 4.3 Evaluation Metric

To evaluate GASP against other algorithms, we compared the results from three perspectives: recoverability, running time, and memory usage.

### 4.3.1 Frequent Patterns Recoverability

Zhu *et al.* [28] proposed an approximate sequential pattern mining algorithm that works with long sequences in multi-item event sequences. The authors suggested using error tolerance to determine which subsequences generated by their method should be included in the set of approximate frequent patterns. In other words, the error is the percentage of the dissimilarity between two patterns. Given the idea of error tolerance, as GASP also target multi-event sequences, we want a more direct measurement to see how dissimilar two sequences are and how well GASP recovered frequent patterns. Thus Levenshtein distance is used.

The definition is given below:

**Definition 5.** ***(Levenshtein distance)*.**

*Given two sequences $s_1, s_2$, the Levenshtein distance between them is the minimum number of single-item edits, including insertions, deletions and substitutions, required to change $s_1$ to the $s_2$ or vice versa.*

More formally, the Levenshtein distance between string a and b is given by $lev_{a,b}(\mid a \mid, \mid$

$b \mid$), where $\mid a \mid, \mid b \mid$ are the lengths of $a$ and $b$ such that:

$$lev_{a,b}(i,j) = \begin{cases} max(i,j) & \text{if min(i, j) = 0,} \\ min \begin{cases} lev_{a,b}(i-1,j)+1 \\ lev_{a,b}(i,j-1)+1 \\ lev_{a,b}(i-1,j-1)+1_{a_j \neq b_j} \end{cases} & \text{otherwise.} \end{cases} \tag{4}$$

To compute the Levenshtein distance between two sequences $a$, $b$, we first transform both $a$ and $b$ to strings. For instance, $\langle\langle 53, 98\rangle, \langle 10\rangle\rangle$ is transferred to string "53 98 10". Then, the editing distance between $\langle\langle 53, 98\rangle, \langle 10\rangle\rangle$ and $\langle\langle 53\rangle, \langle 10\rangle\rangle$ is 1. Note that the editing distance between $\langle\langle 53, 98\rangle, \langle 10\rangle\rangle$ and $\langle\langle 53, 98\rangle\rangle$ is also 1.

In order to compare patterns generated by GASP or GraSeq to those generated by exact sequential pattern mining algorithms, we take the average of Levenshtein distances between two lists of patterns. For example, when comparing GASP with CM-SPADE, for each frequent pattern $s$ in CM-SPADE, we look for the closest match of $s$ on the list of patterns generated by GASP. A sequence is considered as the closest match if it has the minimum Levenshtein distance with $s$. Then, we take an average of the Levenshtein distances between frequent patterns from CM-SPADE and their closest matches.

### 4.3.2 Computational Time

The computational time considers the total time of the algorithm. For GASP, a Python timer is used to calculate the difference between the beginning of the program and the end of the program, and thus includes the subsequence generation, graph construction, and random walk. For CM-SPADE, CM-SPAM, and FAST, the implementation in the SPMF library provides the running time of these algorithms.

### 4.3.3 Memory Usage

We use memory-profiler 0.57.0 to monitor the memory consumption of every algorithm. Memory-profiler checks the memory usage of the program every 0.5 seconds and we report the max memory. Note that in the initial implementation of CM-SPADE, CM-SPADE and FAST, Fournier-Viger *et al.* sample limited time-points to check the memory usage. Thus, the max memory reported by SPMF fluctuates between different runs and may not be accurate. To address this limitation, we also use memory-profiler to measure the max memory used by the SPMF-based algorithms.

# Chapter 5

# Empirical Results

## 5.1 Evaluation with Approximate Sequential Pattern Mining

### 5.1.1 CMS (date) Dataset

Both GraSeq and GASP are run on CMS (date) with the same number of iterations (0.1M) using our customized random walk algorithm. The output of the two algorithms is compared with the sequential patterns generated by CM-SPADE using the support of 20%. Table 4 shows the average Levenshtein distance for GASP and GraSeq to CM-SPADE. For every two lists of sequences that we are comparing, we take first top K% sequences to compute the average Levenshtein distance. The editing distance between CM-SPADE and GraSeq is much larger than the editing distance between CM-SPADE and GASP on every Top K percent of generated sequences.

Table 4: The average Levenshtein distance from CM-SPADE to GASP and GraSeq on CMS (date). The iteration number of random walk is 0.1M.

| TopK | 10% | 20% | 30% | 40% | 50% | 60% | 70% | 80% | 90% | 100% |
|---|---|---|---|---|---|---|---|---|---|---|
| GASP | 1.239 | 1.196 | 1.184 | 1.181 | 1.182 | 1.195 | 1.206 | 1.220 | 1.231 | 1.240 |
| GraSeq | 2.034 | 1.971 | 1.978 | 1.987 | 1.994 | 2.006 | 2.015 | 2.025 | 2.035 | 2.042 |

Table 5: The average Levenshtein distance from CM-SPADE to GASP and GraSeq on FIFA. The iteration number of random walk is 0.5M.

| TopK | 10% | 20% | 30% | 40% | 50% | 60% | 70% | 80% | 90% | 100% |
|---|---|---|---|---|---|---|---|---|---|---|
| GASP | 1.371 | 1.432 | 1.543 | 1.565 | 1.678 | 1.701 | 1.759 | 1.794 | 1.823 | 1.901 |
| GraSeq | 1.532 | 1.662 | 1.72 | 1.745 | 1.748 | 1.823 | 1.869 | 1.905 | 2.101 | 2.15 |

### 5.1.2 FIFA Dataset

A similar experiment as CMS (date) is performed on the FIFA dataset. We assess the performance of GraSeq and GASP by comparing them to CM-SPADE on 5% support. Both GraSeq and GASP are run with 0.5M iterations for a random walk.

As Table 5 presents, GASP still outperforms GraSeq on every top K%, but the difference between these two methods decreases. Since GraSeq was initially designed to deal with single-event SDBs, it is expected that GraSeq can achieve similar performance on FIFA than on CMS (date).

## 5.2 Evaluation with Exact Sequential Pattern Mining

### 5.2.1 CMS (interval) Dataset

We observed that on CMS (interval) dataset, the exact sequential pattern mining algorithms presented in the SPMF library return a different number of frequent patterns.

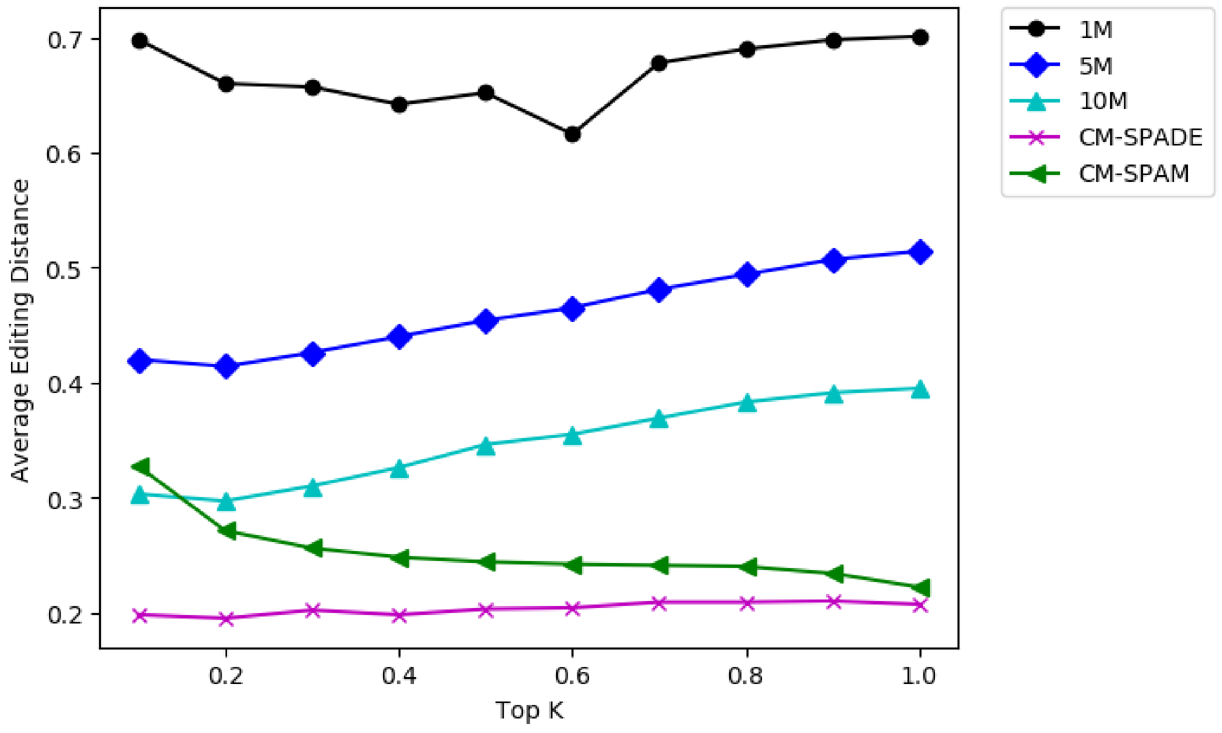

FIGURE 5: Average Levenshtein distance from FAST to CM-SPADE, CM-SPAM, GASP with 1 million, 5 million, 10 million iterations respectively on CMS (interval)

Using a 20% support, FAST, CM-SPADE, CM-SPAM, PrefixSpan return 74,065, 67,708, 63,520, 64,229 patterns respectively. That means, on multi-event sequential datasets like CMS, these algorithms may not be exact as well. FAST is the algorithm that yields the most frequent patterns. Thus, we compare GASP, CM-SPADE, and CM-SPAM against FAST to see how far away these algorithms are from the "truth".

Figure 5 shows the average Levenshtein distance from FAST to CM-SPADE, CM-SPAM, and GASP with 1M, 5M, 10M iterations respectively. GASP with 10M iterations perform very close to CM-SPADE and CM-SPAM on Top 30%, and even in Top 100%, the difference in Levenshtein distance doesn't exceed 0.2. Moreover, Figure 5 shows that as we increase the random walk's iteration number, the distance between GASP and FAST drops substantially.

Table 6: Comparison of time and memory usage of FAST, CM-SPADE, CM-SPAM, GASP with 1 million, 5 million, and 10 million iterations respectively on CMS (interval). The memory is in megabyte and time is in seconds.

| Model | Time (s) | Memory (MB) |
|---|---|---|
| FAST | 2240 | 22296 |
| CM-SPAM | 2080 | 1968 |
| CM-SPADE | 321 | 10747 |
| GASP-1M | 286 | 347 |
| GASP-5M | 627 | 763 |
| GASP-10M | 1055 | 1756 |

Table 6 presents the memory and time comparison between these algorithms. From the results, CM-SPADE generates the most similar patterns as FAST does and it finishes almost 700% times faster than FAST. However, it uses 10 times more memory than GASP does even when considering the 10M iterations of random walk

Compared to CM-SPAM, which is based on a similar logic, CM-SPADE is fast because it is parallelized in the implementation. Also, it makes a trade-off between time and memory based on the fact that CM-SPADE uses almost 10 times more memory than CM-SPAM and GASP do. In the most top K% comparison, CM-SPAM is the second closest algorithm to FAST. However, it is 3 times slower than GASP with 5 million iterations, and 2 times slower than GASP with 10 million iterations.

### 5.2.2 CMS (date) Dataset

For the CMS (date) dataset, as our server (with 128GB RAM) cannot run FAST on low support count (< 30% support count) because of insufficient memory, we run CM-SPAM and CM-SPADE using a support of 20%. We choose a low support because we want to get as many frequent patterns as we can so that we can test thoroughly the recoverability

Table 7: Comparison of time and memory usage of CM-SPADE, CM-SPAM, GASP with 0.1 million, 1 million, 5 million, 10 million iterations respectively on CMS (date). The memory is in megabyte and time is in seconds.

| Model | Time (s) | Memory (MB) |
|---|---|---|
| CM-SPAM | 3798 | 1937 |
| CM-SPADE | 815 | 11008 |
| GASP-0.1M | 220 | 399 |
| GASP-1M | 292 | 485 |
| GASP-5M | 604 | 887 |
| GASP-10M | 965 | 1856 |

of GASP. As a result, CM-SPADE gives us 127941 frequent patterns and CM-SPAM gives 124776 frequent patterns. Since CM-SPADE gives us more patterns, we benchmark CM-SPAM and GASP against CM-SPADE. It is worth noting that the patterns generated by CM-SPADE may not encompass all of the frequent sequential patterns. These are the closest approximation to the actual frequent patterns.

First, we compare the computation time and memory usage of CM-SPADE, CM-SPAM, and GASP with 0.1 million, 1 million, 5 million, and 10 million iterations in random walk respectively are shown in Table 7. It's clear that even though the running time of CM-SPADE is comparable to GASP , it uses 2,759%, 2,270%, 1,241% and 593% memory as GASP uses with 0.1M, 1M, 5M, 10M iterations respectively. For CM-SPAM, although it has comparable memory usage as GASP , it uses 1,726%, 1,300%, 629% and 394% time as GASP uses with 0.1M, 1M, 5M, 10M iterations respectively.

Figure 6 presents the average Levenshtein distance from CM-SPADE to CM-SPAM and GASP with 0.1 million, 1 million, 5 million, and 10 million iterations respectively. CM-SPAM generates the closest patterns as CM-SPADE generates. GASP with 10M iterations are close to CM-SPAM on Top 10% and Top 20%.

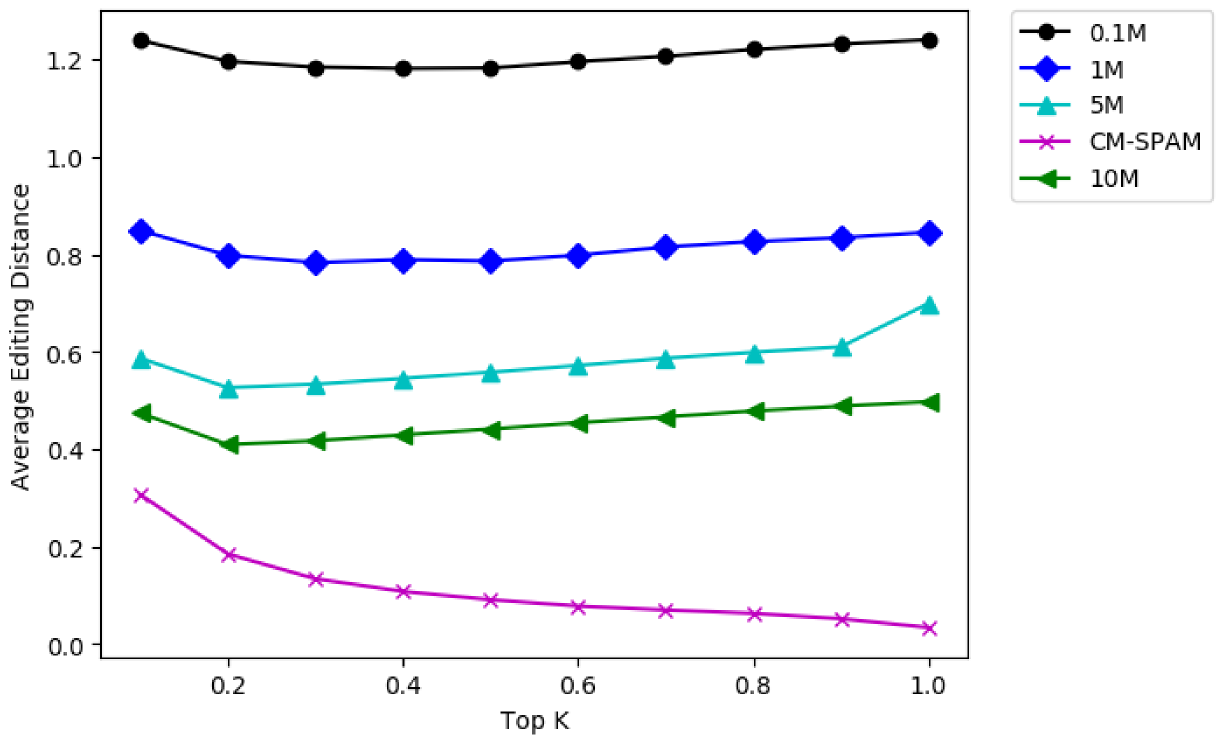

Figure 6: Average Levenshtein distance from CM-SPADE to CM-SPAM, GASP with 0.1, million, 1 million, 5 million, 10 million iterations respectively on CMS (date).

Table 8: The average Levenshtein distance from CM-SPADE to GASP with 5M iterations.

| TopK | 10% | 20% | 30% | 40% | 50% | 60% | 70% | 80% | 90% | 100% |
|---|---|---|---|---|---|---|---|---|---|---|
| GASP-5M | 0.701 | 0.689 | 0.714 | 0.801 | 0.876 | 0.989 | 1.15 | 1.223 | 1.345 | 1.452 |

### 5.2.3 FIFA Dataset

The same experiment is done on the FIFA dataset to assess the performance of GASP on single-event sequences. On such datasets, CM-SPADE, CM-SPAM, and FAST give us the same amount of frequent patterns using the same support. Thus, we compare GASP to the exact frequent patterns generated by them to see how many patterns GASP can recover.

Table 8 indicates GASP performs worse on FIFA than on CMS. One possibility is that we are trying to recover 910422 frequent patterns, which is a hard task. The result also indicates that GASP is more appropriate to be used on a multi-event sequential database.

Table 9: Comparison of time and memory usage of CM-SPADE, CM-SPAM, GASP with 0.5 million, 1 million, 5 million, 10 million iterations respectively on FIFA. The memory is in megabyte and time is in seconds.

| Model | Time (s) | Memory (MB) |
|---|---:|---:|
| CM-SPADE | 645 | 7406 |
| CM-SPAM | 2247 | 2601 |
| GASP-0.5M | 81 | 459 |
| GASP-1M | 141 | 536 |
| GASP-5M | 579 | 942 |
| GASP-10M | 1194 | 1201 |

Table 9 shows the time and memory usage comparison between CM-SPADE, CM-SPAM and GASP with 0.5 million, 1 million, 5 million, and 10 million iterations. We choose 0.5 million as a starting point because exact sequential pattern mining algorithms return 910,422 frequent patterns. 0.1 million iterations cannot produce a comparable list of patterns to such a great amount of frequent patterns. Since GASP with 5M iterations have similar running time to CM-SPADE and much smaller memory usage than both CM-SPADE and CM-SPAM, we showcase the average Levenshtein distance from CM-SPADE to GASP with 5M iterations.

# Chapter 6

# Conclusions

In this thesis, we propose GASP, a new approach for approximate sequential pattern mining. We present a new weighted graph structure using both directed and undirected edges which compresses the sequential information. Also, we introduce a variant of a random walk model that uses the probability in the graph to identify frequent patterns. This novel approach shows a significant improvement in both time and memory usage with a great performance on recoverability which denotes that GASP works for SDB with a long sequence length. Our empirical evaluations verify that GASP is capable of mining both single-item and multi-item event sequences.

We leave implementing a more efficient random walk as future work. Extensive research has been done on optimizing random walk. And adapting or improving those techniques into our customized random walk method can enhance the overall performance by running with more number iterations in a short computational time with a better frequent pattern recoverability. Since CMS is a synthesized dataset, further experiments on real dataset with multi-item events can help us test the performance of

`GASP` more extensively.

# Bibliography

[1] R. Agrawal and R. Srikant. Mining sequential patterns. *Proceedings of the Eleventh International Conference on Data Engineering*, pages 3–14, 1995.

[2] R. Agrawal, R. Srikant, et al. Fast algorithms for mining association rules. In *Proc. 20th int. conf. very large data bases, VLDB*, volume 1215, pages 487–499, 1994.

[3] S. Aseervatham, A. Osmani, and E. Viennet. bitspade: A lattice-based sequential pattern mining algorithm using bitmap representation. In *Sixth International Conference on Data Mining (ICDM'06)*, pages 792–797. IEEE, 2006.

[4] J. Ayres, J. Flannick, J. Gehrke, and T. Yiu. Sequential pattern mining using a bitmap representation. In *Proceedings of the eighth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 429–435, 2002.

[5] J. H. Chang and W. S. Lee. Efficient mining method for retrieving sequential patterns over online data streams. *Journal of Information Science*, 31(5):420–432, 2005.

[6] P. Fournier-Viger, A. Gomariz, M. Campos, and R. Thomas. Fast vertical mining of sequential patterns using co-occurrence information. In *Pacific-Asia Conference on Knowledge Discovery and Data Mining*, pages 40–52. Springer, 2014.

[7] P. Fournier-Viger, J. C.-W. Lin, A. Gomariz, T. Gueniche, A. Soltani, Z. Deng, and H. T. Lam. The spmf open-source data mining library version 2. In *Joint European conference on machine learning and knowledge discovery in databases*, pages 36–40. Springer, 2016.

[8] P. Fournier-Viger, J. C.-W. Lin, R. U. Kiran, Y. S. Koh, and R. Thomas. A survey of sequential pattern mining. *Data Science and Pattern Recognition*, 1(1):54–77, 2017.

[9] P. Fournier-Viger, J. C.-W. Lin, B. Vo, T. T. Chi, J. Zhang, and H. B. Le. A survey of itemset mining. *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery*, 7(4):e1207, 2017.

[10] P. Fournier-Viger, C.-W. Wu, and V. S. Tseng. Mining maximal sequential patterns without candidate maintenance. In *International Conference on Advanced Data Mining and Applications*, pages 169–180. Springer, 2013.

[11] F. Fumarola, P. F. Lanotte, M. Ceci, and D. Malerba. CloFAST: closed sequential pattern mining using sparse and vertical id-lists. *Knowledge and Information Systems*, 48(2):429–463, 2016.

[12] A. Gomariz, M. Campos, R. Marin, and B. Goethals. Clasp: An efficient algorithm for mining frequent closed sequences. In *Pacific-Asia Conference on Knowledge Discovery and Data Mining*, pages 50–61. Springer, 2013.

[13] E.-Z. Guan, X.-Y. Chang, Z. Wang, and C.-G. Zhou. Mining maximal sequential patterns. In *2005 International Conference on Neural Networks and Brain*, volume 1, pages 525–528. IEEE, 2005.

[14] J. Han, J. Pei, B. Mortazavi-Asl, Q. Chen, U. Dayal, and M.-C. Hsu. Freespan: frequent pattern-projected sequential pattern mining. In *Proceedings of the sixth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 355–359, 2000.

[15] J. Han, J. Pei, B. Mortazavi-Asl, H. Pinto, Q. Chen, U. Dayal, and M. Hsu. Prefixspan: Mining sequential patterns efficiently by prefix-projected pattern growth. In *proceedings of the 17th international conference on data engineering*, pages 215–224. Citeseer, 2001.

[16] H.-C. Kum, J. Pei, W. Wang, and D. Duncan. Approxmap: Approximate mining of consensus sequential patterns. In *Proceedings of the 2003 SIAM International Conference on Data Mining*, pages 311–315. SIAM, 2003.

[17] E. Lee and J. Ho. Fuzzygap: Sequential pattern mining for predicting chronic heart failure in clinical pathways. *AMIA Joint Summits on Translational Science proceedings. AMIA Joint Summits on Translational Science*, 2019:222, 2019.

[18] H. Li and H. Chen. Graseq: A novel approximate mining approach of sequential patterns over data stream. In *International Conference on Advanced Data Mining and Applications*, pages 401–411. Springer, 2007.

[19] S. Lu and C. Li. Aprioriadjust: An efficient algorithm for discovering the maximum sequential patterns. In *Proc. Intern. Workshop knowl. Grid and grid intell*, 2004.

[20] C. Luo and S. M. Chung. Efficient mining of maximal sequential patterns using multiple samples. In *Proceedings of the 2005 SIAM International Conference on Data Mining*, pages 415–426. SIAM, 2005.

[21] K. Pearson. The problem of the random walk. *Nature*, 72(1867):342–342, 1905.

[22] E. Salvemini, F. Fumarola, D. Malerba, and J. Han. Fast sequence mining based on sparse id-lists. In *International Symposium on Methodologies for Intelligent Systems*, pages 316–325. Springer, 2011.

[23] R. Srikant and R. Agrawal. Mining sequential patterns: Generalizations and performance improvements. In *International Conference on Extending Database Technology*, pages 1–17. Springer, 1996.

[24] J. Wang, J. Han, and C. Li. Frequent closed sequence mining without candidate maintenance. *IEEE Transactions on Knowledge and Data Engineering*, 19(8):1042–1056, 2007.

[25] X. Yan, J. Han, and R. Afshar. Clospan: Mining: Closed sequential patterns in large datasets. In *Proceedings of the 2003 SIAM international conference on data mining*, pages 166–177. SIAM, 2003.

[26] Z. Yang and M. Kitsuregawa. Lapin-spam: An improved algorithm for mining sequential pattern. In *21st International Conference on Data Engineering Workshops (ICDEW'05)*, pages 1222–1222. IEEE, 2005.

[27] M. J. Zaki. Spade: An efficient algorithm for mining frequent sequences. *Machine learning*, 42(1-2):31–60, 2001.

[28] F. Zhu, X. Yan, J. Han, and S. Y. Philip. Efficient discovery of frequent approximate sequential patterns. In *Seventh IEEE International Conference on Data Mining (ICDM 2007)*, pages 751–756. IEEE, 2007.