Memory-Efficient Sequential Pattern Mining with Hybrid Tries

Amin Hosseininasab　　　　　　　　　　AMIN.HOSSEININASAB@WARRINGTON.UFL.EDU
Warrington College of Business
University of Florida
Gainesville, FL, USA

Willem-Jan van Hoeve　　　　　　　　　　VANHOEVE@ANDREW.CMU.EDU
Tepper School of Business
Carnegie Mellon University
Pittsburgh, PA, USA

Andre A. Cire　　　　　　　　　　ANDRE.CIRE@ROTMAN.UTORONTO.CA
Rotman School of Management
University of Toronto
Toronto, ON, Canada

Abstract

This paper develops a memory-efficient approach for Sequential Pattern Mining (SPM), a fundamental topic in knowledge discovery that faces a well-known memory bottleneck for large data sets. Our methodology involves a novel hybrid trie data structure that exploits recurring patterns to compactly store the data set in memory; and a corresponding mining algorithm designed to effectively extract patterns from this compact representation. Numerical results on small to medium-sized real-life test instances show an average improvement of 85% in memory consumption and 49% in computation time compared to the state of the art. For large data sets, our algorithm stands out as the only capable SPM approach within 256GB of system memory, potentially saving 1.7TB in memory consumption.

Keywords: Sequential pattern mining, Memory efficiency, Large-scale pattern mining, Trie data set models.

1. Introduction

Data volume is growing at an exponential rate (Taylor, 2021), with modern machine learning data sets often containing trillions of data points (Villalobos and Ho, 2022). While supervised machine learning algorithms have thrived by training on such large data sets, unsupervised algorithms face ongoing challenges in scalability due to their memory requirements. In particular, Sequential Pattern Mining (SPM), a prominent topic in unsupervised learning, encounters a well-known memory bottleneck in its two most prevalent algorithms. The Apriori algorithm (Agrawal et al., 1994) suffers from the explosion of candidate patterns that are costly to store in memory, and the prefix-projection algorithm (Han et al., 2001) requires fitting the entire training data set into memory. This has limited extant SPM algorithms to smaller-sized data sets and rendered them impractical for larger ones (Pillai and Vyas, 2011).

Larger data sets are inherently richer in information, and mining them can uncover intricate patterns that facilitate a deeper understanding of the relationships in data. This includes rare-event patterns and those with long-term dependencies that are more prevalent in larger samples of the population, but may be infrequent in smaller subsets. Such patterns are of interest in numerous applications, such as fraud detection (Kim et al., 2022), medical research (Ji et al., 2012), bioinformatics (Béchet et al., 2012), and market basket analysis (Pillai and Vyas, 2011), to name a few. Additionally, patterns mined from a larger sample have less variance and are statistically more robust than the same patterns mined from a smaller subset (Hämäläinen and Nykänen, 2008). This highlights the need for memory-efficient SPM algorithms that can handle larger data sets and adapt to their rapidly growing data environment.

The primary approach to applying SPM algorithms on large data sets is by using more hardware, either on a single computing unit or via a parallelized structure, such as Gan et al. (2019); Huynh et al. (2018); Chen et al. (2017); Yu et al. (2019); Saleti and Subramanyam (2019); Chen et al. (2013). However, using additional hardware is costly, requires specialized machinery in the case of parallelization, and is often capped due to technology and other system specifications. Maximizing performance under capped memory is thus a practical use-case that is of interest in SPM (Wang et al., 2003), and has been shown to be beneficial in various machine learning techniques such as Pleiss et al. (2017); Gruslys et al. (2016); Si et al. (2017). Benefits include, for example, faster data set access and higher time efficiency, reduced overhead and optimized resource utilization, reduced hardware maintenance costs, facilitation of use in dynamic and online data streams, reduced environmental impact, and higher energy efficiency. Most importantly, a memory-efficient algorithm makes SPM available to a broader range of users who are constrained by hardware limitations.

Motivated by these benefits, this paper aims to enhance the memory efficiency of SPM algorithms, while simultaneously preserving or increasing their time efficiency. We focus on the prefix-projection algorithm, which has been shown to improve over other alternative SPM algorithms in terms of time efficiency (Han et al., 2001). The memory bottleneck of prefix-projection arises from the necessity to fit the entire data set into memory, prompting a critical examination of how sequential data sets are modeled and stored. The dominant approach is to model the data set using a relational or *vector* model, such as the ones discussed in Fournier-Viger et al. (2017). An advantage of vector models is their simple structure, leading to a straightforward mining process. Nevertheless, a major disadvantage of vector models is their high memory usage which becomes increasingly inefficient as data sets grow in size. This has led researchers to explore alternative models such as *trie* structures which are known for their concise representation of strings.

A trie is a graph-based data structure commonly used to store associative arrays or sets of strings in an efficient manner. A major advantage of trie models is their ability to model multiple overlapping subsequences using only a single trie path, which can potentially lead to higher time and memory efficiency in the mining algorithm (Mabroukeh and Ezeife, 2010). For example, trie models of data sets have been shown to provide benefits such as faster item-set mining (Han et al., 2004; Pyun et al., 2014; Borah and Nath, 2018), effective Apriori and candidate pattern storage (Ivancsy and Vajk, 2005; Pyun et al., 2014; Huang et al., 2008; Fumarola et al., 2016; Antunes and Oliveira, 2004; Wang et al., 2006; Bodon and Rónyai, 2003; Masseglia et al., 2000), effective web access mining (Yang et al., 2007; Pei et al., 2000;

Lu and Ezeife, 2003), mining long biological sequences (Liao and Chen, 2014), up-down SPM (Chen, 2009), incorporating constraints into SPM (Masseglia et al., 2009; Hosseininasab et al., 2019; Wang et al., 2022; Kadıoğlu et al., 2023), progressive SPM (Huang et al., 2006; El-Sayed et al., 2004), and faster SPM (Rizvee et al., 2020).

While trie models can provide a more memory-efficient model of the data set, using that model for SPM poses an entirely different challenge. Unlike vector models which have a one-to-one correspondence between their vectors and sequences of the data set, a single path of a trie model may correspond to several sequences. This makes tracking the frequency of patterns nontrivial, requiring additional information to be stored at the nodes of the trie. In fact, the distinguishing factor between the many trie-based approaches in the literature is the information stored at the nodes of their tries and how it is used in the mining algorithm. Unfortunately, all current trie models involve storing a rich set of information at their nodes in favor of time efficiency, often increasing their memory requirements beyond that of vector models.

For example, in web access mining (a special case of SPM with simplified sequence structures), Pei et al. (2000) propose to use a hash table of linked nodes to traverse and scan their trie data set model. Using hash tables, the data set is recursively projected onto conditional smaller tries and mined accordingly. The disadvantage of this approach is the memory overhead of the hash table and the memory and time spent constructing the conditional tries. Instead of constructing conditional tries, Lu and Ezeife (2003) propose to store at each node of the trie model an integer position value, and Yang et al. (2007) propose to recursively generate sub-header tables in the mining algorithm. Such integer position values grow exponentially with the size of the trie, and similarly, the generation of many sub-header tables leads to higher memory consumption.

Following works in web access mining, Rizvee et al. (2020) extend the trie model of a sequential data set to accommodate SPM. This is a challenging task, due to the different structural properties of data sets in SPM compared to web access mining (Mabroukeh and Ezeife, 2010). Their *TreeMiner* algorithm uses a similar idea to hash tables, and stores at each node of the trie a matrix of links that are used to traverse and mine the trie. Although this can improve the time efficiency of the TreeMiner algorithm, the matrix of links grows linearly with the size of the data set and makes the algorithm costly in memory usage.

To improve over the high memory usage of vector and trie models, we begin by introducing a new binary trie model of the data set that stores a constant amount of information at each node. We prove that this information suffices for SPM, and develop a novel algorithm with significantly lower memory requirement than that of Rizvee et al. (2020). Furthermore, we prove that our trie model is always asymptotically smaller than a vector model, giving it a major advantage in SPM of large data sets. On the other hand, in data sets where only a few subsequences overlap, such as smaller data sets or ones with longer sequences, the memory overhead of modeling a sequence by a trie path may be higher than its vector representation.

To improve memory efficiency for such data sets, we build on our binary trie and develop a hybrid trie-vector model of the data set. The idea is to take advantage of the strengths of both trie and vector models and further increase the time and memory efficiency of the subsequent SPM task. In particular, we exploit the fact that data set sequences have high overlap in their initial entries (that is, their prefixes)—which can be effectively modeled via a trie—and low overlap in their latter entries (that is, their suffixes)—which are more efficiently

modeled via a vector. Our hybrid model is thus designed to find an effective transition from a trie model of prefixes to a vector model of suffixes that improves memory consumption.

To mine our hybrid data set structure, we combine our binary trie mining algorithm with a vector-based prefix-projection algorithm to develop a novel hybrid mining algorithm. We experimentally show that our hybrid algorithm outperforms both trie and vector models in time and memory usage. In particular, our hybrid algorithm can model and mine orders of magnitude larger data sets that are out of reach for any other SPM algorithm. Although primarily designed to handle large data sets, our algorithm improves time and memory usage when applied to small to medium-sized data sets, showcasing its potential for larger data sets in time efficiency.

The rest of the paper is organized as follows. We begin by discussing the preliminaries of SPM in §2, including vector and trie models of the literature. We then develop our novel binary trie and hybrid models and associated mining algorithms in §3. Numerical results on real-life large-size data sets are given in §4, and the paper is concluded in §5.

2. Preliminaries

This section provides preliminaries on SPM, starting with the definition of vector models of sequential data sets and followed by the definition of trie models. Throughout the paper, and for a thorough space complexity analysis, we consider that a vector of integers uses a constant $c^{\mathcal{V}}$ memory overhead, and that an integer j has $\mathcal{O}(\log(|j|))$ space complexity (Papadimitriou and Steiglitz, 1998). We also discuss the relaxation of the later assumption for *reasonably-sized* integers that can be stored in memory using constant overhead.

2.1 Vector Models of the Data Set for SPM

Let $\mathbb{E} = \{e_1, \ldots, e_{|\mathbb{E}|}\}$ be a finite set of literals, representing possible *events* or items within an application of interest. An *itemset* $\mathbb{I} = \{e_1, \ldots, e_{|\mathbb{I}|}\}$ is a set of events such that $\mathbb{I} \subseteq \mathbb{E}$. Although events of an itemset can be of any order, they are generally assumed to satisfy a monotone property and ordered accordingly (Han et al., 2001). A *sequence* $\mathcal{S}_i = \left\langle \mathbb{I}_i^1, \ldots, \mathbb{I}_i^{L_i} \right\rangle$ is an ordered list of L_i itemsets, with size $|\mathcal{S}_i| = \sum_{j=1}^{L_i} |\mathbb{I}_i^j|$. An event $e \in \mathbb{E}$ can occur at most once in an itemset, but the same itemset can occur multiple times in a sequence. Sequences may be equivalently represented in the event space $\mathcal{S}_i = \left\langle \bar{e}_1, \ldots, e_j, \bar{e}_{j+1}, \ldots, e_{|\mathcal{S}_i|} \right\rangle$, with the first event of any itemset indicated by an accented event $\bar{e}$. This notation is adopted throughout this paper.

A *sequential data set* $\mathcal{SD}$ is a list of N sequences $\mathcal{SD} = \langle \mathcal{S}_1, \ldots, \mathcal{S}_N \rangle$, with the size of its largest sequence denoted by $M = \max_{i \in \{1, \ldots, N\}} |\mathcal{S}_i|$. Example 1 describes a small instance of a sequential data set given in Table 1, which we use as a running example.

Example 1 *Table 1 gives a vector model of a sequential data set that includes $N = 20$ sequences $\mathcal{S}_1, \ldots, \mathcal{S}_{20}$ with events $\mathbb{E} = \{a, b, c\}$, and maximum sequence length $M = 6$. Sequences are given in itemset form, but can be equivalently represented in their event space. For instance, sequence $\mathcal{S}_1$ is a list of (ordered) itemsets $\langle \{a, b\}, \{a\}, \{b\} \rangle$ that can be equivalently represented as $\langle \bar{a}, b, \bar{a}, \bar{b} \rangle$.*

Table 1: Vector model of a sequential data set.

$\mathcal{S}_1$	$\langle\{a,b\},\{a\},\{b\}\rangle$	$\mathcal{S}_{11}$	$\langle\{a,c\},\{a,c\}\rangle$
$\mathcal{S}_2$	$\langle\{a,b,c\},\{c\}\rangle$	$\mathcal{S}_{12}$	$\langle\{a\}\rangle$
$\mathcal{S}_3$	$\langle\{a\},\{b\}\rangle$	$\mathcal{S}_{13}$	$\langle\{a,b\},\{a\},\{b,c\}\rangle$
$\mathcal{S}_4$	$\langle\{a,c\},\{a\}\rangle$	$\mathcal{S}_{14}$	$\langle\{a,c\},\{a,c\},\{a,c\}\rangle$
$\mathcal{S}_5$	$\langle\{a,b,c\}\rangle$	$\mathcal{S}_{15}$	$\langle\{a,c\},\{a\}\rangle$
$\mathcal{S}_6$	$\langle\{a,b\}\rangle$	$\mathcal{S}_{16}$	$\langle\{a,b\},\{a\},\{b\}\rangle$
$\mathcal{S}_7$	$\langle\{a,b,c\}\rangle$	$\mathcal{S}_{17}$	$\langle\{a,b\},\{a\},\{b\},\{a,c\}\rangle$
$\mathcal{S}_8$	$\langle\{a,b,c\},\{c\}\rangle$	$\mathcal{S}_{18}$	$\langle\{a,c\},\{a,c\},\{b,c\}\rangle$
$\mathcal{S}_9$	$\langle\{a,c\},\{a\}\rangle$	$\mathcal{S}_{19}$	$\langle\{a,b,c\},\{c\}\rangle$
$\mathcal{S}_{10}$	$\langle\{a,b\}\rangle$	$\mathcal{S}_{20}$	$\langle\{a,b\},\{a\}\rangle$

A sequence $\mathcal{S}_{i'}$ is said to be a *subsequence* of another sequence $\mathcal{S}_i$, denote $\mathcal{S}_{i'} \sqsubseteq \mathcal{S}_i$, if there exist integers $k_1 < \cdots < k_{L_{i'}}$ such that $\mathbb{I}_{i'}^j \subseteq \mathbb{I}_i^{k_j}$ for all $j = 1, \ldots, L_{i'}$. A *prefix* $\check{\mathcal{S}}_i$ is a contiguous subsequence of $\mathcal{S}_i$, with $k_1 = 1$ and $k_{j+1} = k_j + 1$ for all $j = 1, \ldots, |\check{\mathcal{S}}_i| - 1$. Similarly, a *postfix* $\hat{\mathcal{S}}_i$ is a contiguous subsequence of $\mathcal{S}_i$, with $k_1 = |\mathcal{S}_i| - |\hat{\mathcal{S}}_i| + 1$ and $k_{j+1} = k_j + 1$ for all $j = 1, \ldots, |\hat{\mathcal{S}}_i| - 1$.

The SPM task involves finding the set of *frequent patterns* within a data set $\mathcal{SD}$. A pattern $\mathcal{P}$ is a subsequence that satisfies $\mathcal{P} \sqsubseteq \mathcal{S}_i$ for at least one sequence $\mathcal{S}_i \in \mathcal{SD}$. The *support* $supp(\mathcal{P}) \in \mathbb{Z}_{>0}$ of a pattern $\mathcal{P}$ is the number of distinct sequences in $\mathcal{SD}$ for which $\mathcal{P} \sqsubseteq \mathcal{S}_i$. A pattern is considered frequent if $supp(\mathcal{P}) \geq \theta \times N$, where $0 < \theta \leq 1$ is a user-defined *minimum support threshold*.

Frequent patterns are generally found iteratively, where at each iteration a frequent pattern $\mathcal{P}$ is extended by a single event $e \in \mathbb{E}$, to $\langle\mathcal{P}, e\rangle$, and checked to satisfy $supp(\langle\mathcal{P}, e\rangle) \geq \theta \times N$. Pattern extensions are classified into an *itemset extension* or a *sequence extension*. In an itemset extension, pattern $\mathcal{P}$ is extended to $\langle\mathcal{P}, e\rangle$, extending its last itemset $\mathbb{I}^{|\mathcal{P}|}$ by a single event. In a sequence extension, a pattern is extended to $\langle\mathcal{P}, \bar{e}\rangle$, extending $\mathcal{P}$ by a new itemset $\mathbb{I} = \{\bar{e}\}$. Example 2 demonstrates a frequent pattern in our running example.

Example 2 *Given a support threshold of $\theta = 0.2$, an example frequent pattern in Table 1 is $\mathcal{P} = \langle\{a,b\}\rangle$. The pattern has support $supp(\mathcal{P}) = 12 \geq 4$, as it is a subsequence of sequences $\mathcal{S}_1, \mathcal{S}_2, \mathcal{S}_5, \mathcal{S}_6 - \mathcal{S}_8, \mathcal{S}_{10}, \mathcal{S}_{13}, \mathcal{S}_{16}, \mathcal{S}_{17}, \mathcal{S}_{19}, \mathcal{S}_{20}$ with $k_1 = 1, k_2 = 2$ for all sequences. An example sequence extension of $\mathcal{P}$ is $\mathcal{P}' = \langle\{a,b\},\{c\}\rangle$ with support $supp(\mathcal{P}') = 5$ (sequences $\mathcal{S}_2, \mathcal{S}_8, \mathcal{S}_{13}, \mathcal{S}_{17}, \mathcal{S}_{19}$). An example itemset extension is $\mathcal{P}'' = \langle\{a,b,c\}\rangle$ with support $supp(\mathcal{P}'') = 5$ (sequences $\mathcal{S}_2, \mathcal{S}_5, \mathcal{S}_7, \mathcal{S}_8, \mathcal{S}_{19}$).*

The literature generally models sequential data sets via vectors, where each sequence $\mathcal{S}_i \in \mathcal{SD}$ is stored in memory using a single vector. Current state-of-the-art SPM algorithms that use vector models store their entire representation of the data set in memory, which can be costly in terms of memory usage. In particular, Lemma 1 gives the worst-case space complexity of a vector-based SPM algorithm.

Lemma 1 *The worst-case space complexity of a vector-based SPM algorithm is $\mathcal{O}\left(NM \log(N)\right)$, and $\mathcal{O}\left(NM\right)$ for reasonably-sized integers N.*

Proof A sequential data set $\mathcal{SD}$ uses one vector per sequence $\mathcal{S}_i \in \mathcal{SD}$ and stores an integer $j \leq |\mathbb{E}|$ per event $e \in \mathcal{S}_i$. Assuming that $|\mathbb{E}| \leq N$ and $M \leq N$ (which generally hold in practice), the worst-case space complexity of a vector model is $\mathcal{O}\left(Nc^{\mathcal{V}} + NM \log(|\mathbb{E}|)\right) = \mathcal{O}\left(NM \log(|\mathbb{E}|)\right)$.

The most memory-efficient and basic SPM algorithm involves pseudo-projection (Han et al., 2001). Pseudo-projection stores a sequence ID-integer position pair (i, j) (with $i \leq N$ and $j \leq M$) for at most all $\mathcal{S}_i \in \mathcal{SD}$ and positions M. The worst-case space complexity of a vector-based SPM algorithm is thus $\mathcal{O}(NM \log(M) + NM \log(N)) = \mathcal{O}(NM \log(N))$.

Putting the two together, the worst-case space complexity of vector-based SPM algorithms is $\mathcal{O}(NM \log(|\mathbb{E}|) + NM \log(N)) = \mathcal{O}(NM \log(N))$. Reasonably sized integers N consume constant memory, and reduce the complexity to $\mathcal{O}(NM)$ ∎

As shown in Lemma 1, the memory consumption of vector models and algorithms grows linearly with N—for reasonably sized values of N. In practice, this can be highly memory-consuming, for example, for the large-size data sets of Villalobos and Ho (2022); Criteo AI Lab (2014); Consortium et al. (2015) that include billions to trillions of sequences. For such large data sets, vector-based SPM algorithms cannot fit their data set representation into system memory, which consequently prevents them from performing their mining task. To improve on this memory deficiency, we next examine trie models of the data set, which have the potential to conserve memory by representing multiple sequences of the sequential data set using a single path.

2.2 Trie Models of the Data Set for SPM

For a sequential data set $\mathcal{SD}$, let $\mathcal{T} := (\mathcal{N}, \mathcal{A}, \mathcal{L})$ be its *labeled* trie model with node set $\mathcal{N}$, arc set $\mathcal{A}$, and a set of labels $\mathcal{L}$. The node set $\mathcal{N}$ can be partitioned into $M + 1$ subsets $\mathcal{N}_0, \ldots, \mathcal{N}_M$, referred to as layers. Layer $\mathcal{N}_0 := \{\mathbf{r}\}$ is a singleton containing auxiliary root node $\mathbf{r}$, and the remaining $j = 1, \ldots, M$ layers model the events of sequences $\mathcal{S}_i \in \mathcal{SD}$. Accordingly, each node $n \in \mathcal{N} \setminus \{\mathbf{r}\}$ is associated with an *event label* $e_n \in \mathcal{L}$, which denotes the event literal $e \in \mathbb{E}$ modeled by node n; and an *itemset label* $\mathbb{I}_n$, which denotes the position j of itemset $\mathbb{I}_i^j$ in the sequence $\mathcal{S}_i$ *modeled* by path $\mathbf{P}$.

Using event and itemset labels e_n, a trie path $\mathbf{P} = (\mathbf{r}, n_1, \ldots, n_{|\mathbf{P}|})$ models the sequence $\mathcal{S}_i = \langle e_{n_1}, \ldots, e_{n_{|\mathbf{P}|}} \rangle$ belonging to itemsets $\langle \mathbb{I}_{n_1}, \ldots, \mathbb{I}_{n_{|\mathbf{P}|}} \rangle$. For notation convenience, we specify the sequence $\mathcal{S}_i$ modeled by a trie path $\mathbf{P}$ using a *transformation function* $\mathbf{S}(\mathbf{P}) = \mathcal{S}_i$.

The main advantage of trie models is their ability to model all overlapping prefixes $\check{\mathcal{S}}_i$ of sequences in $\mathcal{SD}$ using only a single path $\mathbf{P} : \mathbf{S}(\mathbf{P}) = \check{\mathcal{S}}_i$. Accordingly, nodes $n \in \mathcal{N} \setminus \{\mathbf{r}\}$ are associated with a positive integer *frequency label* $f_n \in \mathcal{L}$ that denotes the number of prefixes $\check{\mathcal{S}}_i$ of sequences $\mathcal{S}_i \in \mathcal{SD}$ modeled by path $\mathbf{P} = (\mathbf{r}, \ldots, n)$. Example 3 illustrates the general trie model of the data set in our running example.

Example 3 *Figure 1 depicts the trie model of the data set given in Table 1. Event labels are displayed inside each node, frequency labels are given above each node, and node layers are given above each column of nodes. The trie models the 20 sequences of the sequential data using 6 maximal paths and a total of $|\mathcal{N}| = 17$ nodes. Note that a vector model uses 20 vectors and a total of $\sum_{\mathcal{S}_i \in \mathcal{SD}} |\mathcal{S}_i| = 72$ units of memory to store the same data set. This is considerably less memory efficient even for our small running example. Although the trie*
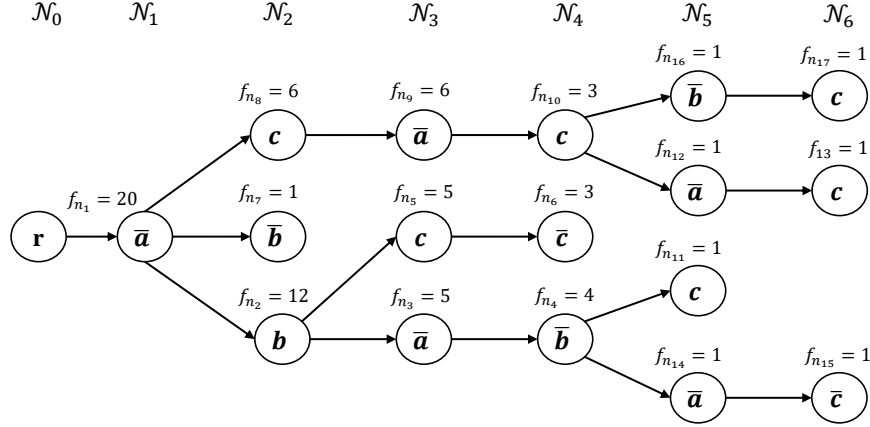
Figure 1: General trie model $\mathcal{T}$ of the data set in Table 1. Although the general trie structure correctly models all sequences of the data set and their frequencies, its labels are insufficient for SPM.

correctly models all sequences of the data set and their frequencies, its labels are insufficient for SPM.

Forgoing the differences between web access mining and SPM, the trie models of Yang et al. (2007); Pei et al. (2000); Lu and Ezeife (2003); Rizvee et al. (2020) are identical in their node and arc set $(\mathcal{N}, \mathcal{A})$. In particular, the contribution of the trie models in the literature is the additional labels in $\mathcal{L}$ that are required to perform web access mining or SPM (see, for example, Pei et al. (2000); Lu and Ezeife (2003); Yang et al. (2007); Rizvee et al. (2020)). These label sets are critical in the mining process, and in the case of web access mining, do not generalize to SPM. This is due to the inherent difference between web access mining and SPM, where web access algorithms cannot distinguish between itemset and sequence extensions (Mabroukeh and Ezeife, 2010).

In order to model and mine data sets in SPM, Rizvee et al. (2020) expand the label set $\mathcal{L}$ of each node $n \in \mathcal{N}$ by two additional labels. The first "next-links" label is a matrix of node pointers, where each row corresponds to an event $e \in \mathbb{E}$, and each column points to the first node $n' : e_{n'} = e$ of each path $(n, \ldots, n')$ spanning from node n. The second "parent-info" label is a bitset that determines which events $e \in \mathbb{E}$ fall into the same itemset as event e_n on the path $(\mathbf{r}, \ldots, n)$. These labels are used in a tailored mining algorithm, TreeMiner, to mine all patterns. Figure 2 shows the trie model of TreeMiner for our running example.

TreeMiner traverses its trie model using the next-link matrices and mines patterns using the information stored in the parent-info and itemset labels. Although next-links can enable faster traversal of the trie and SPM is possible using the parent-info and itemset labels, storing them at each node of the trie model is costly in memory usage. In particular, Lemma 2 gives the worst-case space complexity of the TreeMiner algorithm.

Lemma 2 *The worst-case space complexity of TreeMiner is $\mathcal{O}\left(\min\{N, |\mathbb{E}|^M\}M|\mathbb{E}|^2\log(N)\right)$, and $\mathcal{O}\left(\min\{N, |\mathbb{E}|^M\}M|\mathbb{E}|^2\right)$ for a reasonably-sized N.*

Proof A node $n \in \mathcal{T}$ of the trie model of TreeMiner stores integer values f_n, e_n, $\mathbb{I}_n$, which are all bounded by $\mathcal{O}(\log(N))$, a next-link matrix bounded by $\mathcal{O}((|\mathbb{E}| + 1)c^{\mathcal{V}} +$
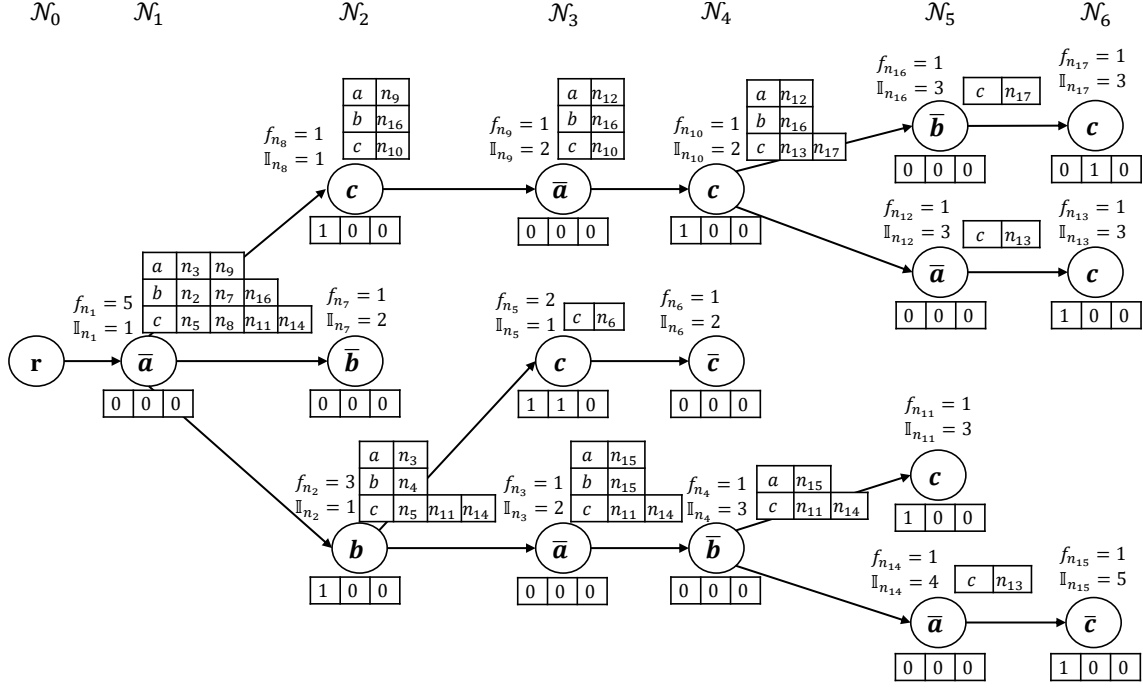
Figure 2: Trie model of TreeMiner for the data set of Table 1. The trie is updated with itemset labels $\mathbb{I}$, next-links displayed as a matrix adjacent to each node, and the parent-info bitset displayed as a vector under each node. TreeMiner mines the trie by traversing it using next-links and determining valid extensions using parent-info and itemset labels.

$\mathbb{E}|^2 \log(N)))$, a parent-info bitset bounded by $\mathcal{O}(c^{\mathcal{V}} + |\mathbb{E}|)$, and a children vector bounded by $\mathcal{O}\left(c^{\mathcal{V}} + |\mathbb{E}| \log(N)\right)$. The space complexity of a TreeMiner node is thus $\mathcal{O}\left(|\mathbb{E}|^2 \log(N)\right)$.

By definition, a node n of a trie model has at most $2|\mathbb{E}|$ children, an event $e \in \mathbb{E}$ representing an itemset extension and an event $\bar{e} \in \mathbb{E}$ representing a sequence extension. The maximum number of children for a trie layer $\mathcal{N}_j$ is thus $(2|\mathbb{E}|)^{|\mathcal{N}_j|}$. As $|\mathcal{N}_0| = 1$, the maximum number of nodes for any layer is $(2|\mathbb{E}|)^M$. On the other hand, we can have at most N nodes at any layer of a trie model of a sequential data set $\mathcal{SD}$, where each sequence $\mathcal{S}_i \in \mathcal{SD}$ is modeled by exactly one path. The maximum number of nodes in a trie model is thus $\mathcal{O}\left(\min\{N, |\mathbb{E}|^M\}M\right)$. For TreeMiner, this gives the overall worst-case memory complexity of its trie model as $\mathcal{O}\left(\min\{N, |\mathbb{E}|^M\}M|\mathbb{E}|^2 \log(N)\right)$.

For its mining algorithm, TreeMiner stores one positional integer $j \leq M$ for at most every node in its trie, bounding its worst-case space complexity by $\mathcal{O}\left(\min\{N, |\mathbb{E}|^M\}M \log(N)\right)$. Putting the two together, gives the overall space complexity of TreeMiner as $\mathcal{O}\left(\min\{N, |\mathbb{E}|^M\}M|\mathbb{E}|^2 \log(N)\right)$. Reasonably sized integers N consume constant memory and reduce the complexity to $\mathcal{O}\left(\min\{N, |\mathbb{E}|^M\}M|\mathbb{E}|^2\right)$. ∎

The memory efficiency of TreeMiner is highly dependent on the number of events $|\mathbb{E}|$ and maximum sequence length M. Consequently, TreeMiner may use more memory than a vector model when either value is high, and the data set does not contain many overlapping

sequences. We indeed observe this in a number of data sets in our numerical results. For a more memory-efficient SPM algorithm, we propose two novel approaches in the following section.

3. Binary and Hybrid Tries for Memory Efficient SPM

We introduce our binary trie $\mathcal{BT}$ in §3.1, and its corresponding mining algorithm in §3.2. We build on $\mathcal{BT}$ to develop a hybrid trie $\mathcal{HT}$ and its corresponding mining algorithm in §3.3.

3.1 Binary Tries

A binary trie $\mathcal{BT}$ is a doubly chained implementation of a trie $\mathcal{T}$ (Sussenguth Jr, 1963), with a novel addition to its set of labels $\mathcal{L}$. The binary structure of the trie is intended to reduce memory consumption in practice by avoiding the use of vectors to store a node's children. Instead, a node $n \in \mathcal{BT}$ is associated with at most one child node $chl(n)$ and one sibling node $sib(n)$. A child node $chl(n)$ models the first child of node n, with the remaining children modeled as contiguous siblings of $chl(n)$.

The main contribution of $\mathcal{BT}$ is its updated set of labels which consumes a constant amount of memory. In addition to event, frequency, and itemset labels of a general trie model, $\mathcal{BT}$ stores at each node $n \in \mathcal{N}$ an additional *ancestral label* $a_n \in \mathcal{L}$, given by Definition 3.

Definition 3 *The ancestor label a_n of a node $n \in \mathcal{BT}$ is defined as:*

$$a_n = \begin{cases} n' & \text{if } \exists n' : (n', \ldots, n) \in \mathcal{BT}, e_{n'} = e_n, \text{ and} \\ & e_{n''} \neq e_n \forall n'' \in (n', \ldots, n) : n'' \neq n, n'' \neq n', \\ \mathbf{r} & \text{otherwise.} \end{cases}$$

Intuitively, the ancestor label of a node n tracks the first occurrence of event e_n prior to node n, on the path $(\mathbf{r}, \ldots, n) \in \mathcal{BT}$. As we later show, ancestor labels are sufficient to effectively mine all patterns from a trie model, and can be efficiently generated during its construction. Example 4 illustrates the binary-trie model for our running example.

Example 4 *Consider the binary trie model of our running example given in Figure 3. The nodes n_2, n_7, n_8 are all siblings and model the children of their parent node n_1. Similarly, nodes n_3 and n_5 are siblings with parent node n_2. The ancestor of node n_7 is $a_{n_7} = \mathbf{r}$ as there is no node $n' : e_{n'} = e_{n_7}$ on the path $(\mathbf{r}, n_1, n_7)$. The ancestor of node n_{14} is $a_{n_{14}} = n_3$, as node n_3 is the first prior node to n_{14} on the path $(\mathbf{r}, n_1, n_2, n_3, n_4, n_{14})$ with the same item $e_{n_{14}} = e_{n_3}$.*

Construction of $\mathcal{BT}$ involves two scans of $\mathcal{SD}$. In the first scan, we follow the works of Pei et al. (2000) and Lu and Ezeife (2003) and perform support-based filtering on the data set. Support-based filtering involves removing any infrequent event $e \in \mathbb{E} : supp(e) < \theta \times N$ from $\mathcal{SD}$. Such infrequent events cannot be part of any frequent pattern due to the antimonotone property of $supp(\mathcal{P})$, and thus can be removed without affecting the generation of frequent patterns. In the next scan, $\mathcal{BT}$ is constructed by iterating over the sequences $\mathcal{S}_i \in \mathcal{SD}$. In each iteration, a sequence $\mathcal{S}_i$ is modeled by increasing the frequency values f_n for any node $n \in \mathbf{P} : \mathbf{S}(\mathbf{P}) = \mathcal{S}_i$ constructed in previous iterations, or by creating a new node if no such node exists in $\mathcal{BT}$. The complete procedure is given in Algorithm 1.
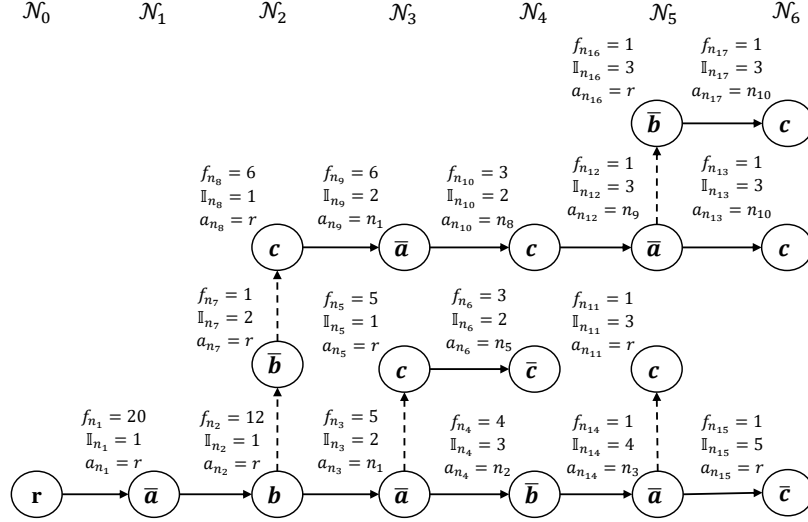
Figure 3: The $\mathcal{BT}$ model of the data set in Table 1. Sibling nodes are connected by dashed arcs. The trie is updated with itemset labels $\mathbb{I}_n$ and ancestor labels a_n, which are sufficient for SPM.

3.2 The $\mathcal{BT}$Miner Algorithm

Given a data set $\mathcal{SD}$, the SPM task involves finding all patterns $\mathcal{P}$ such that $supp(\mathcal{P}) \geq \theta \times N$. In prefix-projection, the algorithm is initialized by frequent patterns containing a single event, that is, $\mathcal{P} = \langle \bar{e} \rangle : e \in \mathbb{E}, supp(e) \geq \theta \times N$. At each subsequent iteration, a pattern $\mathcal{P}$ is extended by a single event $e \in \mathbb{E}$ to $\langle \mathcal{P}, e \rangle$, and checked to satisfy $supp(\langle \mathcal{P}, e \rangle) \geq \theta \times N$.

To mine a $\mathcal{BT}$ model of the data set, we first define $\mathcal{N}(\mathcal{P})$ as the set of terminal nodes $\{n\}$ of all minimal paths $\mathbf{P} = (\mathbf{r}, \dots, n) \in \mathcal{BT} : \mathcal{P} \sqsubseteq \mathbf{S}(\mathbf{P})$. Paths $\mathbf{P}$ are minimal in that $\forall n' \in (\mathbf{r}, \dots, n) : n' \neq n$, we have $\mathcal{P} \not\sqsubseteq \mathbf{S}((\mathbf{r}, \dots, n'))$. Proposition 4 proves that node sets $\mathcal{N}(e)$ model all minimum-sized prefixes $\check{\mathcal{S}}_i \sqsubseteq \mathcal{S}_i : \mathcal{P} \sqsubseteq \check{\mathcal{S}}_i$ for all $\mathcal{S}_i \in \mathcal{SD}$.

Proposition 4 *Node set $\mathcal{N}(\mathcal{P})$, as defined above, models all minimum-sized prefixes $\check{\mathcal{S}}_i \sqsubseteq \mathcal{S}_i : \mathcal{P} \sqsubseteq \check{\mathcal{S}}_i$ for all $\mathcal{S}_i \in \mathcal{SD}$.*

Proof The proof follows from the minimal property of paths $\mathbf{P} = (\mathbf{r}, \dots, n) : n \in \mathcal{N}(\mathcal{P})$. Assume by contradiction that the prefix $\check{\mathcal{S}}_i : \mathcal{P} \sqsubseteq \check{\mathcal{S}}_i$ modeled by a minimal path $\mathbf{P} : \mathbf{S}(\mathbf{P}) = \check{\mathcal{S}}_i$ is not of minimum size. Then there must exist another prefix $\check{\mathcal{S}}_i' : \check{\mathcal{S}}_i' \sqsubset \check{\mathcal{S}}_i, \mathcal{P} \sqsubseteq \check{\mathcal{S}}_i'$. Let $\mathbf{P}' = (\mathbf{r}, \dots, n')$ be the trie path that models prefix $\check{\mathcal{S}}_i'$. As trie paths are unique and $\check{\mathcal{S}}_i' \sqsubset \check{\mathcal{S}}_i$, we have $\mathbf{P} = \langle \mathbf{P}', \dots, n \rangle$, a contradiction to the minimality of path $\mathbf{P}$. As node set $\mathcal{N}(\mathcal{P})$ contains all nodes $n : \mathcal{P} \sqsubseteq \mathbf{S}((\mathbf{r}, \dots, n))$, then it models all prefixes $\check{\mathcal{S}}_i \sqsubseteq \mathcal{S}_i : \mathcal{P} \sqsubseteq \check{\mathcal{S}}_i$ for all $\mathcal{S}_i \in \mathcal{SD}$. ∎

Our mining algorithm, $\mathcal{BT}$Miner, initializes with the set of pattern-node-set pairs $(\mathcal{P} = \{e\}, \mathcal{N}(\mathcal{P}))$ for all events $e \in \mathbb{E}$. This can be done effectively by tracking the first occurrence of events $e \in \mathbb{E}$ of a sequence $\mathcal{S}_i \in \mathcal{SD}$ during the construction of $\mathcal{BT}$. In the next steps, $\mathcal{BT}$Miner iteratively takes a pattern-node-set pair $(\mathcal{P} = \{e\}, \mathcal{N}(\mathcal{P}))$, and attempts to construct sets $\mathcal{N}(\langle \mathcal{P}, e \rangle)$ by following Proposition 5

Algorithm 1 Construction of $\mathcal{BT}$.

1: Filter the data set and remove all events $e \in \mathbb{E} : supp(e) < \theta \times N$.
2: Initialize $\mathcal{BT}$ with $\mathcal{N} := \{\mathbf{r}\}$, $\mathcal{A} := \emptyset$
3: Let $n = \mathbf{r}$
4: **for** each $\mathcal{S}_i \in \mathcal{SD}$ **do**
5:     Let $\text{Anct}(e) = \mathbf{r}$ for all $e \in \mathbb{E}$.
6:     Let $\text{ItmSet} = 0$.
7:     **for** each $e_j \in \mathcal{S}_i$ **do**
8:         **if** node n has a child $n' = chl(n)$ **then**
9:             **while** $e_{n'} \neq e_j$ and n' has a sibling node $n'' = sib(n')$ **do**
10:                 Let $n = n'$,
11:                 Let $n' = n''$.
12:             **if** $e_{n'} = e_j$ **then**
13:                 Update $f_{n'} = f_{n'} + 1$, and set $n = n'$.
14:             **else**
15:                 Add $n' : e_{n'} = e_j, f_{n'} = 1, a_{n'} = \text{Anct}(e_j), \mathbb{I}_{n'} = \text{ItmSet}$ to $\mathcal{N}_j$.
16:                 Add sibling arc (n, n') to $\mathcal{A}$.
17:         **else**
18:             Add $n' : e_{n'} = e_j, f_{n'} = 1, a_{n'} = \text{Anct}(e_j), \mathbb{I}_{n'} = \text{ItmSet}$ to $\mathcal{N}_j$.
19:             Add child arc (n, n') to $\mathcal{A}$.
20:         **if** $e_j = \bar{e}_j$ **then**
21:             Update $\text{ItmSet} = \text{ItmSet} + 1$.
22:         Update $\text{Anct}(e_j) = n'$.
23: **return** $\mathcal{BT}$.

Proposition 5 *Given a node set $\mathcal{N}(\mathcal{P})$, node set $\mathcal{N}(\langle \mathcal{P}, e \rangle)$ is constructed by finding all minimal paths $(n, \ldots, n') \in \mathcal{BT} : e_{n'} = e$, for all nodes $n \in \mathcal{N}(\mathcal{P})$.*

Proof By Proposition 4, all paths $(\mathbf{r}, \ldots, n) : n \in \mathcal{N}(\mathcal{P})$ model all minimum-sized prefixes $\check{\mathcal{S}}_i \sqsubseteq \mathcal{S}_i : \mathcal{P} \sqsubseteq \check{\mathcal{S}}_i$ for all $\mathcal{S}_i \in \mathcal{SD}$. As paths $(n, \ldots, n')$ are minimal by definition and have $e_{n'} = e$, then paths $(\mathbf{r}, \ldots, n')$ are also minimal and satisfy $\langle \mathcal{P}, e \rangle \sqsubseteq \mathbf{S}\left((\mathbf{r}, \ldots, n') \right)$. Finding all such paths for all nodes $n \in \mathcal{N}(\mathcal{P})$ thus constructs $\mathcal{N}(\langle \mathcal{P}, e \rangle)$. $\blacksquare$

Finding minimal paths $(n, \ldots, n')$ in a trie is a chalenging task, with extant trie-based SPM algorithms requiring a rich set of information to be stored at their nodes. For example, TreeMiner requires traversing the tree using next-links and using parent info information to determine valid pattern extensions. For $\mathcal{BT}$Miner, the process involves a depth-first-search of the sub-trie rooted at node n and finding nodes n' according to Theorem 6.

Theorem 6 *Let n' be a node traversed on a path $\mathbf{P} = (n, \ldots, n')$ rooted at a node $n \in \mathcal{N}(\mathcal{P})$. Path $(n, \ldots, n')$ is minimal for the construction of set $\mathcal{N}(\langle \mathcal{P}, e_n \rangle)$ if and only if:*

- *For a sequence extension $\langle \mathcal{P}, \bar{e}_{n'} \rangle$ we have $\mathbb{I}_{n'} \neq \mathbb{I}_n$ and $\mathbb{I}_{a_{n'}} \leq \mathbb{I}_n$,*

- *For an itemset extension $\langle \mathcal{P}, e_{n'} \rangle$ we have*

    – $\mathbb{I}_{n'} = \mathbb{I}_n$, or

    – $\forall e_j \in \mathbb{I}^{|\mathcal{P}|}, \exists n'' \in \mathbf{P} : e_{n''} = e_j, \mathbb{I}_{n''} = \mathbb{I}_{n'}$ and

        * $\mathbb{I}_{a_{n'}} < \mathbb{I}_n$, or

        * $\forall n''' \in \mathbf{P} : e_{n'''} = e_{n'}$ we have $n''' \notin \mathcal{N}\left(\langle \mathcal{P}, e_{n'} \rangle\right)$.

Proof For a sequence extension $\langle \mathcal{P}, \bar{e}_{n'} \rangle$, assume by contradiction that $\mathbf{P} = (n, \ldots, n')$ is minimal, but $\mathbb{I}_{n'} = \mathbb{I}_n$ or $\mathbb{I}_{a_{n'}} > \mathbb{I}_n$. If $\mathbb{I}_{n'} = \mathbb{I}_n$ then node n' belongs to the same itemset as node n and cannot be used for a sequence extension, a contradiction. If $\mathbb{I}_{a_{n'}} > \mathbb{I}_n$, then we must have $\mathbf{P} = (n, \ldots, a_{n'}, \ldots, n')$, and consequently $\langle \mathcal{P}, \bar{e}_{n'} \rangle \sqsubseteq \mathbf{S}\left((n, \ldots, a_{n'})\right)$, contradicting the minimality of $\mathbf{P}$.

    For the converse, assume by contradiction that $\mathbb{I}_{n'} \neq \mathbb{I}_n$ and $\mathbb{I}_{a_{n'}} \leq \mathbb{I}_n$, but that path $\mathbf{P}$ is not minimal. Then path $\mathbf{P}$ must be of the form $\mathbf{P} = (n, \ldots, n'', \ldots, n')$, such that $\mathbf{P}' = (n, \ldots, n'') : \langle \mathcal{P}, \bar{e}_{n'} \rangle \sqsubseteq \mathbf{S}\left((n, \ldots, a_{n''})\right)$ is minimal. As n'' corresponds to a sequence extension $\langle \mathcal{P}, \bar{e}_{n'} \rangle$, we must have $e_{n''} = e_{n'}, \mathbb{I}_{n''} > \mathbb{I}_n$. This contradicts $\mathbb{I}_{a_{n'}} \leq \mathbb{I}_n$, by definition of ancestor label $a_{n'}$.

    For an itemset extension $\langle \mathcal{P}, e_{n'} \rangle$, assume by contradiction that $\mathbf{P} = (n, \ldots, n')$ is minimal, but,

1. $\mathbb{I}_{n'} \neq \mathbb{I}_n$ and

2. $\exists e_j \in \mathbb{I}^{|\mathcal{P}|} : \nexists n'' \in \mathbf{P} : e_{n''} = e_j, \mathbb{I}_{n''} = \mathbb{I}_{n'}$, or

3. $\forall e_j \in \mathbb{I}^{|\mathcal{P}|}, \exists n'' \in \mathbf{P} : e_{n''} = e_j, \mathbb{I}_{n''} = \mathbb{I}_{n'}$ and

    (a) $\mathbb{I}_{a_{n'}} \geq \mathbb{I}_n$, and

    (b) $\exists n''' \in \mathbf{P} : e_{n'''} = e_{n'}$ and $n''' \in \mathcal{N}\left(\langle \mathcal{P}, e_{n'} \rangle\right)$.

Consider conditions 1 and 2. Due to condition 1, node n' models an event within another itemset to the event modeled by node n. As path $\mathbf{P}$ is minimal, we must have $\mathbb{I}^{|\mathcal{P}|} \sqsubset \mathbb{I}^{\mathbb{I}_{n'}}$. This means $\forall e_j \in \mathbb{I}^{|\mathcal{P}|}, \exists n'' \in \mathbf{P} : e_{n''} = e_j, \mathbb{I}_{n''} = \mathbb{I}_{n'}$, a contradiction to condition 2. Consider conditions 1 and 3. Due to condition 3b, path $(n, \ldots, n'')$ is minimal, a contradiction to the minimality of path $\mathbf{P}$.

    For the converse, assume by contradiction that the statement holds, but path $\mathbf{P}$ is not minimal. Then there must exist a node $n''' \in \mathbf{P}, n''' \neq n'$ such that path $(n, \ldots, n'')$ is minimal for the construction of set $\mathcal{N}\left(\langle \mathcal{P}, e_{n'} \rangle\right)$. This contradicts $\mathbb{I}_{a_{n'}} < \mathbb{I}_n$ by the definition of ancestor label $a_{n'}$, and contradicts $n''' \notin \mathcal{N}\left(\langle \mathcal{P}, e_{n'} \rangle\right)$ otherwise. ∎

    At each iteration, $\mathcal{BT}$Miner takes a tuple $(\mathcal{P}, \mathcal{N}(\mathcal{P}))$ and extends $\mathcal{P}$ by searching the sub-tries rooted at nodes $n \in \mathcal{N}(\mathcal{P})$ and checking the conditions of Theorem 6. The process can be made more efficient by following a two-phase depth-first-search procedure:

Phase 1: Search the paths rooted at node n up until the first node $n' : \mathbb{I}_{n'} \neq \mathbb{I}_n$. Here, the conditions of Theorem 6 are automatically satisfied, with nodes $n'' \neq n'$ modeling an itemset extension and nodes n' modeling a sequence extension, as proved in Lemma 7.

Lemma 7 *Let $\mathbf{P} = (n, \ldots, n') \in \mathcal{BT}$ be a path rooted at a node $n \in \mathcal{N}(\mathcal{P})$ and terminating at the first node $n' : \mathbb{I}_{n'} \neq \mathbb{I}_n$. The node n' models a sequence extension $\langle \mathcal{P}, \bar{e}_{n'} \rangle$, and all nodes $n'' \in \mathbf{P} : n'' \neq n, n'' \neq n'$ model an itemset extension $\langle \mathcal{P}, e_{n''} \rangle$.*

Proof By the statement, all nodes $n'' \in \mathbf{P} : n'' \neq n'$ have $\mathbb{I}_{n''} = \mathbb{I}_n$ and thus belong to the same itemset. By definition, events within an itemset are unique and cannot be repeated. Therefore, we have $\mathbb{I}_{a_{n''}} < \mathbb{I}_n$, which by Theorem 6, means that all nodes n'' model an itemset extension $\langle \mathcal{P}, e_{n''} \rangle$.

Similarly, as node n' is the first node in $\mathbf{P}$ such that $\mathbb{I}_{n'} \neq \mathbb{I}_n$, its ancestor $a_{n'}$ must either occur prior to node n or within the same itemset as node n. This means $\mathbb{I}_{a_{n'}} \leq \mathbb{I}_n$, which by Theorem 6, means that node n' models a sequence extension $\langle \mathcal{P}, e_{n'} \rangle$. ∎

Phase 2: Search the paths rooted at nodes n' found in phase 1, and check the conditions of Theorem 6 to determine itemset and sequence extensions for any traversed node.

The complete $\mathcal{BT}$Miner algorithm for $\mathcal{BT}$ is given in Algorithm 2, proved to find all frequent patterns in Theorem 8, and exemplified in Example 5.

Theorem 8 *A pattern is frequent if and only if it is found by $\mathcal{BT}$Miner.*

Proof Assume that a pattern $\mathcal{P}$ is frequent, and thus has support $supp(\mathcal{P}) \geq \theta \times N$. By the definition of support values, there exists at least $\theta \times N$ sequences $\mathcal{S}_i \in \mathcal{SD}$ such that $\mathcal{P} \sqsubseteq \mathcal{S}_i$. By Proposition 4, node set $\mathcal{N}(\mathcal{P})$ models all minimum-sized prefixes $\check{\mathcal{S}}_i \sqsubseteq \mathcal{S}_i : \mathcal{P} \sqsubseteq \check{\mathcal{S}}_i$ for all $\mathcal{S}_i \in \mathcal{SD}$. By Proposition 5 and Theorem 6, all nodes belonging to set $\mathcal{N}(\mathcal{P})$ are found by $\mathcal{BT}$Miner. Therefore, $\mathcal{BT}$Miner finds all sequences $\mathcal{S}_i : \mathcal{P} \sqsubseteq \mathcal{S}_i$.

Assume that a pattern $\mathcal{P}$ is found by $\mathcal{BT}$Miner. This means that $\sum_{n \in \mathcal{N}(\mathcal{P})} f_n \geq \theta \times N$. As nodes $n \in \mathcal{N}(\mathcal{P})$ model a minimum-sized prefix $\check{\mathcal{S}}_i \sqsubseteq \mathcal{S}_i : \mathcal{P} \sqsubseteq \check{\mathcal{S}}_i$, then there must exist at least $\theta \times N$ sequences $\mathcal{S}_i \in \mathcal{SD}$ such that $\mathcal{P} \sqsubseteq \mathcal{S}_i$. Therefore, $supp(\mathcal{P}) \geq \theta \times N$, and pattern $\mathcal{P}$ is frequent. ∎

Example 5 *In the first iteration, $\mathcal{BT}$Miner initializes with pattern-node pairs $(\langle \bar{a} \rangle, \{n_1\}), (\langle \bar{b} \rangle, \{n_2, n_7, n_{16}\}), (\langle \bar{c} \rangle, \{n_5, n_8, n_{11}, n_{15}\})$. The mining algorithm takes a pattern-node pair, for example, $(\langle \bar{a} \rangle, \{n_1\})$, and in the first phase performs a depth-first-search of the subtrie rooted at node n_1 to find the first nodes $n' : \mathbb{I}_{n'} \neq \mathbb{I}_n$. This results in finding nodes n_3, n_6, n_7, n_9. All these nodes model a sequence extension according to Lemma 7, and give a pattern-node pair $(\langle \bar{a}, \bar{a} \rangle, \{n_3, n_9\}), (\langle \bar{a}, \bar{b} \rangle, \{n_7\}), (\langle \bar{a}, \bar{c} \rangle, \{n_6\})$. On the other hand, any node traversed on the paths $\mathbf{P} = (n_1, \ldots, n'), n' \in \{n_3, n_6, n_7, n_9\}$ models an itemset extension according to Lemma 7, which gives pattern-node pairs $(\langle \bar{a}, b \rangle, \{n_2\})(\langle \bar{a}, c \rangle, \{n_5, n_8\})$.*

In the second phase, search is initiated from nodes n_3, n_6, n_7, n_9, found in phase 1. Any pattern extension is determined by following Theorem 6. For example, Searching the sub-trie rooted at node n_9 first traverses node n_{10}. To check for a sequence extension, we have $\mathbb{I}_{n_{10}} \neq \mathbb{I}_{n_1}$ and $\mathbb{I}_{a_{n_{10}}} \leq \mathbb{I}_{n_1}$, which satisfies the condition of Theorem 6. The corresponding pattern-node pair is then updated to $(\langle \bar{a}, \bar{c} \rangle, \{n_6, n_{10}\})$.

To check for an itemset extension, we have $\mathbb{I}_{n_{10}} \neq \mathbb{I}_{n_1}$ which violates the first condition of Theorem 6, but $e_{n_9} = e_{n1}, \mathbb{I}_{n_9} = \mathbb{I}_{n_{10}}$ which satisfies the first part of the second condition. Checking the second parts of the second condition, we have $\mathbb{I}_{a_{n_{10}}} = \mathbb{I}_{n_1}$, and $n_8 \in \mathcal{N}(\langle \bar{a}, c \rangle)$ which violates both conditions. Therefore, node n_{10} does not model an itemset extension.

Algorithm 2 The $\mathcal{BT}$Miner algorithm.

1: Initialize pattern-node pairs $\{(\mathcal{P}, \mathcal{N}(\mathcal{P}))\}$ for all $\mathcal{P} = \langle \bar{e} \rangle$, $e \in \mathbb{E} : supp(e) \geq \theta \times N$.

2: **for** each pair $(\mathcal{P}, \mathcal{N}(\mathcal{P}))$ **do**

3:     Let $\mathcal{N}(\langle \mathcal{P}, e \rangle) = \emptyset$ for all $e \in \mathbb{E}$, and $\bar{j} = \max_{j' \in \{1, \ldots, |\mathcal{P}|-1\}} j' : \mathbb{I}_{j'} \neq \mathbb{I}_{j'+1}$.

4:     **for** each $n \in \mathcal{N}(\mathcal{P})$ **do**

5:         Initialize DFS queues $Q_{\mathcal{N}}^1 = \langle n \rangle$, $Q_{\mathcal{N}}^2 = \langle \rangle$, $Q_{\mathbb{I}} = \langle \rangle$.

6:         **while** $Q_{\mathcal{N}}^1$ is nonempty **do**

7:             Take n' from the front of queue $Q_{\mathcal{N}}$.

8:             **for** each child (or child-sibling) node n'' of node n' **do**

9:                 **if** $\mathbb{I}_{n'} = \mathbb{I}_{n''}$ **then**

10:                     Add node n'' to $\mathcal{N}(\langle \mathcal{P}, e_{n''} \rangle)$ as an itemset extension,

11:                     Add node n'' to the front of queue $Q_{\mathcal{N}}^1$.

12:                 **else**

13:                     Add node n'' to $\mathcal{N}(\langle \mathcal{P}, \bar{e}_{n''} \rangle)$ as a sequence extension,

14:                     Add node n'' to the front of queue $Q_{\mathcal{N}}^2$.

15:                     **if** $e_{n''} = e_{\bar{j}}$ **then**

16:                         Add 1 to the front of queue $Q_{\mathbb{I}}$.

17:                     **else**

18:                         Add 0 to the front of queue $Q_{\mathbb{I}}$.

19:         **while** $Q_{\mathcal{N}}^2$ is nonempty **do**

20:             Take n' from the front of queue $Q_{\mathcal{N}}^2$, and integer k from front of queue $Q_{\mathbb{I}}$.

21:             **for** each child (or child-sibling) node n'' of node n' **do**

22:                 **if** $\mathbb{I}_{n'} = \mathbb{I}_{n''}$ **then**

23:                     **if** $k = |\mathcal{P}| - j$ and conditions of Theorem 6 are satisfied **then**

24:                       Add node n'' to $\mathcal{N}(\langle \mathcal{P}, e_{n''} \rangle)$ as an itemset extension.

25:                   **if** Conditions of Theorem 6 are satisfied **then**

26:                       Add node n'' to $\mathcal{N}(\langle \mathcal{P}, \bar{e}_{n''} \rangle)$ as a sequence extension.

27:                   Add node n'' to the front of queue $Q_{\mathcal{N}}^2$.

28:                   **if** $k < |\mathcal{P}| - j$ and $e_{n''} = e_{\bar{j}+k}$ **then**

29:                     Add k + 1 to the front of queue $Q_{\mathbb{I}}$.

30:                   **else**

31:                     Add k to the front of queue $Q_{\mathbb{I}}$.

32:                 **else**

33:                   **if** conditions of Theorem 6 are satisfied **then**

34:                     Add node n'' to $\mathcal{N}(\langle \mathcal{P}, \bar{e}_n \rangle)$ as a sequence extension.

35:                   Add node n'' to the front of queue $Q_{\mathcal{N}}^2$.

36:                   **if** $e_{n''} = e_{\bar{j}}$ **then**

37:                     Add 1 to the front of queue $Q_{\mathbb{I}}$.

38:                   **else**

39:                     Add 0 to the front of queue $Q_{\mathbb{I}}$.

40:     **for** each $e \in \mathbb{E}$ **do**

41:         **if** $\sum_{n \in \mathcal{N}(\langle \mathcal{P}, e \rangle)} f_n \geq \theta \times N$ **then**

42:         Add $(\langle \mathcal{P}, e \rangle, \mathcal{N}(\langle \mathcal{P}, e \rangle))$ to the set of pattern-node pairs.

43: **return** Set of frequent patterns $\mathcal{P} : \sum_{n \in \mathcal{N}(\mathcal{P})} f_n \geq \theta \times N$.

The worst-case space complexity of $\mathcal{BT}$Miner is given by Theorem 9.

Theorem 9 *The worst-case space complexity of $\mathcal{BT}$Miner is $\mathcal{O}\left(\min\{N, |\mathbb{E}|^M\}M\log(N)\right)$, and $\mathcal{O}\left(\min\{N, |\mathbb{E}|^M\}M\right)$ for reasonably-sized integers N.*

Proof A node $n \in \mathcal{BT}$ stores integer values f_n, e_n, $\mathbb{I}_n$, a positional value a_n, and two positional values $chl(n)$, $sib(n)$. All values are bounded by $\mathcal{O}(\log(N))$.

As trie models of the data set are identical in their node set $\mathcal{N}$, and by Lemma 2, the number of nodes in a trie model is $\mathcal{O}\left(\min\{N, |\mathbb{E}|^M\}M\right)$. This gives the total worst-case memory complexity of $\mathcal{BT}$ as $\mathcal{O}\left(\min\{N, |\mathbb{E}|^M\}M\log(N)\right)$.

Similarly to TreeMiner, $\mathcal{BT}$ Miner stores a positional integer $j \leq M$ for at most every node $n \in \mathcal{BT}$, bounding its worst-case space complexity by $\mathcal{O}\left(\min\{N, |\mathbb{E}|^M\}M\log(N)\right)$. The total space complexity of $\mathcal{BT}$Miner is thus $\mathcal{O}\left(\min\{N, |\mathbb{E}|^M\}M\log(N)\right)$. Reasonably sized integers N consume constant memory and reduce the complexity to $\mathcal{O}\left(\min\{N, |\mathbb{E}|^M\}M\right)$. $\blacksquare$

Compared to TreeMiner, $\mathcal{BT}$Miner is at least $\mathcal{O}(|\mathbb{E}|^2)$ times more memory efficient and never larger. Similarly, $\mathcal{BT}$Miner is always asymptotically smaller than a vector-based SPM algorithm. In the best case for $\mathcal{BT}$Miner, we have $|\mathbb{E}| = 1$ and space complexity of $M\log(N)$, leading to $\mathcal{O}(N)$ times more efficient memory consumption than a vector model for reasonably-sized integers N. In the worst case, $\mathcal{BT}$ uses the same number of nodes as entries $\sum_{\mathcal{S}_i \in \mathcal{SD}} |\mathcal{S}_i|$. In practice, this leads to higher memory usage due to the memory overhead of label sets $\mathcal{L}_n$ stored at each node $n \in \mathcal{BT}$. We address this deficiency by developing a hybrid model in the next section.

3.3 Hybrid Tries and the $\mathcal{HT}$Miner Algorithm

Trie models are most memory-efficient when modeling sequences $\mathcal{S}_i \in \mathcal{SD}$ that highly overlap. Such sequences can be represented by fewer nodes $|\mathcal{N}|$, leading to a more compact model that compensates for the higher memory overhead $c^{\mathcal{T}}$ of each trie node. On the other hand, when many sequences $\mathcal{S}_i \in \mathcal{SD}$ do not overlap, the model is less compact and can lead to increased memory consumption in practice. For such data sets, vector models are a more memory-efficient approach.

Regardless of the overlap for sequences $\mathcal{S}_i \in \mathcal{SD}$, sequential data sets generally have a high overlap on their prefixes $\check{\mathcal{S}}_i : \check{\mathcal{S}}_i \sqsubseteq \mathcal{S}_i \in \mathcal{SD}$. This is due to the lower number of possible event combinations $|\mathbb{E}|^j \geq |\mathcal{N}_j|$ at any length $j \leq |\mathcal{S}_i|$, which is typically much lower than N for smaller values of j. For such prefixes, a trie model is more memory efficient. As j increases, so does the possible number of unique and non-overlapping sequences, leading to potentially less overlap on postfixes $\hat{\mathcal{S}}_i \sqsubseteq \mathcal{S}_i \in \mathcal{SD}$. For such postfixes, a vector model is more memory efficient in practice. We propose a hybrid model $\mathcal{HT}$ to take advantage of this trade-off.

A hybrid model $\mathcal{HT}$ is a $\mathcal{BT}$ representation of all prefixes $\check{\mathcal{S}}_i \sqsubseteq \mathcal{S}_i$ of length $|\check{\mathcal{S}}_i| = j^*$, that transitions into a vector model for the remaining postfixes $\hat{\mathcal{S}}_i : |\hat{\mathcal{S}}_i| = |\mathcal{S}_i| - j^*$. For the subsequent mining task, $\mathcal{HT}$ also stores at each node n of the transitioning layer $\mathcal{N}_{j^*}$, a vector of ancestor nodes $\vec{a} = \langle a_{e_1}, \ldots, a_{e_{|\vec{a}|}} \rangle$ such that $a_e \in \vec{a}$ if $e \in \mathbf{S}(\mathbf{r}, \ldots, n)$. For example, Figure 4 illustrates an $\mathcal{HT}$ model for our running example that transitions at length $j^* = 4$.

An important challenge of our hybrid model is determining the transition length j^* that provides the highest memory efficiency. This is data set dependent and determined based on
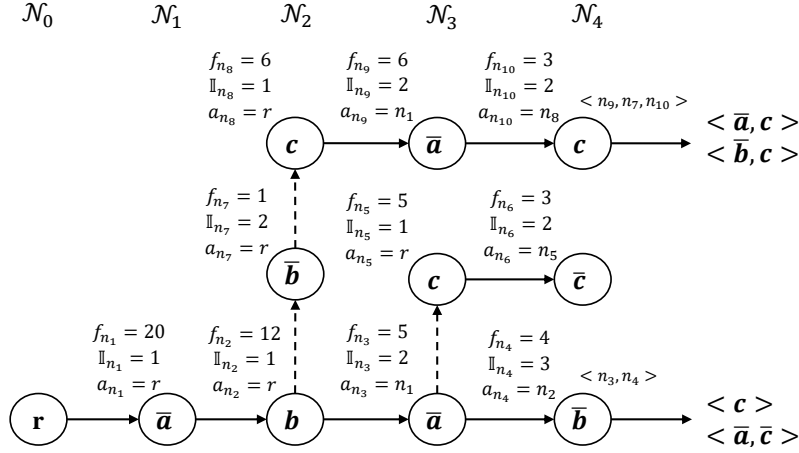
Figure 4: An $\mathcal{HT}$ model of the data set in Table 1. The model transitions from a trie to a vector representation at layer $\mathcal{N}_4$. Transitioning nodes n_4, n_{10} are associated to an ancestor vector that tracks the ancestors for all events $e \in \mathbb{E} : e \in \mathbf{S}(\mathbf{r}, \ldots, n_j), j \in \{4, 10\}$.

the trade-off between the compression that a $\mathcal{BT}$ model can provide for layers $\mathcal{N}_j, 0 < j \leq j^*$, versus the memory overhead introduced by the transition to a vector model for postfixes $\hat{\mathcal{S}}_i : |\hat{\mathcal{S}}_i| = M - j^*$. We choose j^* using Theorem 10, where N_j denotes the number of sequences $\mathcal{S}_i \in \mathcal{SD} : |\mathcal{S}_i| \geq j$, recorded during the preprocessing step of constructing $\mathcal{BT}$.

Theorem 10 *It is most memory-efficient for a hybrid model $\mathcal{HT}$ to transition from a $\mathcal{BT}$ representation to a vector representation at the layer $\mathcal{N}_{j^*}$, where*

$$j^* = \underset{j^* \in \{0,\ldots,M\}}{\arg\min} \ c^{\mathcal{T}} c^{int} \sum_{j=1}^{j^*} |\mathcal{N}_j| + \mathbb{1}(j^* < M) \left(c^{\mathcal{V}} + c^{int} \min\{|\mathbb{E}|, j^*\} \right) |\mathcal{N}_{j^*}| + c^{\mathcal{V}} N_{j^*+1} + c^{int} \sum_{j=j^*+1}^{M} N_j.$$

Proof For an $\mathcal{HT}$ transitioning at a layer $\mathcal{N}_{j'} \in \mathcal{HT} : j' \in \{0, \ldots, M\}$ involves a $\mathcal{BT}$ model of all prefixes $\check{\mathcal{S}}_i \sqsubseteq \mathcal{S}_i \in \mathcal{SD} : |\check{\mathcal{S}}_i| = j'$, and a vector model of postfixes $\hat{\mathcal{S}}_i \sqsubset \mathcal{S}_i : |\hat{\mathcal{S}}_i| = |\mathcal{S}_i| - j'$.

The $\mathcal{BT}$ model of prefixes $\check{\mathcal{S}}_i$ consumes $c^{\mathcal{T}} c^{\text{int}}$ memory overhead per nodes $n \in \mathcal{BT}$. If $j' < M$, $\mathcal{HT}$ also stores a vector of at most $\min\{|\mathbb{E}|, j^*\}$ ancestors per node $n \in \mathcal{N}_{j'}$, which has $c^{\mathcal{V}} + c^{\text{int}} \min\{|\mathbb{E}|, j'\}$ memory overhead. The total memory consumption of the $\mathcal{BT}$ model of $\mathcal{HT}$ is thus $c^{\mathcal{T}} c^{\text{int}} \sum_{j=1}^{j'} |\mathcal{N}_j| + \mathbb{1}(j^* < M) \left(c^{\mathcal{V}} + c^{\text{int}} \min\{|\mathbb{E}|, j'\} \right) |\mathcal{N}_{j'}|$.

The vector model of postfixes $\hat{\mathcal{S}}_i$ uses one vector per $N_{j'+1}$ sequences $\mathcal{S}_i \in \mathcal{SD} : |\mathcal{S}_i| \geq j'+1$, and c^{int} memory overhead per events $e \in \hat{\mathcal{S}}_i$. This gives the total memory consumption of the vector model of $\mathcal{HT}$ as $c^{\mathcal{V}} N_{j'+1} + c^{\text{int}} \sum_{j=j'+1}^{M} N_j$.

The most memory-efficient transition layer $\mathcal{N}_{j^*}$ is determined by the $\mathcal{HT}$ model with the lowest memory consumption, that is,

$$j^* = \underset{j^* \in \{0,\ldots,M\}}{\arg\min} \ c^{\mathcal{T}} c^{\text{int}} \sum_{j=1}^{j^*} |\mathcal{N}_j| + \mathbb{1}(j^* < M) \left(c^{\mathcal{V}} + c^{\text{int}} \min\{|\mathbb{E}|, j^*\} \right) |\mathcal{N}_{j^*}| + c^{\mathcal{V}} N_{j^*+1} + c^{\text{int}} \sum_{j=j^*+1}^{M} N_j.$$
■

Theorem 10 is based on the size of each layer $\mathcal{N}_j \in \mathcal{BT}$. Unfortunately, these values are unknown prior to the construction of the $\mathcal{BT}$ model, but following Theorem 9, can be

---

Algorithm 3 Construction of $\mathcal{HT}$.

1: Filter the data set and remove all events $e \in \mathbb{E} : supp(e) < \theta \times N$, and record N_j, $|\mathbb{E}|_j$ for all $j \in \{0, \ldots, M\}$.
2: Determine j^* according to Theorem 10 and values N_j and $|\mathbb{E}|_j$.
3: **for** each $\mathcal{S}_i \in \mathcal{SD}$ **do**
4:     Construct a $\mathcal{BT}$ model for prefix $\check{\mathcal{S}}_i \sqsubseteq \mathcal{S}_i : |\check{\mathcal{S}}_i| = j^*$.
5:     Add child arc (n, n') to $\mathcal{A}$, where n is the last node constructed by $\mathcal{BT}$ and n' is contains the vector that models $\hat{\mathcal{S}}_i : \mathcal{S}_i = \langle \check{\mathcal{S}}_i, \hat{\mathcal{S}}_i \rangle$.
6:     Store ancestor vector constructed at $\mathcal{BT}$ at n'.
7: **return** $\mathcal{HT}$.

---

approximated by the upper bound $|\mathcal{N}_j| \leq \min\{\prod_{j'=0}^{j} |\mathbb{E}|_{j'}, N_j\}$. Here, $|\mathbb{E}|_j$ is the number of unique events at the length j of sequences $\mathcal{S}_i \in \mathcal{SD}$, that are recorded during preprocessing. Length j^* is then found by iterating over $j \in \{0, \ldots, M\}$ and calculating the equation of Theorem 10. In the extreme case a trie representation does not provide any memory saving potential, we have $j^* = 0$ and $\mathcal{HT} = \mathcal{V}$. On the other hand, if a trie model of the entire data set is predicted to be more efficient, we have $j^* = M$ and $\mathcal{HT} = \mathcal{BT}$. Algorithm 3 gives the full construction procedure for $\mathcal{HT}$, and Example 6 demonstrates the procedure of determining j^* for our running example.

Example 6 *We have $N_j = \langle 20, 19, 16, 10, 4, 3 \rangle$, and $|\mathbb{E}|_j = \langle 1, 3, 2, 3, 3, 2 \rangle$. For our system, we have $c^{int} = 4, c^{\mathcal{V}} = 24$, and by Theorem 9, $c^{\mathcal{T}} = 6$. Checking the equation of Theorem 10, we have the following memory estimates for $j \in \{0, \ldots, 6\}$:*

$j = 0$: $24 \times 20 + 4 \times (20 + 19 + 16 + 10 + 4 + 3) = 768$.
$j = 1$: $6 \times 4 \times 1 + (24 + 4 \times 1) \times 1 + 24 \times 19 + 4 \times (19 + 16 + 10 + 4 + 3) = 716$.
$j = 2$: $6 \times 4 \times (1 + 3) + (24 + 4 \times 2) \times 3 + 24 \times 16 + 4 \times (16 + 10 + 4 + 3) = 708$.
$j = 3$: $6 \times 4 \times (1 + 3 + 6) + (24 + 4 \times 3) \times 6 + 24 \times 10 + 4 \times (10 + 4 + 3) = 764$.
$j = 4$: $6 \times 4 \times (1 + 3 + 6 + 10) + (24 + 4 \times 3) \times 10 + 24 \times 4 + 4 \times (4 + 3) = 964$.
$j = 5$: $6 \times 4 \times (1 + 3 + 6 + 10 + 4) + (24 + 4 \times 3) \times 4 + 24 \times 3 + 4 \times (3) = 804$.
$j = 6$: $6 \times 4 \times (1 + 3 + 6 + 10 + 4 + 3) = 648$.

We thus have $j^* = M = 6$, and $\mathcal{HT} = \mathcal{BT}$ for our running example.

To mine $\mathcal{HT}$ models, we combine $\mathcal{BT}$Miner and prefix-projection into a novel $\mathcal{HT}$Miner algorithm. The $\mathcal{HT}$Miner algorithm mines its trie model of prefixes $\check{\mathcal{S}}_i$ using the procedure of $\mathcal{BT}$Miner, and its vector model of postfixes $\hat{\mathcal{S}}_i$ using prefix-projection. If the mining algorithm traverses a transitioning node $n \in \mathcal{N}_{j^*}$, it uses the ancestor vector $\vec{a}$ and Theorem 6 to determine valid pattern extensions in the vector model. The complete procedure is given in Algorithm 4.

The worst-case space complexity of $\mathcal{HT}$Miner is given by Lemma 11.

Lemma 11 *The worst-case space complexity of $\mathcal{HT}$Miner is $\mathcal{O}\left(N_{j^*} M \log(N)\right)$, and $\mathcal{O}\left(N_{j^*} M\right)$ for reasonably-sized integers N_{j^*}.*

Proof The largest layer of $\mathcal{HT}$ is $\mathcal{N}_{j^*}$, which is bounded by $|\mathcal{N}_{j^*}| \leq \min\{|\mathbb{E}|^{j^*}, N_{j^*}\}$. At each node of the transitioning layer $\mathcal{N}_{j^*}$ the ancestor vector uses $\mathcal{O}(\min\{|\mathbb{E}|^{j^*}, N_{j^*}\} \min\{|\mathbb{E}|, j^*\})$

Algorithm 4 The $\mathcal{HT}$Miner algorithm.

1: Initialize pattern-node pairs $\{(\mathcal{P}, \mathcal{N}(\mathcal{P}))\}$ for all $\mathcal{P} = \langle \bar{e} \rangle$, $e \in \mathbb{E}$, $e \in \check{\mathcal{S}}_i : |\check{\mathcal{S}}_i| \leq j^*, supp(e) \geq \theta \times N$.

2: Initialize pattern-position pairs $\{(\mathcal{P}, (i,j))\}$ for all $\mathcal{P} = \langle \bar{e} \rangle$, $e \in \mathbb{E}$, $e \in \hat{\mathcal{S}}_i : |\hat{\mathcal{S}}_i| = |\mathcal{S}_i| - j^*, \mathcal{S}_i \in \mathcal{SD}, supp(e) \geq \theta \times N$.

3: **for** each pair $(\mathcal{P}, (i,j))$ **do**

4:     Mine the vector $\mathcal{V}$ pointed to by i starting from position j according to prefix-projection.

5:     Add pattern-position pairs $\{(\langle \mathcal{P}, e_{j'} \rangle, (i,j'))\}$ for all events $e_j \in \mathcal{V}$ that give an extension for $\mathcal{P}$.

6: **for** each pair $(\mathcal{P}, \mathcal{N}(\mathcal{P}))$ **do**

7:     **for** all nodes $n \in \mathcal{N}(\mathcal{P})$ **do**

8:         Traverse all paths $\mathbf{P} = (n, \ldots, n') \in \mathcal{BT}$ part of $\mathcal{HT}$ and mine it according to Algorithm 2.

9:         **for** all vector children $\mathcal{V}$ pointed by node $n' \in Q^1_{\mathcal{N}}$ in $\mathcal{BT}$Miner **do**

10:             Mine the vector $\mathcal{V}$ starting from position 1 according to prefix-projection.

11:             Add pattern-position pairs $\{(\langle \mathcal{P}, e_j \rangle, (i,j))\}$ for all events $e_j \in \mathcal{V}$ that give an extension for $\mathcal{P}$ using ancestor vector $\vec{a}$, and according to Theorem 6.

12:         **for** all vector children $\mathcal{V}$ pointed by node $n' \in Q^2_{\mathcal{N}}$ in $\mathcal{BT}$Miner **do**

13:             Mine the vector $\mathcal{V}$ starting from position 1 according to prefix-projection.

14:             Add pattern-position pairs $\{(\langle \mathcal{P}, e_j \rangle, (i,j))\}$ for all events $e_j \in \mathcal{V}$ that give an extension for $\mathcal{P}$ according to Theorem 6.

15: **return** Set of frequent patterns $\mathcal{P} : \sum_{n \in \mathcal{N}(\mathcal{P})} f_n \geq \theta \times N$.

---

memory. By Theorem 9 this gives the worst-case space complexity of the $\mathcal{BT}$ part of $\mathcal{HT}$ and its corresponding mining algorithm as $\mathcal{O}\left(\min\{|\mathbb{E}|^{j^*}, N_{j^*}\} j^* \log(N)\right)$.

The largest number of sequences of the vector model of $\mathcal{HT}$ is N_{j^*+1}. By Lemma 1, this has a worst-case space complexity of $\mathcal{O}\left(N_{j^*+1}(M - j^*) \log(N)\right)$. As the number of sequences are non-increasing in sequence length, we have $N_{j^*+1} \leq N_{j^*}$. This gives the worst-case space complexity of the vector part of $\mathcal{HT}$ and its corresponding algorithm as $\mathcal{O}\left(N_{j^*}(M - j^*) \log(N)\right)$.

Adding the two complexities together gives the total worst-case space complexity of $\mathcal{O}\left(\min\{|\mathbb{E}|^{j^*}, N_{j^*}\} j^* \log(N) + N_{j^*}(M - j^*) \log(N)\right) \leq \mathcal{O}(N_{j^*} M \log(N))$. Reasonably sized integers N consume constant memory and reduce the complexity to $\mathcal{O}(N_{j^*} M)$. ■

As $j^* \leq M$, $\mathcal{HT}$Miner is always asymptotically smaller and more memory efficient than both $\mathcal{BT}$Miner and vector-based SPM algorithms. Moreover, by Theorem 10, $\mathcal{HT}$Miner is also more memory efficient than $\mathcal{BT}$Miner and vector-based SPM in practice. We indeed observe this in our numerical results, given in the following section.

Table 2: Data sets.

Size	Name	N	$|\mathbb{E}|$	M	$avg(|\mathcal{S}_i|)$	$max(|\mathbb{P}^j|)$	$avg(|\mathbb{P}^j|)$	$\sum_{\mathcal{S}_i \in \mathcal{SD}} |\mathcal{S}_i|$
Large	Criteo	4,373,472,329	214	20	20	1	1	87,469,446,580
	Genome	2,049,780,092	20	37	13	1	1	26,485,605,043
Medium	Twitch	15,524,308	790,100	456	30.3	170	1.2	469,655,703
	Spotify	124,950,054	5	20	2.4	5	1.1	294,302,842
Small	Kosarak	990,002	41,270	2,498	8.1	1	1	8,019,015
	MSNBC	989,818	17	14,795	4.8	1	1	4,698,794

4. Numerical Results

For our numerical tests, we evaluated the effect of different data set models on the performance of state-of-the-art mining algorithms in large-scale SPM. We use PrefixSpan by Han et al. (2001), which forms the basis of almost all state-of-the-art vector-based prefix-projection algorithms, and TreeMiner (Rizvee et al., 2020), which is the only available trie model for SPM. Note that TreeMiner uses several additional mining techniques, such as co-occurrence information of events (Fournier-Viger et al., 2014) for faster SPM, which are not implemented in our mining algorithms for a base-case comparison. Nonetheless, any such algorithmic enhancement developed for time efficiency in the rich literature of SPM can also be implemented on our models without loss of generality.

All algorithms were coded in C++ and executed on a PC with an Intel Xeon W-2255 processor, 256GB of memory, and Ubuntu 20.04.1 operating system.[1] Note that our system memory is considerably higher than an average PC, often ranging between 8-32GB. We limit all tests to one core of the CPU and a 36,000-second time limit.

4.1 Data Sets

We used six real-life data sets in our numerical tests, given in Table 2. The data sets are chosen based on common applications of SPM, namely, click-stream mining and bioinformatics (Fournier-Viger et al., 2017). The data sets are grouped into three sizes: large, medium and small.

The small data sets include Kosarak and MSNBC, which have been benchmark click-stream instances for SPM since the early 2010s (Fournier-Viger et al., 2016). The medium data sets include the Twitch data set (Rappaz et al., 2021), and the Spotify data set (Brost et al., 2019). The Twitch data set includes user streaming content on the live streaming platform Twitch. Here, events $e \in \mathbb{E}$ are defined as a unique streamer, and consecutive streamers watched by a user form a sequence $\mathcal{S}_i \in \mathcal{SD}$. Two consecutive events are assumed to be in the same itemset if they are visited by the user within a one hour time frame. The Spotify data set (Brost et al., 2019) is a collection of user music consumption on the media streaming platform Spotify. Events $e \in \mathbb{E}$ are considered to be the *time-signature* (refer to Brost et al. (2019) for definition of time-signature) of listened to tracks. Events are considered to be in different itemsets if a *context switch* (refer to Brost et al. (2019) for definition of a context switch) occurs between them.

---

1. Our algorithms are open source and available at https://github.com/aminhn/BDTrie

Table 3: Number of frequent patterns by data set and support threshold.

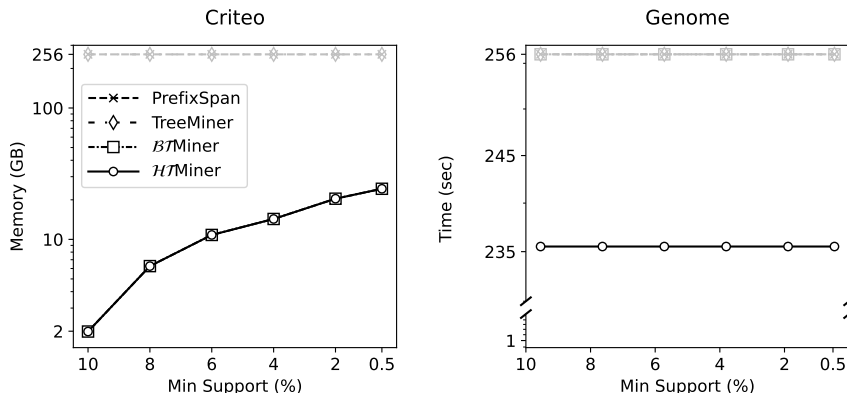|	Criteo (large)					
θ	10%	8%	6%	4%	2%	0.5%
Patterns found	10,705	20,132	(41,559)	(78,598)	(88,434)	(182,541)
	Genome (large)					
θ	10%	8%	6%	4%	2%	0.5%
Patterns found	(264)	(314)	(459)	(552)	(598)	(615)
	Twitch (medium)					
θ	8%	4%	2%	1%	0.5%	0.15%
Patterns found	2	14	160	1,382	13,493	789,685
	Spotify (medium)					
θ	1%	1e-1%	1e-2%	1e-3%	1e-4%	1e-5%
Patterns found	59	377	2,559	16,253	107,257	696,098
	Kosarak (small)					
θ	4%	2%	1%	0.5%	0.25%	0.1%
Patterns found	29	94	329	1,462	8,427	758,141
	MSNBC (small)					
θ	3%	1%	0.3%	0.1%	0.03%	0.01%
Patterns found	22	254	2,014	16,620	303,917	102,108,060

Note. Parenthesis represent lower bound.

The large data sets include the Criteo 1TB click-stream data set (Criteo AI Lab, 2014), and the amino acid representation of the 1000 Genomes data set (Consortium et al., 2015). Unfortunately, the full size of both data sets is larger than 1TB in size and impossible to fit into memory for most systems, including ours. We thus use a subset of each data set for our numerical tests. The Criteo data set includes 20 out of the 40 possible events in each sequence, and the Genome data set includes the first subset "ERR3988796" of genomes as specified by AWS (2023).

An important aspect of mining the above data sets is the imposed support threshold θ. Higher support thresholds lead to fewer frequent patterns, mainly showcasing the time and memory efficiency of building and storing a data set representation in memory. Lower support thresholds lead to a larger number of frequent patterns, mainly showcasing the time and memory efficiency of a mining algorithm. We choose a wide range of thresholds for each data set tailored to observe and showcase both scenarios. The thresholds and number of frequent patterns found for each data set are given in Table 3. Note that when all algorithms exceeded the imposed time, the reported number of frequent patterns is a lower bound.
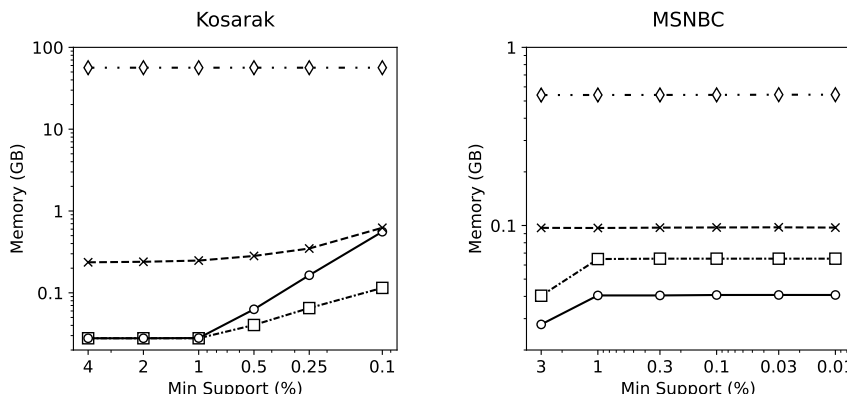
4.2 Memory Consumption

Figure 5 gives the peak memory consumption of all algorithms. In the small data set Kosarak, all algorithms, with the exception of TreeMiner, consume less than 1GB of memory. TreeMiner uses close to 57GB of memory, mainly due to the higher number of events $|\mathbb{E}|$ in this data set which is detrimental to its memory efficiency. The $\mathcal{HT}$Miner algorithm uses less than 0.7GB of memory which amounts to an average improvement of 68% (less than

(a) Large data sets. PrefixSpan exceeds system memory at 15% of the Criteo data set and 62% of the Genome data set. TreeMiner exceeds system memory during mining of the Criteo data set, and at 3% of the Genome data set. $\mathcal{BT}$Miner exceeds system memory at 29% of the Genome data set. $\mathcal{HT}$Miner is the only algorithm capable of mining both data sets within system memory.
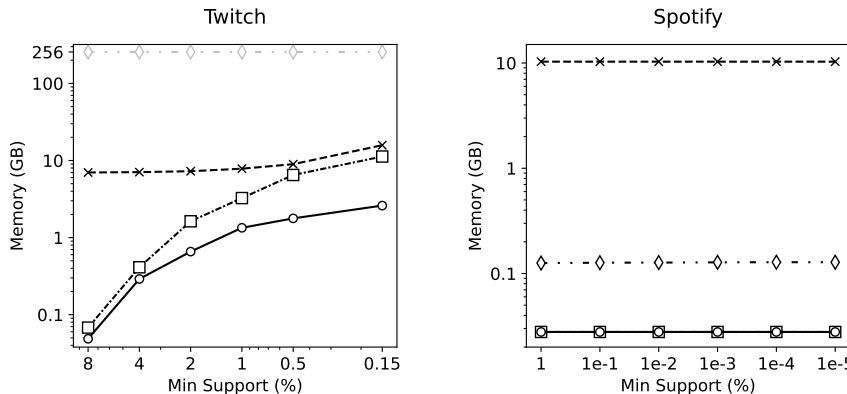
(b) Medium data sets. TreeMiner exceeds system memory at 0.6% of the Twitch data set.

(c) Small data sets.

Figure 5: Peak memory consumption. Algorithms that exceed system memory are shown in gray.

0.2GB) over PrefixSpan and 99.9% (more than 56GB) over TreeMiner. In contrast to all other data sets, $\mathcal{HT}$Miner uses slightly higher memory (0.1GB on average, and 0.4GB at most) compared to $\mathcal{BT}$Miner in lower support thresholds. This is mainly due to the heuristic selection of the transitioning layer, where values $|\mathcal{N}_j|$ were overestimated, leading to a lower than optimal value j^*.

In the small data set MSNBC, $\mathcal{HT}$Miner shows an average improvement of 60% (less than 0.1GB) over PrefixSpan and 93% (approximately 0.5GB) over TreeMiner. Similarly, $\mathcal{BT}$Miner shows an average improvement of 37% (less than 0.1GB) over PrefixSpan and 89% (approximately 0.5GB) over TreeMiner. These improvements show the slight advantage of compactly fitting the data in memory using a trie with efficient labels, even for smaller data sets. $\mathcal{HT}$Miner shows an average improvement of 36% (less than 0.1GB) over $\mathcal{BT}$Miner, mainly due to the longer sequences in the MSNBC data set which are more efficiently modeled by a hybrid trie structure.

In the medium data set Twitch, PrefixSpan uses less than 16GB of memory, $\mathcal{BT}$Miner uses approximately 11GB of memory with an average improvement of 64% over PrefixSpan, and $\mathcal{HT}$Miner uses less than 3GB of memory with an average improvement of 89% over PrefixSpan and 54% over $\mathcal{BT}$Miner. The improvement of $\mathcal{HT}$Miner and $\mathcal{BT}$Miner is higher for larger support thresholds due to a more compact representation of the data set. TreeMiner exceeds the 256GB system memory and can only fit into memory only 0.6% of the Twitch data set. This is again mainly due to the high number of events $|\mathbb{E}|$ which negatively affects the memory efficiency of TreeMiner.

In the Spotify data set, the trie models show more than two orders of magnitude memory saving over the vector model of PrefixSpan. In particular, PrefixSpan uses approximately 10GB of memory while TreeMiner uses 0.12GB of memory with an average improvement of 98.8%. The $\mathcal{BT}$Miner and $\mathcal{HT}$Miner algorithms use less than 0.02GB of memory and show improvements of 99.7% over PrefixSpan and 78% over TreeMiner. These results are mainly due to the low number of events $|\mathbb{E}|$ in the Spotify data set, which allows higher compression in trie models. The $\mathcal{BT}$Miner and $\mathcal{HT}$Miner are equivalent in the Spotify data set as the hybrid algorithm determined the transition length as $j^* = M$, giving a full trie model of the data set for $\mathcal{HT}$Miner. Both algorithms are more efficient than TreeMiner due to their lower memory overhead at each trie node.

The $\mathcal{BT}$Miner and $\mathcal{HT}$Miner algorithms enjoy the most significant improvements in large data sets. In the Criteo data set, PrefixSpan exceeds the 256GB of system memory and is only able to model 15% of the data set. Although TreeMiner can model the entire Criteo data set, it exceeds system memory during its mining algorithm and terminates. This is in contrast to $\mathcal{BT}$Miner and $\mathcal{HT}$Miner, which use at least 2GB and at most 25GB of memory to mine the entire data set. This is more than an order of magnitude memory saving, and potentially amounts to 1.7TB lower memory usage than PrefixSpan requires to model the full data set into memory. This showcases the strength of trie models in modeling overlapping sequences using only a single path, while vector models use multiple vectors. Similar to the Spotify data set, $\mathcal{BT}$Miner and $\mathcal{HT}$Miner are equivalent with $j^* = M$.

The Genome data set is the most memory intensive, with $\mathcal{HT}$Miner the only capable SPM algorithm within system memory. PrefixSpan exceeds system memory at 62% of the Genome data set, TreeMiner at 3%, and $\mathcal{BT}$Miner at 29%. This indicates a lower sequence overlap in the Genome data set, which gives an advantage to $\mathcal{HT}$Miner over $\mathcal{BT}$Miner.

Overall, $\mathcal{HT}$Miner is the most memory-efficient SPM algorithm, closely followed by $\mathcal{BT}$Miner. TreeMiner is the least memory-efficient algorithm, using considerably higher memory than all other algorithms and often failing to model medium to large data sets within system memory. While less efficient, the memory usage of PrefixSpan is still within the range of what is typical for regular PCs for small to medium data sets. However, this is not the case for large data sets, with $\mathcal{HT}$Miner being the only capable SPM algorithm. We conclude that $\mathcal{HT}$Miner provides compelling benefits in memory efficiency for any data set size over all other alternatives.

4.3 Computational Time

Figure 6 gives the computational time of all algorithms. The computational time of any algorithm that exceeds system memory is irrelevant and not reported.

In the small data set Kosarak, TreeMiner is the slowest algorithm, especially for higher support thresholds. This is mainly due to the time required to populate its trie labels, which turned out to be costly in both time and memory efficiency. All other algorithms are mostly the same for higher support thresholds—where fewer patterns are mined. However, the improvements of $\mathcal{HT}$Miner and $\mathcal{BT}$Miner over PrefixSpan become larger for smaller thresholds, with more than a 60% speedup (approximately 800 seconds) for both algorithms. Both algorithms show an average improvement of 95% over TreeMiner, emphasizing the benefits in computational time provided by a less memory intensive trie model.

For the small data set MSNBC, all algorithms are mostly similar in time efficiency on higher support thresholds, but significantly different on lower support thresholds. This indicates that all algorithms are similar in constructing their data set representation, but not in mining it. PrefixSpan and $\mathcal{BT}$Miner require approximately 11,000 seconds to mine the MSNBC data set at the lowest support threshold (mining more than 100,000,000 patterns). In contrast, TreeMiner and $\mathcal{HT}$Miner show close to an order of magnitude faster mining, requiring approximately 1,800 seconds. The faster mining of TreeMiner shows the possible benefits of a richer set of trie labels, which increases mining efficiency when it can be efficiently modeled and fitted into memory. The faster mining of $\mathcal{HT}$Miner shows the benefit of a hybrid data structure in the mining process—equaling the benefits provided by richer trie labels in time efficiency while consuming less memory.

In the medium-size Twitch data set, $\mathcal{BT}$miner and $\mathcal{HT}$Miner show an average speedup of 50% over PrefixSpan. Both the PrefixSpan and $\mathcal{BT}$Miner reach the imposed time limit for the lowest support threshold, where PrefixSpan finds 15% of total patterns and $\mathcal{BT}$Miner finds 65% of total patterns. $\mathcal{HT}$Miner can mine the entire data set in approximately 9,000 seconds, showing at least a 75% speedup. These show the benefits of mining trie paths that model multiple sequences over mining vector representations which model each sequence separately. In particular, the mining algorithm can mine multiple sequences in a pass over a single trie path $(\mathbf{r}, \ldots, n)$, which requires f_n more passes in vector models to mine the same sequences. When f_n is lower, such as in the postfixes of the longer sequences of the MSNBC data set, $\mathcal{HT}$Miner becomes more time-efficient than $\mathcal{BT}$Miner.

Results on the Spotify data set show similar computational time on all algorithms, with at most a $[20 - 40]$ seconds difference. TreeMiner is slightly faster (20 seconds) in higher support thresholds, while $\mathcal{HT}$Miner and $\mathcal{BT}$Miner are slightly faster (10 seconds) for lower
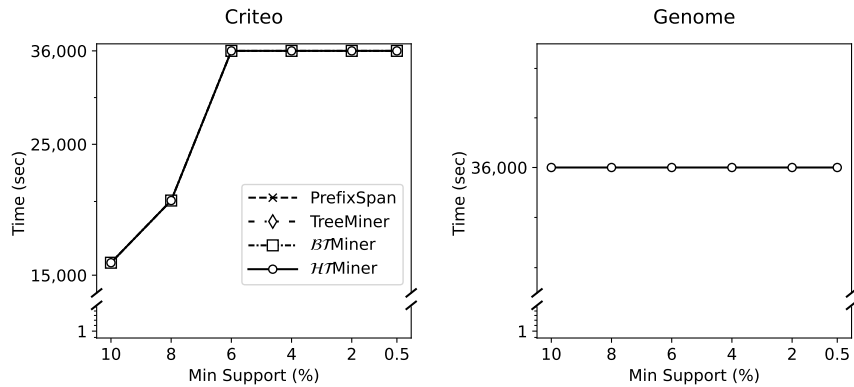
(a) Large data sets. $\mathcal{BT}$Miner and $\mathcal{HT}$Miner exceed the imposed time limit of 36000 seconds when mining below the 6% minimum support threshold for the Criteo data set. $\mathcal{HT}$Miner exceeds the imposed time limit when mining the Genome data set at all thresholds. PrefixSpan and TreeMiner cannot model and mine either data set within system memory and are not reported for computational time.
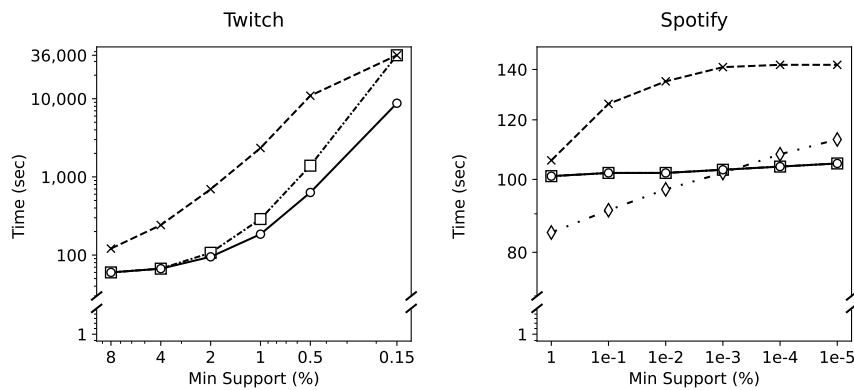
(b) Medium data sets. TreeMiner cannot mine the Twitch data set within system memory and is not reported for computational time.
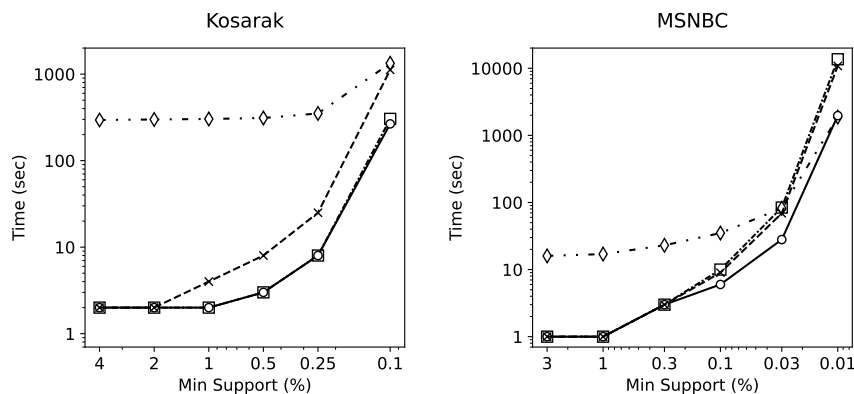
(c) Small data sets.

Figure 6: Computation time.

support thresholds. PrefixSpan is consistently slower (5-40 seconds), with the difference highest for small support thresholds. This is mainly to the high compression of trie models given the low number of events $|\mathbb{E}|$ in this data set. Interestingly, $\mathcal{HT}$Miner and $\mathcal{BT}$Miner stay relatively constant in computational time over all support thresholds. This shows that most patterns are shorter in length, and demonstrates the efficiency of using ancestor labels in fast pruning of pattern extensions.

The $\mathcal{BT}$Miner and $\mathcal{HT}$Miner algorithms show high computational time in large data sets. In the Criteo data set, both algorithms reach the imposed 36,000 seconds time limits for support thresholds lower or equal to 6%. The computational time was considerably high for the Genome data set, where $\mathcal{HT}$Miner reached the time limit at all thresholds. In particular, we observed slower than usual mining speed in the Genome data set as memory consumption was near system limits, increasing the overhead of memory access by the CPU. This shows that it is time-efficient to reduce memory consumption even when operating within—but close to—system limits.

Overall, we observe that $\mathcal{HT}$Miner is also faster than the state of the art, especially for larger data sets. Note that this is generally not the case for memory-efficient algorithms, as memory and time efficiency are often a trade-off. For both time and memory efficiency, we thus conclude that $\mathcal{HT}$Miner is superior to all other SPM alternatives, providing considerably higher memory efficiency and often coupled with higher time efficiency.

5. Conclusion

This paper develops a memory-efficient SPM algorithm using trie models of the data set. Our methodology involves a new binary trie model $\mathcal{BT}$ that stores minimal information at its nodes to compactly represent the data set in memory. We show that this compact representation is sufficient for the subsequent SPM task, and develop a novel $\mathcal{BT}$Miner algorithm to mine all sequential patterns. We build on our trie model to develop a hybrid model $\mathcal{HT}$ that models prefixes with high overlap using a trie model, and postfixes with low overlap using a vector model. We integrate $\mathcal{BT}$Miner and prefix-projection to develop $\mathcal{HT}$Miner, which can effectively mine $\mathcal{HT}$. We proved that $\mathcal{HT}$Miner is always smaller and more memory efficient than pure vector or trie models and their corresponding algorithms.

Numerical results on real-life test instances showed that on small and medium data sets, $\mathcal{HT}$Miner provides, on average, 79% (approximately 5GB) and 90% (approximately 19GB) memory savings compared to the state-of-the-art vector and trie models, respectively. Furthermore, $\mathcal{HT}$Miner was shown to be the only SPM algorithm capable of mining large data sets within 256GB of system memory, potentially saving 1.7TB in memory consumption. While memory efficiency is often a trade-off with time efficiency, $\mathcal{HT}$Miner also showed lower computational time than the state of the art, with an average improvement of 43% over vector models and 54% over trie models.

References

Rakesh Agrawal, Ramakrishnan 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.

Cláudia Antunes and A Oliveira. Sequential pattern mining algorithms: trade-offs between speed and memory, 2004.

AWS. The 1000 genomes project. *Amazon Web Services*, , 2023. Accessed: 2023-11-01.

Nicolas Béchet, Peggy Cellier, Thierry Charnois, Bruno Crémilleux, and Marie-Christine Jaulent. Sequential pattern mining to discover relations between genes and rare diseases. In *2012 25th IEEE International Symposium on Computer-Based Medical Systems (CBMS)*, pages 1–6. IEEE, 2012.

Ferenc Bodon and Lajos Rónyai. Trie: an alternative data structure for data mining algorithms. *Mathematical and Computer Modelling*, 38(7-9):739–751, 2003.

Anindita Borah and Bhabesh Nath. Fp-tree and its variants: Towards solving the pattern mining challenges. In *Proceedings of First International Conference on Smart System, Innovations and Computing*, pages 535–543. Springer, 2018.

Brian Brost, Rishabh Mehrotra, and Tristan Jehan. The music streaming sessions dataset. In *The World Wide Web Conference*, pages 2594–2600, 2019.

Chun-Chieh Chen, Chi-Yao Tseng, and Ming-Syan Chen. Highly scalable sequential pattern mining based on mapreduce model on the cloud. In *2013 IEEE International Congress on Big Data*, pages 310–317. IEEE, 2013.

Chun-Chieh Chen, Hong-Han Shuai, and Ming-Syan Chen. Distributed and scalable sequential pattern mining through stream processing. *Knowledge and Information Systems*, 53(2): 365–390, 2017.

Jinlin Chen. An updown directed acyclic graph approach for sequential pattern mining. *IEEE transactions on knowledge and data engineering*, 22(7):913–928, 2009.

1000 Genomes Project Consortium et al. A global reference for human genetic variation. *Nature*, 526(7571):68, 2015.

Criteo AI Lab. Criteo 1tb click logs dataset. *Criteo AI Lab*, , 2014. Accessed: 2023-11-01.

Maged El-Sayed, Carolina Ruiz, and Elke A Rundensteiner. Fs-miner: efficient and incremental mining of frequent sequence patterns in web logs. In *Proceedings of the 6th annual ACM international workshop on web information and data management*, pages 128–135, 2004.

Philippe Fournier-Viger, Antonio Gomariz, Manuel Campos, and Rincy 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.

Philippe Fournier-Viger, Jerry Chun-Wei Lin, Antonio Gomariz, Ted Gueniche, Azadeh Soltani, Zhihong Deng, and Hoang Thanh 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.

Philippe Fournier-Viger, Jerry Chun-Wei Lin, Rage Uday Kiran, Yun Sing Koh, and Rincy Thomas. A survey of sequential pattern mining. *Data Science and Pattern Recognition*, 1 (1):54–77, 2017.

Fabio Fumarola, Pasqua Fabiana Lanotte, Michelangelo Ceci, and Donato Malerba. Clofast: closed sequential pattern mining using sparse and vertical id-lists. *Knowledge and Information Systems*, 48(2):429–463, 2016.

Wensheng Gan, Jerry Chun-Wei Lin, Philippe Fournier-Viger, Han-Chieh Chao, and Philip S Yu. A survey of parallel sequential pattern mining. *ACM Transactions on Knowledge Discovery from Data (TKDD)*, 13(3):1–34, 2019.

Audrunas Gruslys, Rémi Munos, Ivo Danihelka, Marc Lanctot, and Alex Graves. Memory-efficient backpropagation through time. *Advances in neural information processing systems*, 29, 2016.

Wilhelmiina Hämäläinen and Matti Nykänen. Efficient discovery of statistically significant association rules. In *2008 Eighth IEEE International Conference on Data Mining*, pages 203–212. IEEE, 2008.

Jiawei Han, Jian Pei, Behzad Mortazavi-Asl, Helen Pinto, Qiming Chen, Umeshwar Dayal, and MC Hsu. Prefixspan: Mining sequential patterns efficiently by prefix-projected pattern growth. In *proceedings of the 17th international conference on data engineering*, pages 215–224, 2001.

Jiawei Han, Jian Pei, Yiwen Yin, and Runying Mao. Mining frequent patterns without candidate generation: A frequent-pattern tree approach. *Data mining and knowledge discovery*, 8(1):53–87, 2004.

Amin Hosseininasab, Willem-Jan van Hoeve, and Andre A Cire. Constraint-based sequential pattern mining with decision diagrams. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 33, pages 1495–1502, 2019.

Jen-Wei Huang, Chi-Yao Tseng, Jian-Chih Ou, and Ming-Syan Chen. On progressive sequential pattern mining. In *Proceedings of the 15th ACM international conference on Information and knowledge management*, pages 850–851, 2006.

Jen-Wei Huang, Chi-Yao Tseng, Jian-Chih Ou, and Ming-Syan Chen. A general model for sequential pattern mining with a progressive database. *IEEE Transactions on Knowledge and Data Engineering*, 20(9):1153–1167, 2008.

Bao Huynh, Cuong Trinh, Huy Huynh, Thien-Trang Van, Bay Vo, and Vaclav Snasel. An efficient approach for mining sequential patterns using multiple threads on very large databases. *Engineering Applications of Artificial Intelligence*, 74:242–251, 2018.

Renata Ivancsy and Istvan Vajk. Efficient sequential pattern mining algorithms. *WSEAS Transactions on Computers*, 4(2):96–101, 2005.

Yanqing Ji, Hao Ying, John Tran, Peter Dews, Ayman Mansour, and R Michael Massanari. A method for mining infrequent causal associations and its application in finding adverse drug reaction signal pairs. *IEEE transactions on Knowledge and Data Engineering*, 25(4): 721–733, 2012.

Serdar Kadıoğlu, Xin Wang, Amin Hosseininasab, and Willem-Jan van Hoeve. Seq2pat: Sequence-to-pattern generation to bridge pattern mining with machine learning. *AI Magazine*, 44(1):54–66, 2023.

Junghee Kim, Haemin Jung, and Wooju Kim. Sequential pattern mining approach for personalized fraudulent transaction detection in online banking. *Sustainability*, 14(15): 9791, 2022.

Vance Chiang-Chi Liao and Ming-Syan Chen. Dfsp: a depth-first spelling algorithm for sequential pattern mining of biological sequences. *Knowledge and information systems*, 38 (3):623–639, 2014.

Yi Lu and Christie I Ezeife. Position coded pre-order linked wap-tree for web log sequential pattern mining. In *Pacific-asia conference on knowledge discovery and data mining*, pages 337–349. Springer, 2003.

Nizar R Mabroukeh and Christie I Ezeife. A taxonomy of sequential pattern mining algorithms. *ACM Computing Surveys (CSUR)*, 43(1):1–41, 2010.

Florent Masseglia, Pascal Poncelet, and Rosine Cicchetti. An efficient algorithm for web usage mining. *Networking and Information Systems Journal*, 2(5/6):571–604, 2000.

Florent Masseglia, Pascal Poncelet, and Maguelonne Teisseire. Efficient mining of sequential patterns with time constraints: Reducing the combinations. *Expert Systems with Applications*, 36(2):2677–2690, 2009.

Christos H Papadimitriou and Kenneth Steiglitz. *Combinatorial optimization: algorithms and complexity*. Courier Corporation, 1998.

Jian Pei, Jiawei Han, Behzad Mortazavi-Asl, and Hua Zhu. Mining access patterns efficiently from web logs. In *Pacific-Asia Conference on Knowledge Discovery and Data Mining*, pages 396–407. Springer, 2000.

Jyothi Pillai and OP Vyas. User centric approach to itemset utility mining in market basket analysis. *International Journal on Computer Science and Engineering*, 3(1):393–400, 2011.

Geoff Pleiss, Danlu Chen, Gao Huang, Tongcheng Li, Laurens Van Der Maaten, and Kilian Q Weinberger. Memory-efficient implementation of densenets. *arXiv preprint arXiv:1707.06990*, 2017.

Gwangbum Pyun, Unil Yun, and Keun Ho Ryu. Efficient frequent pattern mining based on linear prefix tree. *Knowledge-Based Systems*, 55:125–139, 2014.

Jérémie Rappaz, Julian McAuley, and Karl Aberer. Recommendation on live-streaming platforms: Dynamic availability and repeat consumption. In *Fifteenth ACM Conference on Recommender Systems*, pages 390–399, 2021.

Redwan Ahmed Rizvee, Mohammad Fahim Arefin, and Chowdhury Farhan Ahmed. Tree-miner: mining sequential patterns from sp-tree. *Advances in Knowledge Discovery and Data Mining*, 12085:44, 2020.

Sumalatha Saleti and RBV Subramanyam. A novel mapreduce algorithm for distributed mining of sequential patterns using co-occurrence information. *Applied Intelligence*, 49(1): 150–171, 2019.

Si Si, Cho-Jui Hsieh, and Inderjit S Dhillon. Memory efficient kernel approximation. *Journal of Machine Learning Research*, 18(20):1–32, 2017.

Edward H Sussenguth Jr. Use of tree structures for processing files. *Communications of the ACM*, 6(5):272–279, 1963.

Petroc Taylor. Volume of data/information created, captured, copied, and consumed world-wide from 2010 to 2020, with forecasts from 2021 to 2025 (in zettabytes). *Statista*, https://www.statista.com/statistics/871513/worldwide-data-created/, 2021.

Pablo Villalobos and Anson Ho. Trends in training dataset sizes, 2022. URL https://epochai.org/blog/trends-in-training-dataset-sizes. Accessed: 05-01-2024.

Jiahong Wang, Yoshiaki Asanuma, Eiichiro Kodama, and Toyoo Takata. A scalable sequential pattern mining algorithm. In *IEEE International Conference on Computer Systems and Applications, 2006.*, pages 437–444. IEEE Computer Society, 2006.

Jianyong Wang, Jiawei Han, and Jian Pei. Closet+ searching for the best strategies for mining frequent closed itemsets. In *Proceedings of the ninth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 236–245, 2003.

Xin Wang, Amin Hosseininasab, Pablo Colunga, Serdar Kadıoğlu, and Willem-Jan van Hoeve. Seq2pat: Sequence-to-pattern generation for constraint-based sequential pattern mining. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 36, pages 12665–12671, 2022.

Sen Yang, Jiankui Guo, and Yangyong Zhu. An efficient algorithm for web access pattern mining. In *Fourth International Conference on Fuzzy Systems and Knowledge Discovery (FSKD 2007)*, volume 2, pages 726–729. IEEE, 2007.

Xiao Yu, Qing Li, and Jin Liu. Scalable and parallel sequential pattern mining using spark. *World Wide Web*, 22(1):295–324, 2019.