A Fast Minimal Infrequent Itemset Mining Algorithm

Kostiantyn Demchuk
MSc

National University of Ireland, Maynooth

Hamilton Institute

Thesis submitted for the degree of Master of Science

May 2014

Head of Department: Prof. Douglas Leith

Supervisor: Prof. Douglas Leith

Research supported by IBM PhD Fellowship and Science Foundation Ireland under Grant No. 11/PI/1177.

Contents

List of Figures

List of Tables

Declaration

Abstract

A novel fast algorithm for finding quasi identifiers in large datasets is presented. Performance measurements on a broad range of datasets demonstrate substantial reductions in run-time relative to the state of the art and the scalability of the algorithm to realistically-sized datasets up to several million records.

Keywords: itemset mining, breadth-first algorithm, frequency-based analysis, k-anonymity, performance, load balancing.

Chapter 1

Introduction

In this thesis we introduce a new algorithm, called Kyiv [DL14], for finding all unique and minimal attribute combinations within a data set. On realistic data sets this algorithm is demonstrated to be considerably faster than state of the art algorithms. We also present an extended Kyiv algorithm designed for infrequent and minimal attribute combinations showing similar performance.

One application of this algorithm is in statistical disclosure control [MH05, HM07, GGM04, Ell07, TMK14]. In statistical disclosure control the released data, for example census microdata, is required to be suitably anonymised. Of particular concern is the removal of quasi-identifiers *i.e.* a subset of attribute values that can uniquely identify one or more entries in a data set. Even apparently innocuous data can act as a quasi-identifier when multiple values are combined together. For example, the seminal study of Sweeney [Swe02] showed that 87% of the US population are uniquely identified by the three attributes gender, zip code and date of birth and demonstrated the use of this fact to de-anonymise published health data. It is therefore of fundamental interest to enumerate those combinations of entries within a dataset which occur either uniquely or sufficiently infrequently.

Other applications of our algorithm include rare itemset mining [KR05, TKD11, SVN10, TKD13]. In rare itemset mining the aim is to discover unusual, but informative, relationships between entries in a data set. This is in contrast to frequent itemset mining where the interest is in discovering relationships which are common within a data set. Rare but interesting items might for example include adverse drug reactions within medical data [JYT$^+$13] and attacker intrusion within network data [REA08, LRRV10, HSE12] *etc.* Since rare items are, by definition, infrequent, a direct approach to discovery is to enumerate the infrequent items and then search for informative relationships, *e.g.* those which are of sufficiently high confidence, within this enumerated set.

The main contributions of the thesis are as follows. We introduce a new algorithm for minimal unique itemset mining, in both sequential and parallel form. The main practical contribution is the speed up of almost two orders of magnitude offered by the proposed algorithm on datasets of realistic complexity. Since execution time is currently the primary bottleneck in finding minimally infrequent itemsets, this is a significant step forward. The main algorithmic novelty (from which the speed up arises) is that by an appropriate choice of data structures and algorithmic formulation the support item test for minimality can be performed in a hugely more efficient manner (essentially with zero cost) than previously possible. A second algorithmic contribution lies in the parallel implementation. Unlike some previous approaches, the proposed approach elegantly allows the work load of parallel threads to be balanced so as to be approximately the same. This means that no single thread becomes the performance bottleneck and therefore ensures better scalability. We note that the speed up in execution time comes at the cost of much higher memory usage. However, since available memory size continues to grow year on year while processor speed has largely stagnated, in many practical applications this trade-off of memory for speed is a favourable one. The new algorithm design is underpinned by new analytic results, the main analytic contribution lying in Lemma 2.4.1 and Corollary 2.4.1. A straightforward generalisation of the unique case of the algorithm is given with the introduction of a new frequency threshold parameter which handles the infrequent case of the algorithm. We present experimental measurements evaluating the performance of the proposed algorithm on a range of synthetic and application datasets, and compare this against the performance of the popular algorithm MINIT [HM07] and of the recently proposed MIWI Miner algorithm [CG13].

First we consider an example of unprotected dataset and note that it is the need for reasonable processing times while avoiding the risk occurring after a publication of such dataset that originally motivated the development of the algorithm. The rest of the thesis is organised as follows. In Section 1.1 we provide with the comprehensive overview of literature on the subject and in Section 1.2 we introduce notation and some basic definitions. Then, in Chapter 2 we introduce our new algorithm, explaining its rationale and performance and in Section 2.5 we present performance measurements for realistic data sets; in Chapter 3 we extend the problem to infrequent itemset mining and present performance measurements for the same set of realistic data sets. Finally we summarise our conclusions and

Table 1.1: Extracts from AOL web search dataset

ID	Query	Date	Link Clicked
3302	uterine bleeding and coumadin	2006-03-23 11:23:35	www.nlm.nih.gov
3302	children who have died from moms postpartum depression	2006-03-24 15:41:21	www.cbsnews.com
6993	american heart association	2006-03-23 18:29:34	www.americanheart.org
6993	high blood pressure	2006-03-23 18:37:10	
7005	notice of demand to pay judgment form	2006-03-21 18:49:01	www.sba.gov
7005	free personal credit report	2006-03-20 11:26:42	www.experian.com
4417749	shadow lake subdivision gwinnett county georgia	2006-04-24 21:48:01	
4417749	jarrett t. arnold eugene oregon	2006-03-23 21:48:01	www2.eugeneweekly.com

share insights for the future work in Chapter 4.

1.0.1 Motivating Example

In 2006 AOL released web search log data in which user identities had been concealed (replaced by unique identity numbers) but other data was left unchanged. Table 1.1 presents some entries from this AOL data set. It can be seen that the search queries and pages clicked are potentially sensitive in nature and it was further demonstrated that de-anonymisation of users was possible *e.g.* that user #4417749 was Thelma Arnold [BZ06].

We consider quasi-identifiers within the search data for the first 65,517 users in more detail. These users carried out 3,558,412 searches using 1,216,655 distinct queries. Of these queries, 736,967 occur only once within the data set and so are potential quasi-identifiers. Restricting consideration to the first three words of each query reduces the number of unique queries to 617,510, while restricting to the first two words reduces this to 488,138 and restricting to the first word only yields 276,074 unique queries. Hence, it can be seen that simply truncating the search queries is not sufficient to prevent a large number of the search queries from acting as quasi-identifiers.

One simple and direct approach to masking these unique queries is to group unique queries together into sets of queries where each set consists of k unique queries, k being a design parameter. In the data set we now replace the query by a reference to the set containing the query. In this way it is ensured that every query value in the modified data set occurs at least k times within the data set. We performed this data transformation on the AOL data using a value $k = 5$. In addition, we performed a similar transformation to the web page clicked by a user following a query, also with $k = 5$. After these changes each query value and each web page clicked value occurs at least $k = 5$ times within the modified data set. Nevertheless, when this query value is combined with the web page clicked value 586,698 of these pairs are still unique within the modified data set. In the unmodified data set there are 1,030,387 unique pairs, so the grouping of query of page clicked values has also reduced the number of unique pairs. However, in view of the large value of unique pairs it is evidently not sufficient to just consider individual entries but rather it is also necessary to consider combinations of entries when anonymising a data set.

The difficulty with considering combinations of entries is that the number of combinations to be tested grows combinatorially and so in realistically sized data sets highly efficient algorithms are needed to test even combinations of 3 or 4 entries. One solution to this combinatorial growth is to use sampling. For example, a subset of entries may be drawn uniformly at random from the full dataset, the number of attribute combinations occurring with less than a specified frequency within this subset determined and then this information is statistically extrapolated to the full dataset. Sampling reduces the computational burden but also carries the obvious risk of missing infrequently occurring entries. More efficient algorithms allow consideration of larger samples and so potentially significantly reduce this risk.

Note that the set of unique or sufficiently infrequently occuring combinations of items within a data set is useful not just for verifying that restrictions on quasi-identifiers are respected by a data set but, when quasi-identifiers are present, this set is also useful as input to tools such as that in [LDR05] for modifying the data that require prior knowledge of the fields which act as quasi-identifiers. In the above AOL example the set of unique combinations is the precisely set of elements from which grouped values need to be constructed.

1.1 Literature Review

Frequent pattern mining comprises association rule induction, frequent item set mining and frequent graph mining. Its application areas include market basket analysis, web link analysis, genome project and drug design. Frequent itemset mining has been the subject of extensive study by the data-mining community (*e.g.* see [AMS+96] and the many papers which cite this seminal work). In [AMS+96], the famous Apriori algorithm was introduced. Since then a series of frequent itemset mining algorithms have been developed with the most popular one, described in [HPYM04], making use of a frequent-pattern tree approach. Although these algorithms differ in the mechanics i.e. how they traverse the search tree, whether they exploit candidate generation or not, they all incorporate a monotonicity principle of pruning the search tree, the so called Apriori property which stands for the fact that no superset of an infrequent item set can be frequent. This key feature is reused in the infrequent itemset mining literature, which has attracted significantly less attention, but is of growing interest.

Apart from the Apriori property, few effective pruning techniques have been reported in the itemset mining literature. An attempt to classify item frequency relations was made by [CG07], where the deduction rules had been well studied. But we believe that many more mathematical properties are yet to be discovered such as Lemmata 2.4.1 and 3.2.1 in this thesis which reduce the cost of the mining operation and increase the speed of algorithm execution time.

The first algorithm for unique itemset mining (the extreme case of infrequent itemset mining) appears to be SUDA (special unique detection algorithm) proposed in [EMF02]. The SUDA algorithm essentially generates all possible column subsets in a depth-first manner and scans the input dataset for unique and minimal (such that no proper subset can be unique) patterns (up to a user-specified size) in those columns. The actual search is supported by partitioning the rows of the dataset according to the value of each of the columns in a given column subset, then unique records are extracted and checked for minimality. This process continues until all possible column subsets have been considered (that is the prefix tree is traversed) with the user-specified size of subsets in mind as an upper bound. Although SUDA expands the risk assessment capability compared to initial developments in the statistical disclosure control, it can only serve small datasets due to complexity constraints. Consequently, SUDA was followed shortly afterwards by the development of the SUDA2 algorithm [MHK08, MH05], which is available in the sdcMicro package for R [TMK13] and is essentially the state-of-the-art algorithm in this area, being used by the UK and Australian national statistics offices [HMM+09] and supported by IHSN (International Household Survey Network).

SUDA2 defines a minimal unique itemset (MUI) I in a dataset as a set of items that satisfies two properties: (i) the pattern described by I appears in exactly one row of the dataset (uniqueness), and (ii) every proper subset of I appears in multiple rows of the dataset (minimality). SUDA2 brought a new method for representing the search space and new observations about the properties of MUIs to prune and traverse this space — it is a recursive depth-first search that utilises repetition counts of individual items and the following properties:

- *Support Row Property.* Given a MUI I of size k at dataset row i there must be at least k rows other than i containing itemsets that differ from I by exactly one item, these are called the support rows of I;
- *Uniform Support Property.* A unique itemset that contains the same item in each of its support rows cannot be minimal;
- *Recursive Property.* Suppose a MUI I of size k appears in a row i in a dataset D. Let S be the $k-1$ items remaining after removing one item, I_j, from I. Let D_{I_j} be the subset of D consisting of only those rows containing I_j. Then it can be seen that S is a MUI of size $k-1$ in D_{I_j};
- *Perfect Correlation Property.* Two perfectly correlated items (that is, they appear in the same set of rows) can not coexist in a MUI.

The Kyiv algorithm, proposed in this thesis, uses three of these properties but does not use the recursive property. SUDA2 efficiently tests itemsets for uniqueness and minimality using the above properties. It is worth mentioning that SUDA2 also is not a candidate generation algorithm and so is suitable for a limited memory environment. Also SUDA2 lends itself readily to parallelisation by allocating disjoint subtrees to different threads which then carry out a depth-first search on the subtree (using the divide-and-conquer nature of the algorithm). However, the work allocated amongst threads may be imbalanced depending on the size and complexity of the subtree assigned to a thread, leading to performance being constrained by the slowest running thread. Thus an optimal scheduling of the work units is required. A number of strategies for that are discussed in [HMM+09], but the computation of a particular subtree is unpredictable and highly dependent upon the input dataset and this makes the choice of scheduling strategy difficult.

Early work on infrequent (rather than only minimal) itemset mining initially made use of variants of the Apriori algorithm for frequent itemset mining, see [DZNJ07] and references therein, but quickly moved on to algorithms specifically tailored to the infrequent mining task. Almost simultaneously three specialised infrequent itemset algorithms were proposed by [ZY07], [SNV07] and [HM07]. In [ZY07] two schemes (the matrix based scheme (MBS) and hash based scheme (HBS)) are proposed to mine association rules among rare items, involving a direct search of item sequences contained in a database using only two scans with pruning based on frequency and interest parameters with the latter defined as $interest(I, J) = |supp(I \cup J) - supp(I)supp(J)|$, where I, J are itemsets and $supp(\cdot)$ is the frequency of itemset in a dataset. HBS was found to perform better than MBS for that specific task, though the work itself is not purely about minimal infrequent itemset mining which is of the main interest here. In [SNV07] an algorithm referred to as ARIMA (a rare itemset miner algorithm) is proposed, and later refined in [SVNG12] by the addition of a depth-first search to narrow the space of frequent itemsets to frequent generators only (an itemset is a generator if it has no proper subset with the same support). In [HM07] the MINIT (minimal infrequent itemsets) algorithm is proposed. MINIT uses a recursive depth-first search with pruning, similarly to the SUDA2 algorithm developed by the same group, and is often used as the baseline algorithm against which the performance of other infrequent mining algorithms is compared. In fact, MINIT is the first successful algorithm in the field of minimal infrequent itemset mining (as SUDA2 is in the field of minimal unique itemset mining). The difference between the two lies in the way they store an input dataset: in SUDA2 the dataset matrix contains integers (and so effectively it is a two-dimensional array), whereas MINIT leverages a binary matrix form. The Minimum Support Property in [HM07] is new saying that an item i must have frequency $supp(i) \geq k + \tau - 2$ in order for i to be part of a minimal τ-infrequent itemset of size k for a given fixed frequency threshold τ (see Section 1.2 for a definition of τ-infrequent itemset). Even though MINIT's parallel form has not yet been presented to the best of our knowledge, it is likely that the performance analysis and the potential bottlenecks observed in [HMM^{+}09] will be repeated due to its design.

In [TSB09, TS13] a breadth-first algorithm, Rarity, aiming at finding not necessarily minimal infrequent itemsets, is introduced. Whereas other algorithms start from small itemsets and increase the size as they search, Rarity takes the opposite approach and proceeds from the largest rare itemsets to smaller ones (referred to in [TSB09, TS13] as a top-down strategy) cutting frequent itemsets. Rarity was found to be faster than ARIMA in general requiring more memory. In [GMB11] a pattern-growth recursive depth-first approach is proposed for minimal infrequent itemset mining and two algorithms called IFP_min and IFP_MLMS (multiple level minimum support) are introduced. These algorithms make use of a residual and projected trees: a residual tree of a particular item is a tree representation of the residual database corresponding to the item, that is the entire database with transactions containing the item removed; on the other hand, the projected database corresponds to the set of transactions that contain that item. Then the inverse FP-tree, a compact representation of input dataset, is built based on the residual and projected trees, from which the algorithm mines needed itemsets. It is observed that there exists a frequency threshold below which MINIT generally outperforms IFP_min and above which IFP_min outperforms MINIT. IFP_min is also observed to outperform MINIT for large dense datasets.

Recently, [CG13] extends consideration to the more general task of discovering infrequent weighted itemsets (IWIs) from a weighted dataset. To address this problem, the IWI-support measure is defined as a weighted frequency of an itemset in the dataset. Occurrence weights are derived from the weights associated with items in each transaction by applying a given cost function, that is, two IWI-support measures are used: the IWI-support-min (IWI-support-max) measure, which relies on a minimum (maximum) cost function – the itemset frequency in a given transaction is weighted by the weight of its least (most) interesting item. This way the minimal IWI mining (MIWI Miner) algorithm is introduced, which also belongs to the family of FP-Growth algorithms with the following principles:

- early FP-tree node pruning driven by the maximum IWI-support constraint;
- cost function-independence: it works in the same way regardless of which constraint (either IWI-support-min or IWI-support-max) is applied;
- early stopping of the recursive FP-tree search to avoid extracting non-minimal IWIs (Apriori-like property).

When a weighting of unity is associated with every itemset then this reduces to the infrequent itemset mining problem. For the datasets considered in [CG13], MIWI Miner is demonstrated to significantly outperform MINIT for infrequent itemset mining. However, it is worth noting that the performance comparison in [CG13] is made only for a small number of datasets.

1.2 Preliminaries

A dataset A is a table with n rows and m columns. The columns in this table contain categorical or finite range continuous data (such as age, income, zip code *etc*). Formally,

Definition 1.2.1 (Item)**.** *An item a is a triple (v, j_a, R_a) in A, where $v \in \mathbb{N}$ is its value, $j_a \in \{1, \ldots, m\}$ is the column of A containing v, and $R_a \subseteq \{1, \ldots, n\}$ is the set of A rows in which the item appears.*

Note that the column in which it appears distinguishes an item, the same value appearing in two different columns being treated as two different items. This is in line with previous work on infrequent itemset mining. Also observe that we consider items with values from the set of positive integer (natural) numbers $\mathbb{N}$, but since any countable set can be mapped on to the integers this restriction is mild (while real values are excluded, finite-precision values are admissible).

Let I_A denote the set of all items in A. We define uniqueness, rareness and uniformity of items in the natural way, as follows:

Definition 1.2.2 (Uniqueness)**.** *An item $a \in I_A$ is unique if $|R_a| = 1$. That is, it occurs in dataset A exactly once. We let $\delta_A \subseteq I_A$ denote the set of unique items in I_A.*

Definition 1.2.3 (τ-Infrequency)**.** *An item $a \in I_A$ is τ-infrequent if it has frequency less than τ* i.e. *$|R_a| \leq \tau$ and so the item occurs in τ or fewer rows of the dataset. We let $r_{A,\tau} \subseteq I_A$ denote the set of τ-infrequent items in I_A. Unless otherwise stated, we confine consideration to τ values less than n, since trivially all elements of the dataset are n-infrequent. Usually $0 < \tau \ll n$.*

Definition 1.2.4 (Uniformity)**.** *Let $B \subseteq \{1, \ldots, n\}$ be a subset of row indices from dataset A, and let $I_B = \{a \in I_A : R_a \cap B \neq \emptyset\}$. An item a is said to be uniform in I_B if $|R_a \cap B| = |B|$. That is, item a occurs in every row of subtable B. We let $U_A = \{a \in I_A : |R_a| = n\}$ denote the set of uniform items in I_A.*

Example 1.2.1. *For dataset*

$$A = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 1 & 2 & 7 & 4 \\ 1 & 6 & 3 & 4 \\ 5 & 2 & 3 & 4 \end{bmatrix}$$

we have

$$\begin{aligned}
I_A &= \{(1,1,\{1,2,3\}),(2,2,\{1,2,4\}),(3,3,\{1,3,4\}),(4,4,\{1,2,3,4\}),\\
&\quad (5,1,\{4\}),(6,2,\{3\}),(7,3,\{2\})\}.\\
\delta_A &= \{(5,1,\{4\}),(6,2,\{3\}),(7,3,\{2\})\}.\\
U_A &= \{(4,4,\{1,2,3,4\})\}.\\
r_{A,\tau} &= \begin{cases} \emptyset & \textit{if } \tau \le 0 \\ \delta_A & \textit{if } 0 < \tau < 3 \\ I_A \setminus U_A & \textit{if } \tau = 3 \\ I_A & \textit{if } \tau > 3 \end{cases}.
\end{aligned}$$

□

Definition 1.2.5 (Frequency)**.** *An itemset $I \subseteq I_A$ is a set of items. A k-itemset refers to an itemset of cardinality k. We let $R_I = \bigcap_{a\in I} R_a$ denote the set of rows in which all items of I appear, and we refer to $|R_I|$ as the frequency of itemset I.*

Definition 1.2.6 (Unique and Minimal Itemsets)**.**
An itemset $I \subseteq I_A$ is unique and minimal if:

1. *Uniqueness: $|R_I| = 1$;*
2. *Minimality: $|R_S| > 1 \ \forall S \subset I,\ S \neq \emptyset$.*

Definition 1.2.7 (τ-Infrequent and Minimal Itemsets)**.**
An itemset $I \subseteq I_A$ is τ-infrequent and minimal if:

1. *τ-Infrequency: $|R_I| \le \tau$;*
2. *Minimality: $|R_S| > \tau \ \forall S \subset I,\ S \neq \emptyset$.*

We refer to the τ-infrequent and minimal itemsets as being the unique and minimal itemsets *and often drop any τ subscripts to streamline notation when $\tau = 1$.*

Note that to establish minimality in Definition 1.2.6 (1.2.7) it is only necessary to test that $|R_S| > 1$ ($|R_S| > \tau$) for sets $S \subset I$ of size $|I|-1$ since $R_{S'} \supseteq R_S \ \forall S' \subset S$. These $|I|-1$ subsets are referred to as the *support itemsets* of I. Notice also that itemsets of size 1 (items) are trivially minimal.

We denote the set of all unique and minimal itemsets by $\mathcal{I}_A \subseteq 2^{I_A}$ and the set of all τ-infrequent and minimal itemsets by $\mathcal{I}_{A,\tau} \subseteq 2^{I_A}$, where 2^{I_A} denotes the set of all subsets of I_A. We use calligraphic script to indicate that $\mathcal{I}_A$ is a set of sets (similarly for $\mathcal{I}_{A,\tau}$) and to distinguish it from the set of items I_A. Notice that $\mathcal{I}_{A,\tau} = \mathcal{I}_A$ when $\tau = 1$.

Chapter 2

Minimal Unique Itemset Mining

In this chapter we introduce a new algorithm for efficiently finding all of the unique and minimal k-itemsets up to a user specified size k_{max}, $1 \leq k \leq k_{max} \leq m$.

2.1 Pre-processing

We begin by observing that uniform items $u \in U_A$ can be deleted from I_A as they are not unique and so cannot belong to the set of unique and minimal itemsets $\mathcal{I}_A$ (moreover uniform items cannot form a minimal unique itemset as they break minimality). Further, the set of unique items δ_A can be readily identified. The remaining set of non-uniform and non-unique items $I'_A = I_A \setminus U_A \setminus \delta_A$ can be partitioned into sets L_A and $\bar{L}_A = I'_A \setminus L_A$ such that (i) $R_a \neq R_b$ $\forall a, b \in L_A$, (ii) $\forall c \in \bar{L}_A$ there exists $d \in L_A$ with $R_c = R_d$. That is, within set L_A no items share the same set of rows. This partitioning can be achieved in the obvious way. Namely, for any set of items in I'_A which share the same set of rows, add one of these items to L_A and the rest to $\bar{L}_A$. Revisiting Example 1.2.1, we have $L_A = \{(1, 1, \{1, 2, 3\}), (2, 2, \{1, 2, 4\}), (3, 3, \{1, 3, 4\})\}$.

The partitioning into L_A and $I'_A \setminus L_A$ possesses the following useful property:

Proposition 2.1.1. *Let $W \subseteq L_A$ be a minimal unique itemset. Let $w' \in I_A \setminus L_A$ with $R_w = R_{w'}$ for some $w \in W$. Then $W \setminus \{w\} \cup \{w'\}$ is also a minimal unique itemset.*

Proof. Since W is minimal and unique, $|R_W| = 1$ and $|R_S| > 1$ for all subsets $S \subset W$ such that $|S| = |W| - 1$, $S \neq \emptyset$. Let $W' = W \setminus \{w\} \cup \{w'\}$. We have $R_{W'} = R_{W \setminus \{w\}} \cap R_{w'} = R_{W \setminus \{w\}} \cap R_w = R_W$ since $R_w = R_{w'}$. Hence, $|R_{W'}| =$

$|R_W| = 1$. Now consider any subset $S' \subset W'$ such that $|S'| = |W'| - 1$. We have $|W'| - 1 = |W| - 1$ and either (i) $S' = S$ when $w \notin S$ or (ii) $S' = S \setminus \{w\} \cup \{w'\}$ when $w \in S$, where $S \subset W$, $|S| = |W| - 1$. Thus, either (i) $R_{S'} = R_S$ or (ii) $R_{S'} = R_{S\setminus\{w\}} \cap R_{w'} = R_{S\setminus\{w\}} \cap R_w = R_S$, respectively. That is, $|R_{S'}| = |R_S| > 1$ and we are done. □

It follows that the importance of the partitioning into L_A and $I'_A \setminus L_A$ is that after finding the set of unique and minimal itemsets $\mathcal{L}_A \subset 2^{L_A}$ of L_A, the set of unique and minimal itemsets $\mathcal{I}_A \subset 2^{I_A}$ of I_A can be obtained immediately. Namely,

Proposition 2.1.2. *For any partition* $(L_A, I'_A \setminus L_A)$ *the following holds:* $\mathcal{I}_A = \mathcal{L}_A \cup \bar{\mathcal{L}}_A \cup \delta_A$, *where* $\bar{\mathcal{L}}_A = \{I \setminus \{a\} \cup \{b\} : I \in \mathcal{L}_A, a \in I, b \in \bar{L}_A, R_a = R_b\}$.

Proof. The proposition states that itemset $I \in \mathcal{I}_A \iff I \in \mathcal{L}_A \cup \bar{\mathcal{L}}_A \cup \delta_A$. "$\Leftarrow$" If itemset $I \in \mathcal{L}_A$ or $I \in \delta_A$ then I is minimal and unique and so $I \in \mathcal{I}_A$; if $I \in \bar{\mathcal{L}}_A$ then, by Proposition 2.1.1, I is minimal and unique and so $I \in \mathcal{I}_A$. "$\Rightarrow$" Suppose $I \in \mathcal{I}_A$. First of all observe that $\tilde{\mathcal{I}}_A = \mathcal{I}_A$, where $\tilde{I}_A = I_A \setminus U_A$ and $\tilde{\mathcal{I}}_A$ is the set of minimal and unique itemsets in $2^{\tilde{I}_A}$. This holds because $I \cap U_A = \emptyset$ for any $I \in \mathcal{I}_A$ (suppose $u \in I$, $u \in U_A$ and I is minimal and unique, then $R_I = R_{I\setminus\{u\}} \cap R_u = R_{I\setminus\{u\}}$ since R_u contains all rows of A; thus $|R_{I\setminus\{u\}}| = |R_I| = 1$ what contradicts the minimality of I). Further, we have $\tilde{\mathcal{I}}_A = \hat{\mathcal{I}}_A \cup \delta_A$ where $\hat{I}_A = I_A \setminus U_A \setminus \delta_A$ and $\hat{\mathcal{I}}_A$ is the set of minimal and unique itemsets in $2^{\hat{I}_A}$. This is because the elements of δ_A are minimal and unique individual items and so if $I \in \tilde{\mathcal{I}}_A$ then either (i) $I \cap \delta_A = \emptyset$ or (ii) $|I| = 1$, $I \in \delta_A$ (if $|I \cap \delta_A| > 1$ then $|I| > 1$ and $\forall a \in I \cap \delta_A : a \in I, a \neq \emptyset, |R_a| = 1$ as $a \in \delta_A$ and so I is not minimal; if $|I \cap \delta_A| = 1$ and $|I| > 1$ then I is not minimal). Hence, we have that $\mathcal{I}_A = \hat{\mathcal{I}}_A \cup \delta_A$. Now $\hat{I}_A = L_A \cup \bar{L}_A$ with $L_A \cap \bar{L}_A = \emptyset$. Hence, if $I \in \hat{\mathcal{I}}_A$ and $I \cap \bar{L}_A = \emptyset$ (so $I \subseteq L_A$) then $I \in \mathcal{L}_A$. If $I \in \hat{\mathcal{I}}_A$ and $I \cap \bar{L}_A \neq \emptyset$ then $I \in \bar{\mathcal{L}}_A$ and we are done. Notice that proof works for any partition $(L_A, I'_A \setminus L_A)$. □

In light of Proposition 2.1.2, our goal can therefore be simplified to finding all unique and minimal k-itemsets of L_A, $1 \leq k \leq k_{max}$.

Example 2.1.1. *For dataset*

$$A = \begin{bmatrix} 1 & 2 & 3 & 4 & 8 \\ 1 & 2 & 7 & 4 & 8 \\ 1 & 6 & 3 & 4 & 8 \\ 5 & 2 & 3 & 4 & 9 \end{bmatrix}$$

we have

$$\begin{aligned} I_A &= \{(1,1,\{1,2,3\}),(2,2,\{1,2,4\}),(3,3,\{1,3,4\}),(4,4,\{1,2,3,4\}),\\ &\quad (5,1,\{4\}),(6,2,\{3\}),(7,3,\{2\}),(8,5,\{1,2,3\}),(9,5,\{4\})\}.\\ \delta_A &= \{(5,1,\{4\}),(6,2,\{3\}),(7,3,\{2\}),(9,5,\{4\})\}.\\ U_A &= \{(4,4,\{1,2,3,4\})\}. \end{aligned}$$

The remaining set of non-uniform and non-unique items is

$$I'_A = I_A \backslash U_A \backslash \delta_A = \{(1,1,\{1,2,3\}),(2,2,\{1,2,4\}),(3,3,\{1,3,4\}),(8,5,\{1,2,3\})\}.$$

The set I'_A can be partitioned into sets $L_A = \{(1,1,\{1,2,3\}),(2,2,\{1,2,4\}),(3,3,\{1,3,4\})\}$ and $\bar{L}_A = I'_A \setminus L_A = \{(8,5,\{1,2,3\})\}$ such that (i) $R_a \neq R_b$ $\forall a,b \in L_A$ ($\{1,2,3\} \neq \{1,2,4\} \neq \{1,3,4\}$ and $\{1,2,3\} \neq \{1,3,4\}$), (ii) $\forall c \in \bar{L}_A$ there exists $d \in L_A$ with $R_c = R_d$ (for $(8,5,\{1,2,3\})$ there is $(1,1,\{1,2,3\})$ in L_A).

Denote $a = (1,1,\{1,2,3\})$, $b = (2,2,\{1,2,4\})$, $c = (3,3,\{1,3,4\})$ and $d = (8,5,\{1,2,3\})$. Proposition 2.1.1 says that if $\{a,b,c\} \subseteq L_A$ is a minimal unique itemset and $d \in I_A \setminus L_A$ with $R_d = R_a$ then $\{d,b,c\}$ is also a minimal unique itemset. Proposition 2.1.2 says that for our chosen partition $(L_A, I'_A \setminus L_A)$ the set of all minimal unique itemsets $\mathcal{I}_A$ can be obtained from the set $\mathcal{L}_A \subset 2^{L_A}$, $\bar{\mathcal{L}}_A$ and δ_A, in our case: $\{a,b,c\} \in \mathcal{L}_A$, $\{d,b,c\} \in \bar{\mathcal{L}}_A$ and unique items from δ_A. $\square$

2.2 Pruning the Search space

Considering the items in L_A to be an alphabet, all of the possible words in the form of ordered sequences that can be built from L_A can be represented by a prefix tree. For example, when $L_A = \{a,b,c,d,e\}$, the associated prefix tree is shown in the Figure 2.1. By starting at the root and traversing the branches of the tree, every possible ordered sequence of letters can be obtained.

In principle, the unique and minimal k-itemsets of L_A can be found by traversing every branch of the tree to depth k_{max} and testing each sequence of items obtained for uniqueness and minimality. However, efficiency can be increased if it is possible to avoid fully traversing every branch i.e. the tree can be pruned.

Basic pruning can be achieved using following fundamental property of itemsets:
Proposition 2.2.1 (Monotonicity). *Let I be an itemset. If I is not minimal*

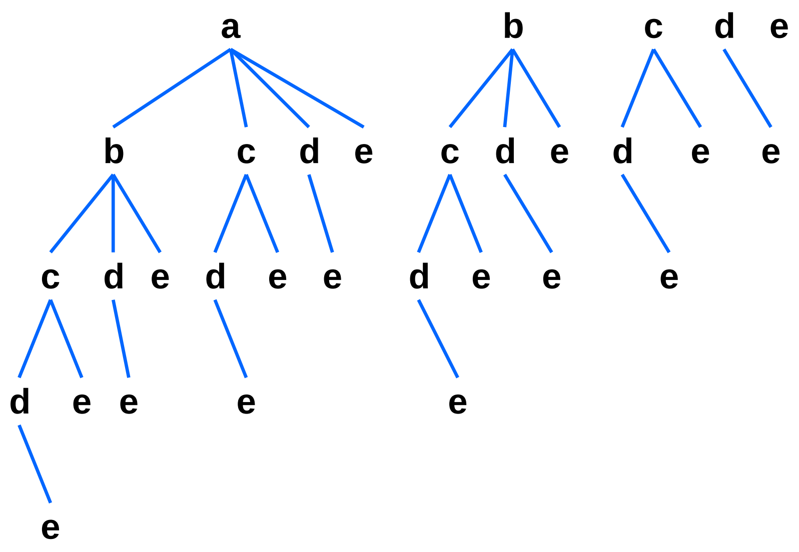

Figure 2.1: Prefix tree for the alphabet $L_A = \{a, b, c, d, e\}$. By starting at the root and traversing the branches of the tree, every possible ordered sequence of letters can be obtained *e.g.* traversing the far left-hand branch yields the sequence *abcde*.

then no superset of I can be minimal.

Proof. Since I is non-minimal there exists $S \subset I, S \neq \emptyset$ such that $|R_S| \leq 1$. It follows that $\forall J \supset I$ there exists $S \subset J$, $S \neq \emptyset$ such that $|R_S| \leq 1$ and so J is also non-minimal. □

Hence, as soon as we determine that the sequence of items in an itemset is non-minimal, we can terminate traversal of that branch of the tree. Note that similar pruning is not possible based on uniqueness since a superset of an itemset I can be unique even if I is not unique due to the decrease in frequency as more and more items are added to an itemset.

Importantly, the prefix tree associated with itemset L_A is not unique since the tree depends on how we choose to order the items in L_A. In general, it is challenging to determine an ordering of items in L_A which minimises the number of vertices which need to be traversed in the prefix tree in order to find the set $\mathcal{L}_A$ of unique and minimal itemsets of L_A. We revisit this question later, in Section 2.5.2.4, but note here that sorting the items of L_A into ascending order using the following item ordering is efficient for a wide range of datasets.

Definition 2.2.1 (Ascending Order)**.** *We order items $a < b$ if (i) $|R_a| < |R_b|$ or (ii) $|R_a| = |R_b|$ and $j_a < j_b$ or (iii) $|R_a| = |R_b|$, $j_a = j_b$ and $\min R_a < \min R_b$.*

Note that due to the pre-processing and partitioning used to obtain L_A, for any items $a \in L_A$, $b \in L_A \setminus \{a\}$ we must have either $a < b$ or $b < a$ *i.e.* strict total order (if $j_a = j_b$, $\min R_a = \min R_b$ then items a and b are both in the same column j_a and row $\min R_a$ of the dataset and so we must have $a = b$, but this contradicts the fact that $b \in L_A \setminus \{a\}$). We let $L_A^{<}$ denote a list of the items in L_A sorted in ascending order.

2.3 Potential Performance Bottlenecks

To evaluate whether an itemset I is minimal or not, we use the support itemset test to verify Definition 1.2.6(2). To evaluate whether an itemset I is unique, we intersect the rows of the elements in I to obtain $R_I = \cap_{a \in I} R_a$ and test whether $|R_I| = 1$ to verify Definition 1.2.6(1). Both of these tests are potentially expensive.

The support itemset test requires enumerating the subsets $S \subset I$, $|S| = |I| - 1$, and calculating $R_S = \cap_{a \in S} R_a$ for each subset. As already noted, testing for uniqueness requires calculating $R_I = \cap_{a \in I} R_a$. For large tables, the row sets R_a may be large and so time consuming to obtain, *e.g.* if the approach taken is to scan the dataset for item a and record the rows in which a appears, plus additionally the complexity of calculating R_I in the obvious manner scales as $O(|I| \min_{a \in I} |R_a|)$.

2.4 Kyiv Algorithm

The Kyiv algorithm performs a breadth first search of the prefix tree defined by ordered list $L_A^{<}$. Branches are pruned using Proposition 2.2.1 – if an itemset I fails the support itemset test in Definition 1.2.6(2) then it must be non-minimal and so the subtree with itemset I at the root can be pruned. The key advantage of the breadth-first approach is that the support row test can be performed extremely efficiently, as discussed in more detail in Section 2.4.1. Pseudo-code for the Kyiv algorithm is given in Algorithm 1.

In Algorithm 1 the collection of sets $\{P_i\}_{i=1}^{t}$ holds the vertices of level $k-1$ of the pruned prefix graph, and the vertices of level k are stored in $\{P_i'\}_{i=1}^{t'}$. Note that there is never any need to store more than two levels of the pruned prefix tree — we discuss these memory requirements in more detail below. The algorithm visits

Algorithm 1 Kyiv

```
 1: **Input**: dataset $A$, threshold $k_{max}$
 2: **Output**: all minimal unique $k$-itemsets, $k \le k_{max}$
 3: compute $I_A = I'_A \cup U_A \cup \delta_A$
 4: compute $L_A$ for chosen partition $(L_A, I'_A \setminus L_A)$
 5: print unique items in $\delta_A$                                  ▷ $k = 1$ case
 6: sort $L_A$ to obtain $L_A^<$
 7: $t \leftarrow 0$, $k \leftarrow 2$
 8: **foreach** $a \in L_A^<$ **do** $t \leftarrow t + 1$, $P_t \leftarrow \{a\}$
 9: **while** $k \le k_{max}$ **do**
10:    $t' \leftarrow 0$
11:    **foreach** $i \in \{1, \ldots, t-1\}$ **do**
12:       $I \leftarrow P_i$
13:       **foreach** $j \in \{i+1, \ldots, t\}$ **do**
14:          $J \leftarrow P_j$
15:          ▷ get the highest order items in $I$ and $J$
16:          $a \leftarrow \max(I),\ b \leftarrow \max(J)$
17:          **if** $I \setminus \{a\} \neq J \setminus \{b\}$ **then**
18:             break                    ▷ itemsets do not share a common prefix
19:          ▷ itemsets $I$ and $J$ differ exactly by one item now
20:          $W \leftarrow I \cup J$
21:          **if** $k > 2$ **then**
22:             ▷ support itemset test, Definition 1.2.6(2)
23:             **if** $\exists\, S \subset W, |S| = |W| - 1 : |R_S| \le 1$ **then**
24:                continue                    ▷ non-minimal, prune this branch
25:             **if** $k = k_{max}$ **then**
26:                ▷ Lemma 2.4.1 and Corollary 2.4.1 (Section 2.4.2)
27:                **if** $|R_I| + |R_J| > |R_{I\setminus\{a\}}| + 1$ **then** continue
28:                $c \leftarrow \max(J \setminus \{b\})$
29:                **if** $\min(|R_{I\setminus\{c\}}| - |R_I|, |R_{J\setminus\{c\}}| - |R_J|) + 1 < |R_{I\setminus\{c\}} \cap R_b|$ **then**
30:                   continue
31:          $R_W \leftarrow R_I \cap R_J$                              ▷ intersect rows
32:          **if** $|R_W| = 0$ or $|R_W| = \min(|R_I|, |R_J|)$ **then**
33:             continue                    ▷ skip absent and uniform itemsets
34:          **if** $|R_W| = 1$ **then**
35:             print $W$                         ▷ minimal unique itemset found
36:             **foreach** $w \in W$ **do**                ▷ apply Proposition 2.1.1
37:                **if** $\exists\, w' \in I'_A \setminus L_A : R_w = R_{w'}$ **then**
38:                   print $W \setminus \{w\} \cup \{w'\}$
39:          **else**                    ▷ need to store non-unique minimal itemset
40:             **if** $k < k_{max}$ **then**
41:                $t' \leftarrow t' + 1$, $P'_{t'} \leftarrow W$
42:    **foreach** $t \in \{1, \ldots, t'\}$ **do** $P_t \leftarrow P'_t$
43:    $k \leftarrow k + 1$, $t \leftarrow t'$
```

each vertex in level k and takes one of three actions: (i) finds that the vertex is a non-minimal itemset and so prunes it (it is not added to P' and its children are not traversed), (ii) finds that the vertex is a minimal unique itemset and so prints it (it is not added to P' and its children are not traversed), (iii) finds the vertex is non-unique and its children must be traversed.

In our implementation of Algorithm 1, we use a recursive data structure called Graph to hold the prefix tree levels. Graph stores an array of references to its children of type Graph and other useful data such as the rows associated with the current node. Each child is an item (v, j_a, R_a) and is identified by index value $mv + j_a$. Fast access to the children is achieved by use of a hash table, which is also stored among the properties of the Graph class.

2.4.1 Highly Efficient Support Itemset Testing

One of the key benefits of adopting a breadth-first approach in Algorithm 1 is that the computational cost of the support itemset test at line 23 can be reduced to essentially zero. This is because the itemsets $S \subset W$ of size $|S| = |W| - 1$, together with the associated row sets R_S, have already been pre-calculated and stored in data structure $\mathcal{P} := \{P_i\}_{i=1}^{t}$. Hence, evaluating whether there exists an S such that $|R_S| \leq 1$ simply involves lookups from $\mathcal{P}$, which can be carried out efficiently using an appropriate data structure for $\mathcal{P}$ such as a hash table.

Observe that acceleration of the support itemset test at line 23 is achieved in Algorithm 1 at the cost of increased RAM memory to store data structure $\mathcal{P}$. This cost is potentially significant, particularly in the middle of the prefix tree where the number of vertices in a level of the tree is largest. However, in view of the fact that the amount of RAM available is growing at a much faster rate than CPU clock speed, this trade-off between of increased memory consumption for a much reduced computational burden may be a favourable one.

2.4.2 Reducing Number of Row Intersections

The remaining computational bottleneck of Algorithm 1 is at line 31. We present performance measurements in Chapter 2.5 that confirm line 31 accounts for the vast majority of the execution time of Algorithm 1. However, we leave as future work the development of more efficient techniques for computing the intersection operation at line 31.

The potential exists to reduce the number of row intersections at the k_{max} level of the prefix tree using the following properties:

Lemma 2.4.1. *Let $I \subseteq I_A$ be an itemset and $a, b \in I_A$ any items in I_A. If*

$$|R_I \cap R_a| + |R_I \cap R_b| > |R_I| + 1 \tag{2.1}$$

then $I \cup \{a, b\}$ is not a unique itemset.

Proof. We proceed by contradiction. Suppose $|R_I \cap R_a| + |R_I \cap R_b| > |R_I| + 1$ and itemset $I \cup \{a, b\}$ is unique (so $|R_I \cap R_a \cap R_b| = 1$). By the distributivity of set intersection, $R_I \cap (R_a \cup R_b) = (R_I \cap R_a) \cup (R_I \cap R_b)$. Hence,

$$\begin{aligned}
&|R_I \cap (R_a \cup R_b)| \\
&\quad = |(R_I \cap R_a) \cup (R_I \cap R_b)| \\
&\quad = |R_I \cap R_a| + |R_I \cap R_b| - |(R_I \cap R_a) \cap (R_I \cap R_b)| \\
&\quad = |R_I \cap R_a| + |R_I \cap R_b| - |R_I \cap R_a \cap R_b|.
\end{aligned}$$

Now $|R_I| \geq |R_I \cap (R_a \cup R_b)|$ and by assumption $|R_I \cap R_a \cap R_b| = 1$. Hence, $|R_I| \geq |R_I \cap R_a| + |R_I \cap R_b| - 1$, yielding the desired contradiction. □

Corollary 2.4.1. *Let $a_1, \ldots, a_k \in I_A$ be any items from I_A, with $k > 2$. If*

$$\Gamma_0 > \min\{\Gamma_1, \Gamma_2\} + 1 \tag{2.2}$$

then $\{a_1, \ldots, a_k\}$ is not a unique itemset, where

$$\begin{aligned}
\Gamma_0 &:= |\cap_{i=1}^{k-3} R_{a_i} \cap R_{a_{k-1}} \cap R_{a_k}|, \\
\Gamma_1 &:= |\cap_{i=1}^{k-3} R_{a_i} \cap R_{a_{k-1}}| - |\cap_{i=1}^{k-3} R_{a_i} \cap R_{a_{k-2}} \cap R_{a_{k-1}}|, \\
\Gamma_2 &:= |\cap_{i=1}^{k-3} R_{a_i} \cap R_{a_k}| - |\cap_{i=1}^{k-3} R_{a_i} \cap R_{a_{k-2}} \cap R_{a_k}|.
\end{aligned}$$

Proof. There are two cases to consider.
Case (i): $\Gamma_0 > \min\{\Gamma_1, \Gamma_2\} + 1 = \Gamma_1 + 1$. Then,

$$\begin{aligned}
&|\cap_{i=1}^{k-3} R_{a_i} \cap R_{a_{k-1}} \cap R_{a_{k-2}}| + |\cap_{i=1}^{k-3} R_{a_i} \cap R_{a_{k-1}} \cap R_{a_k}| \\
&\qquad > |\cap_{i=1}^{k-3} R_{a_i} \cap R_{a_{k-1}}| + 1.
\end{aligned}$$

Let $I = \cup_{i=1}^{k-3} a_i \cup \{a_{k-1}\}$, $a = a_{k-2}$, $b = a_k$. By Lemma 2.4.1 $\{a_1, \ldots, a_k\}$ is not unique.

Case (ii): $\Gamma_0 > \min\{\Gamma_1, \Gamma_2\} + 1 = \Gamma_2 + 1$. Then,

$$\begin{aligned}
&| \cap_{i=1}^{k-3} R_{a_i} \cap R_{a_k} \cap R_{a_{k-2}}| + | \cap_{i=1}^{k-3} R_{a_i} \cap R_{a_k} \cap R_{a_{k-1}}| \\
&\quad > | \cap_{i=1}^{k-3} R_{a_i} \cap R_{a_k}| + 1.
\end{aligned}$$

Let $I = \cup_{i=1}^{k-3} a_i \cup \{a_k\}$, $a = a_{k-2}$, $b = a_{k-1}$. By Lemma 2.4.1 $\{a_1, \ldots, a_k\}$ is not unique. □

In the final iteration (when $k = k_{max}$) we can use Lemma 2.4.1 and Corollary 2.4.1 to test for uniqueness before carrying out the intersection at line 31. If either test concludes that the itemset is non-unique, then there is no need to perform the row intersection.

Example 2.4.1. *To illustrate the operation of Algorithm 1, suppose $k_{max} = 3$ and consider the dataset:*

$$A = \begin{bmatrix} * & * & * & 4 & * \\ 1 & 2 & * & 4 & * \\ 1 & 2 & 3 & 4 & * \\ 1 & 2 & 3 & 4 & 5 \\ 1 & * & 3 & * & 5 \\ * & 2 & 3 & * & 5 \\ * & * & * & * & 5 \end{bmatrix}, \textit{ where } * \textit{ denotes a unique item.}$$

The set δ_A contains the unique items marked by $*$. There are no uniform items, so $U_A = \emptyset$. There exists single partition of I'_A — $(L_A, \emptyset)$, where it can be verified that

$$\begin{aligned}
L_A^< = \{&(1, 1, \{2, 3, 4, 5\}), (2, 2, \{2, 3, 4, 6\}), (3, 3, \{3, 4, 5, 6\}), \\
&(4, 4, \{1, 2, 3, 4\}), (5, 5, \{4, 5, 6, 7\})\} := \{a, b, c, d, e\}.
\end{aligned}$$

The prefix tree of $L_A^<$ is shown schematically in Figure 2.1. After line 8 is executed ($P_1 = \{a\}, P_2 = \{b\}, P_3 = \{c\}, P_4 = \{d\}, P_5 = \{e\}$) and the first level of the prefix tree is built. The first iteration of the main loop at line 9 (when $k = 2 < 3 = k_{max}$ and $t = 5$) is reproduced step-by-step below. Here, $1 \le i \le 4 = t - 1, i < j \le t$ and for each (I, J) the highest order items are the items contained in I and J (which never share a common prefix). The condition at line 21 is false and there are no absent or uniform itemsets ($0 < |R_W| < \min(|R_I|, |R_J|)$ for each (I, J))

after intersection at line 31:

$i = 1 : I = P_1 = \{a\}$

$\quad j = 2 \leq 5 = t : J = P_2 = \{b\},\ a = \max(I),\ b = \max(J),\ W = \{a, b\}$

$\quad\quad R_W = \{2, 3, 4\}$

$\quad\quad 4 > |R_W| > 1 \textit{ and } k = 2 < 3 = k_{max} \Rightarrow t' = 1, P'_1 = \{a, b\}$

$\quad j = 3 \leq 5 = t : J = P_3 = \{c\},\ a = \max(I),\ c = \max(J),\ W = \{a, c\}$

$\quad\quad R_W = \{3, 4, 5\}$

$\quad\quad 4 > |R_W| > 1 \textit{ and } k = 2 < 3 = k_{max} \Rightarrow t' = 2, P'_2 = \{a, c\}$

$\quad j = 4 \leq 5 = t : J = P_4 = \{d\},\ a = \max(I),\ d = \max(J),\ W = \{a, d\}$

$\quad\quad R_W = \{2, 3, 4\}$

$\quad\quad 4 > |R_W| > 1 \textit{ and } k = 2 < 3 = k_{max} \Rightarrow t' = 3, P'_3 = \{a, d\}$

$\quad j = 5 \leq 5 = t : J = P_5 = \{e\},\ a = \max(I),\ e = \max(J),\ W = \{a, e\}$

$\quad\quad R_W = \{4, 5\}$

$\quad\quad 4 > |R_W| > 1 \textit{ and } k = 2 < 3 = k_{max} \Rightarrow t' = 4, P'_4 = \{a, e\}$

$i = 2 : I = P_2 = \{b\}$

$\quad j = 3 \leq 5 = t : J = P_3 = \{c\},\ b = \max(I),\ c = \max(J),\ W = \{b, c\}$

$\quad\quad R_W = \{3, 4, 6\}$

$\quad\quad 4 > |R_W| > 1 \textit{ and } k = 2 < 3 = k_{max} \Rightarrow t' = 5, P'_5 = \{b, c\}$

$\quad j = 4 \leq 5 = t : J = P_4 = \{d\},\ b = \max(I),\ d = \max(J),\ W = \{b, d\}$

$\quad\quad R_W = \{2, 3, 4\}$

$\quad\quad 4 > |R_W| > 1 \textit{ and } k = 2 < 3 = k_{max} \Rightarrow t' = 6, P'_6 = \{b, d\}$

$\quad j = 5 \leq 5 = t : J = P_5 = \{e\},\ b = \max(I),\ e = \max(J),\ W = \{b, e\}$

$\quad\quad R_W = \{4, 6\}$

$\quad\quad 4 > |R_W| > 1 \textit{ and } k = 2 < 3 = k_{max} \Rightarrow t' = 7, P'_7 = \{b, e\}$

$i = 3 : I = P_3 = \{c\}$

$\quad j = 4 \leq 5 = t : J = P_4 = \{d\},\ c = \max(I),\ d = \max(J),\ W = \{c, d\}$

$\quad\quad R_W = \{3, 4\}$

$\quad\quad 4 > |R_W| > 1 \textit{ and } k = 2 < 3 = k_{max} \Rightarrow t' = 8, P'_8 = \{c, d\}$

$\quad j = 5 \leq 5 = t : J = P_5 = \{e\},\ c = \max(I),\ e = \max(J),\ W = \{c, e\}$

$\quad\quad R_W = \{4, 5, 6\}$

$\quad\quad 4 > |R_W| > 1 \textit{ and } k = 2 < 3 = k_{max} \Rightarrow t' = 9, P'_9 = \{c, e\}$

$i = 4 : I = P_4 = \{d\}$

$j = 5 \leq 5 = t : J = P_5 = \{e\},\ d = \max(I),\ e = \max(J),\ W = \{d, e\}$

$R_W = \{4\} \Rightarrow$ *print* $\{d, e\}$ *as minimal unique* 2*-itemset*

Proposition 2.1.1 is not applied because $I'_A \setminus L_A = \emptyset$.

The second level of the prefix tree is now built: $P_1 = \{a, b\}, P_2 = \{a, c\}, P_3 = \{a, d\}, P_4 = \{a, e\}, P_5 = \{b, c\}, P_6 = \{b, d\}, P_7 = \{b, e\}, P_8 = \{c, d\}, P_9 = \{c, e\}$.

The second iteration of the main loop (when $k = 3 = k_{max}$ *and* $t = 9$*) is reproduced step-by-step below. Here,* $1 \leq i \leq 8 = t - 1, i < j \leq t$ *and for each* (I, J) *there are no absent or uniform itemsets after intersection at line 31:*

$i = 1 : I = P_1 = \{a, b\}$

$j = 2 \leq 9 = t : J = P_2 = \{a, c\},\ b = \max(I),\ c = \max(J)$

$I \setminus \max(I) = \{a\} = J \setminus \max(J) : W = \{a, b, c\}$

$k = 3 > 2 :$

support itemset test, Definition 1.2.6(2) at line 23 fails:

there is no $S \subset W, |S| = |W| - 1 : |R_S| \leq 1$

Lemma 2.4.1 (Section 2.4.2) succeeds:

$|\{2, 3, 4\}| + |\{3, 4, 5\}| = 6 > 5 = |\{2, 3, 4, 5\}| + 1$

Algorithm continues to the next J

$j = 3 \leq 9 = t : J = P_3 = \{a, d\},\ b = \max(I),\ d = \max(J)$

$I \setminus \max(I) = \{a\} = J \setminus \max(J) : W = \{a, b, d\}$

$k = 3 > 2 :$

support itemset test, Definition 1.2.6(2) at line 23 fails:

there is no $S \subset W, |S| = |W| - 1 : |R_S| \leq 1$

Lemma 2.4.1 (Section 2.4.2) succeeds:

$|\{2, 3, 4\}| + |\{2, 3, 4\}| = 6 > 5 = |\{2, 3, 4, 5\}| + 1$

Algorithm continues to the next J

$j = 4 \leq 9 = t : J = P_4 = \{a, e\},\ b = \max(I),\ e = \max(J)$

$I \setminus \max(I) = \{a\} = J \setminus \max(J) : W = \{a, b, e\}$

$k = 3 > 2 :$

support itemset test, Definition 1.2.6(2) at line 23 fails:

there is no $S \subset W, |S| = |W| - 1 : |R_S| \leq 1$

Lemma 2.4.1 (Section 2.4.2) fails:

$|\{2,3,4\}| + |\{4,5\}| = 5 \not> 5 = |\{2,3,4,5\}| + 1$

Corollary 2.4.1 (Section 2.4.2) fails:

$\min(|\{2,3,4,6\}| - |\{2,3,4\}|, |\{4,5,6,7\}| - |\{4,5\}|)+$
$1 = 2 \not< 2 = |\{4,6\}|$

$R_W = \{4\} \Rightarrow$ *print* $\{a,b,e\}$ *as minimal unique* 3*-itemset*

Proposition 2.1.1 is not applied because $I'_A \setminus L_A = \emptyset$

$j = 5 \leq 9 = t : J = P_5 = \{b,c\},\ b = \max(I),\ c = \max(J)$

$I \setminus \max(I) = \{a\} \neq \{b\} = J \setminus \max(J)$

Algorithm continues to the next I

$i = 2 : I = P_2 = \{a,c\}$

$j = 3 \leq 9 = t : J = P_3 = \{a,d\},\ c = \max(I),\ d = \max(J)$

$I \setminus \max(I) = \{a\} = J \setminus \max(J) : W = \{a,c,d\}$

$k = 3 > 2 :$

support itemset test, Definition 1.2.6(2) at line 23 fails:

there is no $S \subset W, |S| = |W| - 1 : |R_S| \leq 1$

Lemma 2.4.1 (Section 2.4.2) succeeds:

$|\{3,4,5\}| + |\{2,3,4\}| = 6 > 5 = |\{2,3,4,5\}| + 1$

Algorithm continues to the next J

$j = 4 \leq 9 = t : J = P_4 = \{a,e\},\ c = \max(I),\ e = \max(J)$

$I \setminus \max(I) = \{a\} = J \setminus \max(J) : W = \{a,c,e\}$

$k = 3 > 2 :$

support itemset test, Definition 1.2.6(2) at line 23 fails:

there is no $S \subset W, |S| = |W| - 1 : |R_S| \leq 1$

Lemma 2.4.1 (Section 2.4.2) fails:

$|\{3,4,5\}| + |\{4,5\}| = 5 \not> 5 = |\{2,3,4,5\}| + 1$

Corollary 2.4.1 (Section 2.4.2) succeeds:

$\min(|\{3,4,5,6\}| - |\{3,4,5\}|, |\{4,5,6,7\}| - |\{4,5\}|)+$
$1 = 2 < 3 = |\{4,5,6\}|$

Algorithm continues to the next J

$j = 5 \leq 9 = t : J = P_5 = \{b,c\},\ c = \max(I),\ c = \max(J)$

$I \setminus \max(I) = \{a\} \neq \{b\} = J \setminus \max(J)$

Algorithm continues to the next I

```
$i = 3 : I = P_3 = \{a, d\}$
    $j = 4 \leq 9 = t : J = P_4 = \{a, e\},\ d = \max(I),\ e = \max(J)$
        $I \setminus \max(I) = \{a\} = J \setminus \max(J) : W = \{a, d, e\}$
        $k = 3 > 2 :$
            support itemset test, Definition 1.2.6(2) at line 23 succeeds:
            $\exists\, S = \{d, e\} \subset W, |S| = |W| - 1 : |R_S| = |\{4\}| = 1 \leq 1$
            Algorithm continues to the next J
    $j = 5 \leq 9 = t : J = P_5 = \{b, c\},\ d = \max(I),\ c = \max(J)$
        $I \setminus \max(I) = \{a\} \neq \{b\} = J \setminus \max(J)$
        Algorithm continues to the next I
$i = 4 : I = P_4 = \{a, e\}$
    $j = 5 \leq 9 = t : J = P_5 = \{b, c\},\ e = \max(I),\ c = \max(J)$
        $I \setminus \max(I) = \{a\} \neq \{b\} = J \setminus \max(J)$
        Algorithm continues to the next I
$i = 5 : I = P_5 = \{b, c\}$
    $j = 6 \leq 9 = t : J = P_6 = \{b, d\},\ c = \max(I),\ d = \max(J)$
        $I \setminus \max(I) = \{b\} = J \setminus \max(J) : W = \{b, c, d\}$
        $k = 3 > 2 :$
            support itemset test, Definition 1.2.6(2) at line 23 fails:
            there is no $S \subset W, |S| = |W| - 1 : |R_S| \leq 1$
            Lemma 2.4.1 (Section 2.4.2) succeeds:
            $|\{3, 4, 6\}| + |\{2, 3, 4\}| = 6 > 3 = |\{3, 4\}| + 1$
            Algorithm continues to the next J
    $j = 7 \leq 9 = t : J = P_7 = \{b, e\},\ c = \max(I),\ e = \max(J)$
        $I \setminus \max(I) = \{b\} = J \setminus \max(J) : W = \{b, c, e\}$
        $k = 3 > 2 :$
            support itemset test, Definition 1.2.6(2) at line 23 fails:
            there is no $S \subset W, |S| = |W| - 1 : |R_S| \leq 1$
            Lemma 2.4.1 (Section 2.4.2) fails:
            $|\{3, 4, 6\}| + |\{4, 6\}| = 5 \not> 5 = |\{2, 3, 4, 6\}| + 1$
            Corollary 2.4.1 (Section 2.4.2) succeeds:
            $\min(|\{3, 4, 5, 6\}| - |\{3, 4, 6\}|, |\{4, 5, 6, 7\}| - |\{4, 6\}|) +$
```

$$1 = 2 < 3 = |\{4, 5, 6\}|$$

Algorithm continues to the next J

$j = 8 \leq 9 = t : J = P_8 = \{c, d\},\ c = \max(I),\ d = \max(J)$

$I \setminus \max(I) = \{b\} \neq \{c\} = J \setminus \max(J)$

Algorithm continues to the next I

$i = 6 : I = P_6 = \{b, d\}$

$j = 7 \leq 9 = t : J = P_7 = \{b, e\},\ d = \max(I),\ e = \max(J)$

$I \setminus \max(I) = \{b\} = J \setminus \max(J) : W = \{b, d, e\}$

$k = 3 > 2 :$

support itemset test, Definition 1.2.6(2) at line 23 succeeds:

$\exists\, S = \{d, e\} \subset W, |S| = |W| - 1 : |R_S| = |\{4\}| = 1 \leq 1$

Algorithm continues to the next J

$j = 8 \leq 9 = t : J = P_8 = \{c, d\},\ d = \max(I),\ d = \max(J)$

$I \setminus \max(I) = \{b\} \neq \{c\} = J \setminus \max(J)$

Algorithm continues to the next I

$i = 7 : I = P_7 = \{b, e\}$

$j = 8 \leq 9 = t : J = P_8 = \{c, d\},\ d = \max(I),\ d = \max(J)$

$I \setminus \max(I) = \{b\} \neq \{c\} = J \setminus \max(J)$

Algorithm continues to the next I

$i = 8 : I = P_8 = \{c, d\}$

$j = 9 \leq 9 = t : J = P_9 = \{c, e\},\ d = \max(I),\ e = \max(J)$

$I \setminus \max(I) = \{c\} = J \setminus \max(J) : W = \{c, d, e\}$

$k = 3 > 2 :$

support itemset test, Definition 1.2.6(2) at line 23 succeeds:

$\exists\, S = \{d, e\} \subset W, |S| = |W| - 1 : |R_S| = |\{4\}| = 1 \leq 1$

Algorithm ends.

At the ultimate level k_{max}, the support itemset test for minimality (line 23), Lemma 2.4.1 (line 27) and Corollary 2.4.1 (line 29) are applied in that order to pairs of 2-itemsets from P which share a common prefix. Pairs ($\{a, d\}$, $\{a, e\}$), ($\{b, d\}$, $\{b, e\}$), ($\{c, d\}$, $\{c, e\}$) are pruned by the support itemset test. Pairs ($\{a, b\}$, $\{a, c\}$), ($\{a, b\}$, $\{a, d\}$), ($\{a, c\}$, $\{a, d\}$), ($\{b, c\}$, $\{b, d\}$) are pruned by the lemma. Pairs ($\{a, c\}$, $\{a, e\}$), ($\{b, c\}$, $\{b, e\}$) are pruned by the corollary. Leaving *only* $(\{a,b\},\ \{a,e\})$ *as minimal unique itemset.* □

2.4.3 Correctness

Theorem 2.4.1. *Algorithm 1 terminates in finite time and finds all minimal unique itemsets of* I_A *up to size* k_{max}*.*

Proof. Pre-processing from the beginning to the main loop (line 9) is done in finite time: to compute I_A and L_A algorithm goes through the A elements and counts their frequencies while the size of A is finite ($n, m < +\infty$); printing δ_A, sorting L_A and iterating $|L_A^<|$ times the loop at line 8 all take finite time as $|\delta_A|, |L_A| = |L_A^<| < +\infty$. The search space of the algorithm is the prefix tree which is finite as I_A is finite. If there is no pruning then Algorithm 1 goes through every branch of maximum length k_{max} of the tree, otherwise it processes even fewer branches. It takes finite time to process a single branch (that is: navigate it, intersect itemset rows of finite size and either print (Proposition 2.1.1 takes finite time because $|W|, |I_A' \setminus L_A| < +\infty$) or store the appropriate itemset). Consequently the algorithm terminates in finite time processing all the itemsets of maximum size k_{max} that have not been thrown out by the support itemset test (line 23), Lemma 2.4.1 (line 27) and Corollary 2.4.1 (line 29).

Suppose there is a minimal unique itemset $I \in 2^{I_A}$ that is not found by the algorithm. Proposition 2.1.2 means that the set of all unique and minimal itemsets $\mathcal{I}_A \subset 2^{I_A}$ can be described by any chosen partition $(L_A, \bar{L}_A)$. Thus, either I contains item which does not belong to L_A or $|I| > k_{max}$. The former is impossible while the latter does not contradict the theorem. □

2.4.4 Parallelisation

Algorithm 1 can be readily parallelised using shared-memory threads. Namely, at level k within the prefix tree assign all vertices sharing the same parent at level $k-1$ within the prefix tree to the same thread and then in each thread execute the loop starting at line 13 in Algorithm 1. The shared memory allows each thread access to the prefix tree information stored in P_j, $j \in \{i+1, \cdots, t\}$, but there is otherwise no need for inter-thread communication.

When the number of available threads is less than the number of parent vertices at level $k-1$ in the prefix tree, work must be allocated amongst the threads. As already discussed, the work associated with each parent vertex is dominated by the number of row intersections to be carried out. This number can be accurately estimated based on the number of children of the parent vertex, and so the work associated with each parent vertex estimated in advance. Using these work estimates, load-balanced scheduling of work amongst the threads can then be efficiently realised. As discussed in more detail in Section 2.5, in this way we can ensure that the running time of all threads is similar thereby enhancing the performance gain from parallelisation — we note that imbalanced thread run times is known to be a key bottleneck in the parallelisation of state-of-the-art depth-first approaches such as SUDA2 and MINIT [HMM$^+$09].

Example 2.4.2. *Recall Example 2.4.1. Let* $t = 3$ *be the number of threads. When* $k = 2$*, Algorithm 1 allocates jobs between the* 3 *threads: first an empty array* T *of size* t *is created; then for each item in* $L_A^<$ *the number of higher order items is stored in* T *at the cell which has the minimum value (if there are several such cells, the left-most is chosen). As soon as* T *is filled in, all threads start computation. In our example* $T = \{4, 3, 3\}$ *and the first thread is assigned itemsets,* $\{a, b\}$, $\{a, c\}$, $\{a, d\}$, $\{a, e\}$*, the second* $\{b, c\}$, $\{b, d\}$, $\{b, e\}$ *and the third* $\{c, d\}$, $\{c, e\}$, $\{d, e\}$*. Row intersection of each ordered pair reveals the unique* 2*-itemsets and these itemsets are stored in* P'*:* $\{a, b\}$, $\{a, c\}$, $\{a, d\}$, $\{a, e\}$, $\{b, c\}$, $\{b, d\}$, $\{b, e\}$, $\{c, d\}$ *and* $\{c, e\}$*; at the next iteration they will be copied into* P *for the* $k = 3$ *analysis. Only* $\{d, e\}$ *will be printed out as unique and minimal.*

When $k = 3$ *(the ultimate level* k_{max}*),* $T = \{6, 3, 1\}$ *and the first thread is assigned itemsets* $(\{a, b\}, \{a, c\})$, $(\{a, b\}, \{a, d\})$, $(\{a, b\}, \{a, e\})$, $(\{a, c\}, \{a, d\})$, $(\{a, c\}, \{a, e\})$, $(\{a, d\}, \{a, e\})$*, the second* $(\{b, c\}, \{b, d\})$, $(\{b, c\}, \{b, e\})$, $(\{b, d\}, \{b, e\})$ *and the third* $(\{c, d\}, \{c, e\})$*. As in Example 2.4.1, the support itemset test, Lemma 2.4.1 and Corollary 2.4.1 eliminates all pairs inside the threads except for* $(\{a, b\}, \{a, e\})$. □

2.5 Experimental Results

If not otherwise stated, all experiments in this section were carried out using ascending itemlist order, Lemma 2.4.1 and Corollary 2.4.1.

2.5.1 Hardware and Software Setup

We implemented Algorithm 1 in Java (version 1.7.0_25) using the hppc (version 0.5.2) library, which can be found at `http://labs.carrotsearch.com/hppc.html`. For comparison with the serial version of Algorithm 1, we also implemented a state-of-the-art algorithm MINIT [HM07] in Java (using the C++ implementation kindly provided by the developers of MINIT) and used the C++ implementation of the MIWI algorithm [CG13], kindly provided by its developers.

For testing we used an Amazon cr1.8xlarge instance with an Intel Xeon CPU E5-2670 0 @ 2.60GHz 32 processor (up to 32 hyperthreads), 244Gb of memory, 64-bit Linux operating system (kernel version 3.4.62-53.42. amzn1.x86_64 of Red Hat 4.6.3-2 Linux distribution (Amazon Linux AMI release 2013.09)).

2.5.2 Domain-Agnostic Performance

2.5.2.1 Randomised Datasets

We begin by investigating performance in a domain-agnostic manner using randomised datasets. Each randomised dataset consists of $50,000$ rows with each row having 25 columns. For each column, the size D of the domain of element values is selected i.i.d. uniformly at random from the set $\{10, \cdots, 100\}$. The elements within each column are then selected i.i.d. uniformly at random from domain $\{1, \cdots, D\}$. On average, for these datasets L_A contained 1352 items.

2.5.2.2 Execution Time

Figure 2.2 shows the measured distribution of execution times for Algorithm 1 over 50 randomised datasets when $k_{max} = 5$. It can be seen that the execution times are relatively tightly bunched around the mean value of 280 seconds. Also shown in Figure 2.2 is the corresponding time expended on calculating row intersections at line 31 of Algorithm 1. The mean intersection time is 190 seconds, so 68% of the execution time is expended on row intersections, confirming that these are indeed the primary bottleneck in Algorithm 1. Note that the fraction of execution time expended on row intersections depends on k_{max} and tends to increase as k_{max} decreases *e.g.* when $k_{max} = 3$ row intersections absorb 80% of the execution time.

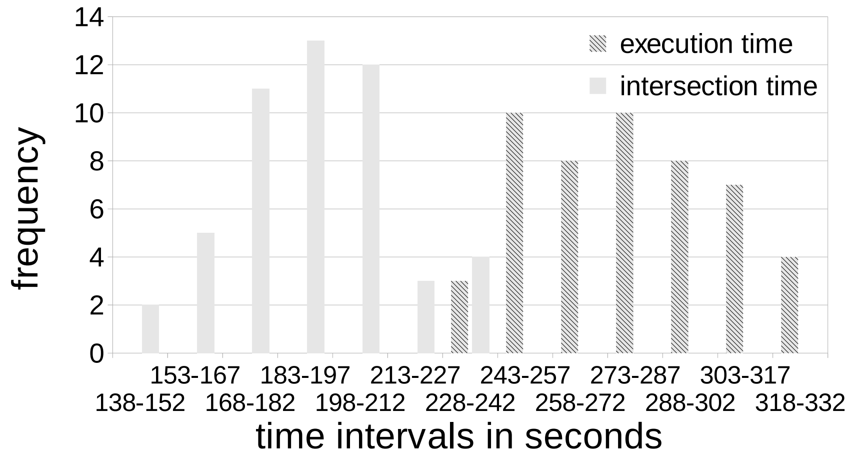

Figure 2.2: Distribution of execution and intersection time for randomised datasets, $k_{max} = 5$.

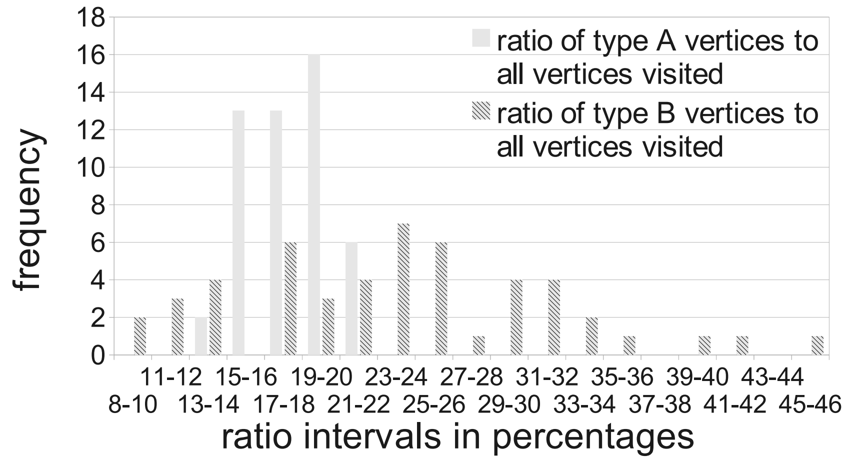

Figure 2.3: Distribution of prefix tree vertices traversed for randomised datasets, $k_{max} = 5$.

2.5.2.3 Prefix tree pruning

Algorithm 1 carries out online pruning of the prefix tree so as to avoid walking the full prefix tree. Importantly, it also tries to avoid carrying out unnecessary row intersections. We can evaluate the efficiency of the latter by distinguishing between three types of vertices visited: vertices that correspond to minimal unique itemsets (A), vertices which are visited but for which a row intersection is not performed (B) and the rest of the vertices visited (C). Figure 2.3 shows the distribution of ratios of the number of vertices of types A and B to the total number of prefix tree vertices visited by the algorithm over 50 randomised datasets when $k_{max} = 5$. On average 17.5% of vertices visited are type A vertices and 23% type

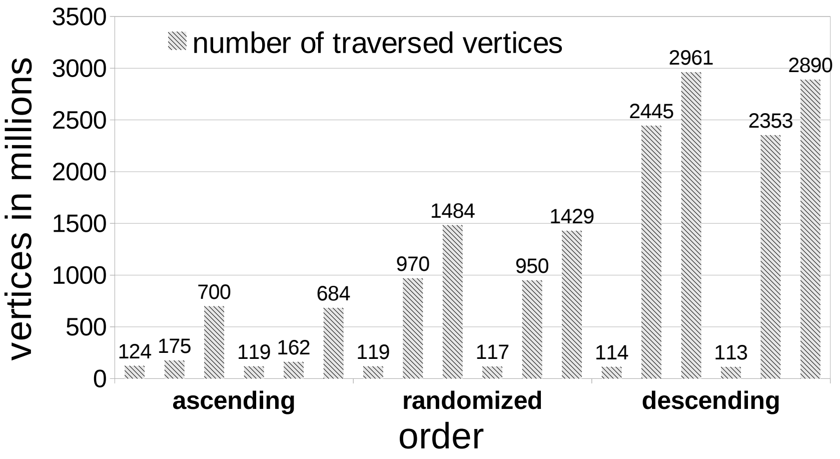

Figure 2.4: Prefix tree vertices traversed vs ordering used for L_A, average over 10 randomised datasets, $k_{max} = 5$. For each ordering 6 values are shown: in the first three Lemma 2.4.1 and Corollary 2.4.1 are used, in the second three these are not used; in each group of three values the first value represents the number of vertices of type A, the second the number of vertices of type B and the third the total number of vertices traversed (that is of type A, B and C).

B vertices, although sometimes up to 45% of vertices visited are of type B.

2.5.2.4 Impact of Ordering Used for L_A

As already noted in Section 2.2, the ordering used to sort set L_A to obtain $L_A^<$ can be expected to have an impact on the amount of pruning of the prefix tree achieved, and so on the execution time of Algorithm 1. To investigate this further, we collected performance measurements for three different choices of ordering: (i) ascending order, (ii) descending order (iii) random order (*i.e.* we draw a permutation uniformly at random from the set of permutations mapping from $\{1, \cdots, |L_A|\}$ to itself and apply this permutation to obtain $L_A^<$).

Figure 2.4 plots the numbers of prefix tree vertices of types A, B and C visited by Algorithm 1 vs the ordering of L_A used. In this figure data is presented for each of the three orderings (ascending, randomised, descending) and for when Lemma 2.4.1/Corollary 2.4.1 are used or not. That is, 6 experiment variants are compared.

It can be seen that use of ascending order significantly reduces the total number of vertices visited, yielding a reduction of roughly a factor of 2 compared to use of a randomised ordering and a factor of 4 compared to descending order. The number of type A vertices visited is, as expected, essentially constant across the

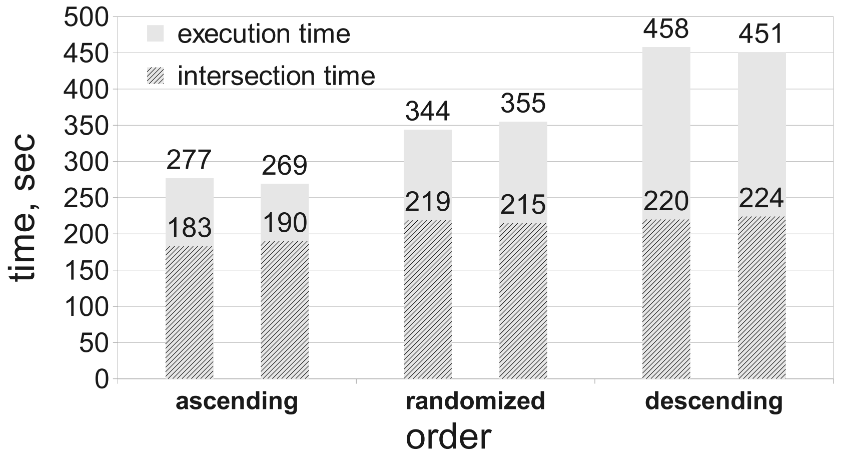

Figure 2.5: Intersection and execution time vs ordering used for L_A, average over 10 randomised datasets, $k_{max} = 5$ (in the left bar Lemma 2.4.1/Corollary 2.4.1 are used, in the right bar they are not used).

tests. However, the number of type B vertices changes significantly and varies such that the number of vertices of type C remains roughly constant. Observe that while use of Lemma 2.4.1 and Corollary 2.4.1 has little impact on performance in these tests. We will revisit this in Section 2.5.3.2 where we find that they can speed the runtime up by more than 50%.

Figure 2.5 plots the corresponding intersection and execution time vs the ordering of L_A used. It can be seen that the execution time is more sensitive to the ordering than the intersection time. When combined with Figure 2.4 this allows us to conclude that it is the number of type B vertices that varies strongly with ordering (the number of type A and type C vertices stays nearly constant) and that ascending order reduces execution time primarily by reducing the number of type B vertices i.e. by more effective pruning of the search tree which reduces the overall number of vertices visited.

2.5.2.5 Impact of Dataset Parameters

To investigate the scaling behaviour of Algorithm 1 to larger datasets we generated a randomised dataset with $1,000,000$ rows and 40 columns yielding an itemlist of size $2,179$.

Taking the first n rows, Figure 2.6 plots the execution time of Algorithm 1 versus n for $k_{max} = 3$. It can be seen that the execution time is approximately linear in n, and so scales well to larger datasets. Although not plotted, memory usage

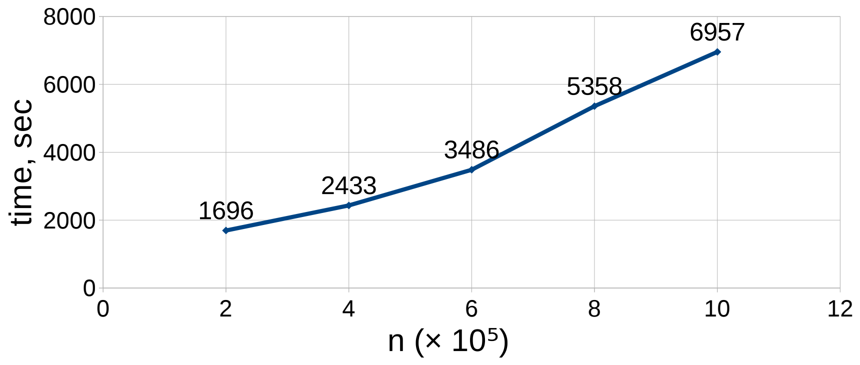

Figure 2.6: Execution time vs number of rows n for a randomised dataset with $m = 40$ columns, $k_{max} = 3$.

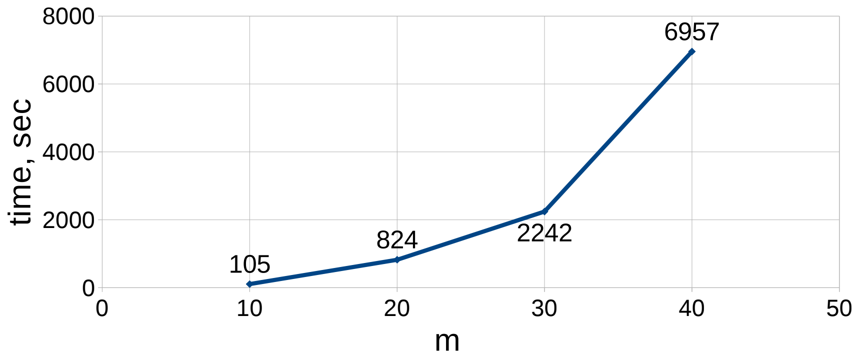

Figure 2.7: Execution time vs number of columns m for a randomised dataset with $n = 1,000,000$ rows, $k_{max} = 3$.

also increased only gradually from 5.6Gb when $n = 200,000$ to 6Gb when $n = 1,000,000$.

Taking the first m columns of the dataset, Figure 2.7 plots the execution time versus m for $k_{max} = 3$. It can be seen that the execution time is approximately exponential in m, and so the algorithm scales less well to datasets with a large number of columns (the size of corresponding itemlist increased from 520 to $2,179$). Note that the memory usage also increases quite rapidly with m, from 0.9Gb when $m = 10$ to 6Gb when $m = 40$.

2.5.3 Domain-Specific Performance

2.5.3.1 Datasets

In this section we present performance measurements for four domain-specific datasets:

1. The Connect dataset is available from `http://fimi.ua.ac.be/data` and contains all legal 8-ply positions in the game of connect-4 in which neither player has won yet, and in which the next move is not forced. There are $67,557$ rows, 43 columns (one for each of the 42 connect-4 squares together with an outcome column - win, draw or lose) and 129 items. It was one of the most computationally challenging datasets for which MINIT was evaluated in [HM07].

2. The Pumsb dataset is census data for population and housing from the PUMS (Public Use Microdata Sample). This dataset is available from `http://fimi.ua.ac.be/data`. There are $49,046$ rows, 74 columns and $1,958$ items.

3. The Poker dataset is available from `http://archive.ics.uci.edu/ml/datasets.html`. Each record is an example of a hand consisting of five playing cards drawn from a standard deck of 52 cards. Each card is described using two attributes (suit and rank), for a total of 10 predictive attributes. There is one Class attribute that describes the "Poker Hand". We removed the last attribute to form a new dataset with $1,000,000$ rows, 10 columns and 117 items.

4. The USCensus1990 dataset, available from `http://archive.ics.uci.edu/ml/datasets.html`, was collected as part of the 1990 census. We considered a subset of this dataset consisting of the first $200,000$ rows and 68 columns, which contained $8,009$ items.

2.5.3.2 Execution Time vs k_{max}

All measurements in the current section are averaged over three consecutive runs of each algorithm.

Figures 2.8, 2.9, 2.10 and 2.11 show the measured execution times of Algorithm 1, MINIT and MIWI Miner measured for the Connect, Pumsb, Poker and USCensus1990 datasets vs k_{max}.

It can be seen that Algorithm 1 consistently outperforms MINIT for all values of k_{max} and for all datasets. For the Connect dataset it can be seen that Algorithm 1 achieves runtimes between 3 and 9 times faster than MINIT. For the Pumsb dataset Algorithm 1 is between 2 and 11 times faster. For the Poker dataset Algorithm 1 is between 2 and 33 times faster (for $k_{max} = 7$ MINIT was

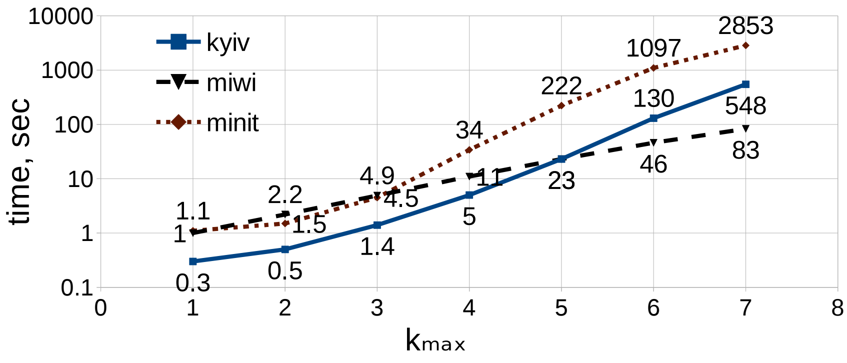

Figure 2.8: Execution time vs k_{max} for Connect dataset.

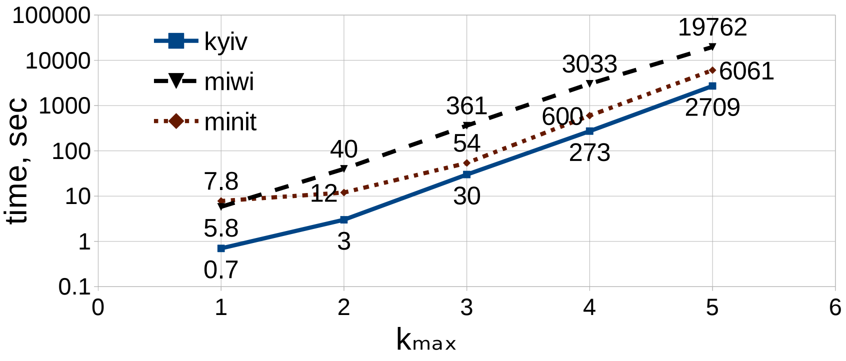

Figure 2.9: Execution time vs k_{max} for Pumsb dataset.

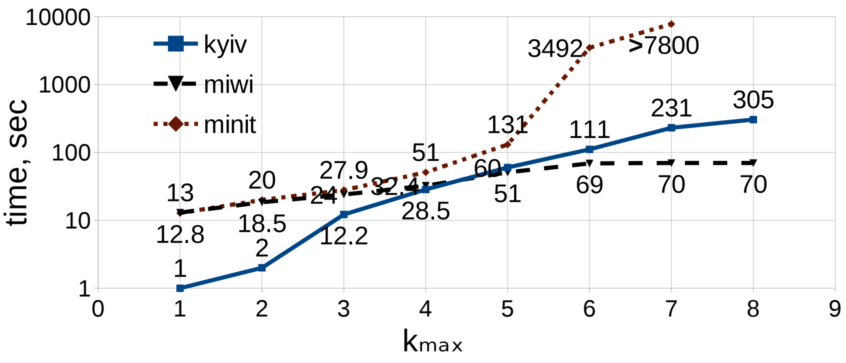

Figure 2.10: Execution time vs k_{max} for Poker dataset.

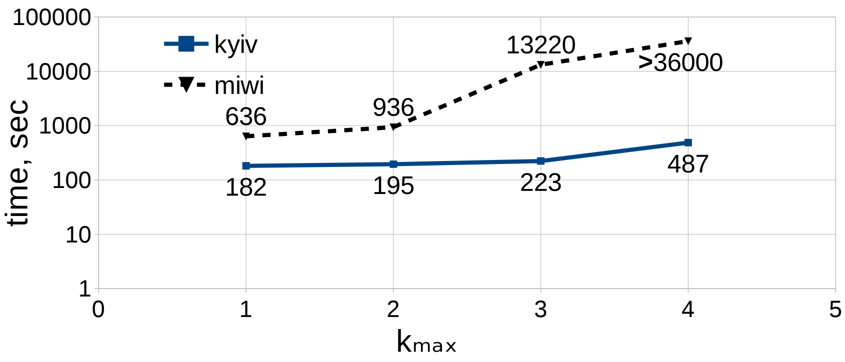

Figure 2.11: Execution time vs k_{max} for USCensus1990 dataset.

terminated after $7,800$ seconds without completing). Data is not shown for the USCensus1990 dataset since both the C++ and Java implementations of MINIT ran out of memory on this demanding dataset (which has $8,009$ items).

For the Connect and Poker datasets MIWI is $2-7$ times faster than Algorithm 1 when $k_{max} > 4$, but MIWI is $2-9$ times slower than Algorithm 1 when $k_{max} \leq 4$. MIWI is also $5-13$ times slower than Algorithm 1 for the Pumsb dataset for all values of k_{max} (and also slower than MINIT for this dataset). For the demanding USCensus1990 dataset MIWI's execution time is 220 minutes when $k_{max} = 3$, and it did not complete within a reasonable time for $k_{max} = 4$. In comparison, Algorithm 1 finds minimal sample uniques for $k_{max} = 4$ in 8 minutes while for $k_{max} = 3$ the execution time reduces to 3 minutes.

Revisiting the order analysis in Section 2.5.2.4, we point out that when Algorithm 1 is run without using Lemma 2.4.1 and Corollary 2.4.1 then the execution time rises to 269 seconds (from 130 seconds) for the Connect dataset, $k_{max} = 6$ and to 410 (from 273 seconds) seconds for the Pumsb dataset, $k_{max} = 4$ for example.

2.5.3.3 Memory Usage

Algorithm 1 intentionally trades increased memory for faster execution times via its use of a breadth-first approach. This is reasonable in view of the favourable scaling of memory size vs CPU speed on modern hardware. Figure 2.12 shows the memory consumption of Algorithm 1 for Connect, Pumsb, Poker and USCensus1990 datasets vs k_{max}. These plots indicate the maximum memory needed during algorithm execution and so this amount of memory ensures the fastest execution time since garbage collection is not required. For smaller amounts of memory the algorithm is observed to become somewhat slower as the Java Virtual

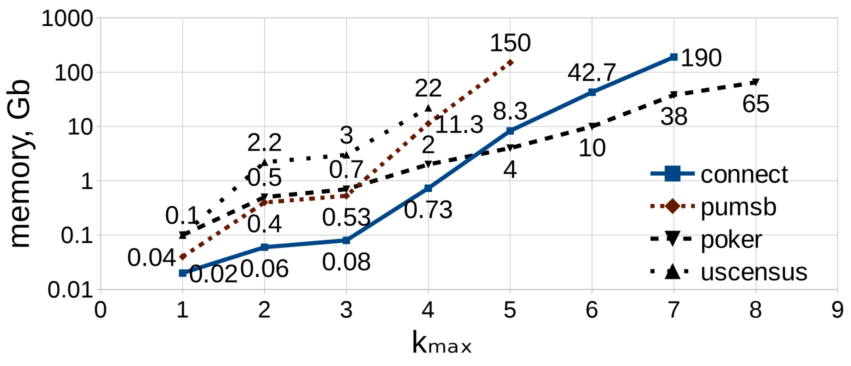

Figure 2.12: Memory consumption of Algorithm 1 vs k_{max}.

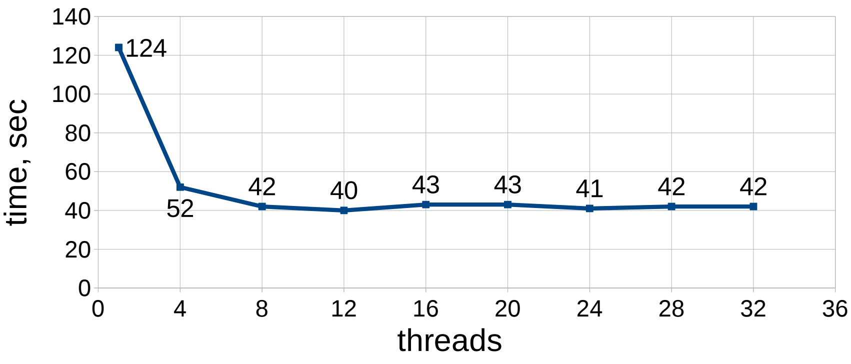

Figure 2.13: Parallel algorithm execution time vs number of threads for Connect, $k_{max} = 6$.

Machine need to start garbage collection.

The memory requirement is dominated by storage of itemset rows to perform intersection. When $1 < k < k_{max}$, two levels of the prefix tree must be stored, but when $k = k_{max}$ (last level), then only one level needs to be stored (for example, the 190Gb in Figure 2.12 is mostly occupied by the 6-itemset rows). Note that there is a level in the prefix tree that requires the largest amount of memory, a sort of equator. Beyond this value Algorithm 1 can compute all minimal unique itemsets without additional memory.

2.5.4 Parallel Algorithm Performance

Figures 2.13 and 2.14 show execution time versus the number of threads used for the Connect, $k_{max} = 6$ and Pumsb, $k_{max} = 5$ datasets respectively. It can be seen that at around 8 threads the performance saturates and additional threads yield little further performance gain.

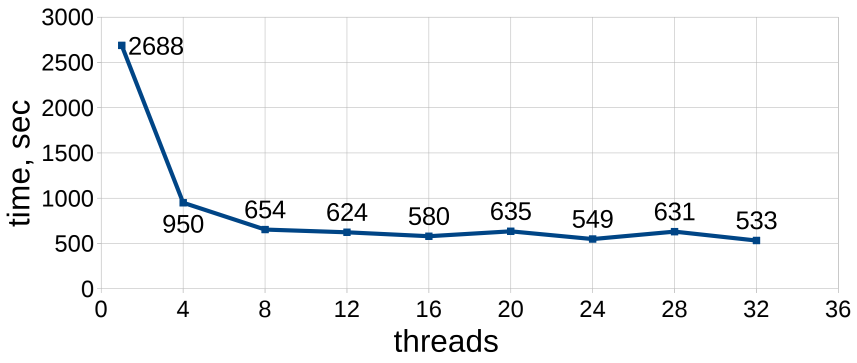

Figure 2.14: Parallel algorithm execution time vs number of threads for Pumsb, $k_{max} = 5$.

Table 2.1: Granularity of 4 threads for Pumsb, $k_{max} = 5$. Time is given in seconds, levelwise. T column shows the total execution time.

k	T	thread 1	thread 2	thread 3	thread 4
3	871	24	24	24	24
4	871	340	344	343	342
5	871	468	501	470	482

Table 2.2: Granularity of 8 threads for Pumsb, $k_{max} = 5$. Time is given in seconds, levelwise. T column shows the total execution time.

k	T	t1	t2	t3	t4	t5	t6	t7	t8
3	674	21	17	19	21	21	21	19	21
4	674	352	284	354	352	285	291	351	352
5	674	297	281	293	293	294	282	289	282

In more detail, tables 2.1, 2.2 and 2.3 show the per thread execution times together with the overall execution time. Data is shown for 4, 8 and 16 threads measured for the Pumsb dataset, $k_{max} = 5$. It can be seen that the thread execution times consistently have a narrow spread, indicating that the workload is divided evenly amongst the threads. That is, there is not one slow thread which dominates parallel execution time. Observe also that the execution times in the last row of each table (when $k = 5 = k_{max}$) decrease as the number of threads is increased but that the maximum thread execution times when $k = 3$ and $k = 4$ do not show a similar decrease. This may be due to the communication overhead when transitioning between layers in the search tree, although we leave detailed

Table 2.3: Granularity of 16 threads for Pumsb, $k_{max} = 5$. Time is given in seconds, levelwise. T column shows the total execution time.

k	T	t1	t2	t3	t4	t5	t6	t7	t8
3	567	20	19	19	20	19	19	20	20
4	567	342	345	258	345	342	333	260	346
5	567	178	171	177	171	170	170	179	179

k	T	t9	t10	t11	t12	t13	t14	t15	t16
3	567	20	19	19	19	20	19	19	19
4	567	270	272	272	345	271	272	342	345
5	567	178	177	171	172	172	177	177	177

analysis of this to future work.

2.6 Summary

In this chapter we introduce a new algorithm for efficiently finding all of the unique and minimal k-itemsets up to a user specified size k_{max}. To achieve this we show how the set of non-uniform and non-unique items can be partitioned into two disjoint subsets of items, the first of which (L_A) represents a search space for Algorithm 1. Then we consider potential performance bottlenecks: the intersection operation at line 31 and the support row test at line 23. This is followed by the presentation of pseudo-code for Algorithm 1 with explanations of its mechanics (the breadth-first search, basic iterations and recursive prefix tree structure). We discuss how the support row test bottleneck can be removed at the cost of much higher memory usage. Then we develop analytical insight into methods for reducing computation at the k_{max} level using Lemma 2.4.1 and Corollary 2.4.1. Finally, the theoretical part concludes with a detailed example and a discussion of parallel operation of the algorithm.

Experimental results present domain-agnostic performance, using randomly generated datasets and evaluating execution time, search space, ordering of items. Domain-specific performance is evaluated via experiments using four well-known datasets, Connect, Pumsb, Poker and USCensus1990, and compared to the two state of the art algorithms in the field, MINIT and MIWI. We find that Algorithm

1 consistently outperforms MINIT for all datasets and all values of k_{max}, while when compared to MIWI, it performs best when the input dataset is computationally expensive (see the results for the Pumsb and USCensus1990 datasets). The scaling of the memory footprint, and the performance of a parallel implementation of Algorithm 1 are presented, and alternate directions for future improvements discussed.

Chapter 3

Minimal Infrequent Itemset Mining

In this chapter we introduce an extended algorithm for efficiently finding all of the τ-infrequent and minimal k-itemsets up to a user specified size k_{max}, and frequency threshold $\tau > 0$. The pre-processing, partitioning, pruning the search space, potential bottlenecks, support itemset testing, reduction in the number of row intersections, correctness theorem and parallelisation for this algorithm remain similar to in the previous chapter with straightforward changes to extend consideration from uniqueness to τ-infrequency.

3.1 Pre-processing

The set of non-uniform and non-τ-infrequent items $I'_{A,\tau} = I_A \setminus U_A \setminus r_{A,\tau}$ with $\tau < |R_a| < n \; \forall a \in I'_{A,\tau}$ can be partitioned into sets $L_{A,\tau}$ and $\bar{L}_{A,\tau} = I'_{A,\tau} \setminus L_{A,\tau}$ such that (i) $R_a \neq R_b \; \forall a, b \in L_{A,\tau}$, (ii) $\forall c \in \bar{L}_{A,\tau}$ there exists $d \in L_{A,\tau}$ with $R_c = R_d$. Revisiting Example 1.2.1, $L_{A,\tau} = \{(1, 1, \{1, 2, 3\}), (2, 2, \{1, 2, 4\}), (3, 3, \{1, 3, 4\})\}$ for $0 < \tau < 3$.

Proposition 3.1.1. *Let $W \subseteq L_{A,\tau}$ be a minimal τ-infrequent itemset. Let $w' \in I_A \setminus L_{A,\tau}$ with $R_w = R_{w'}$ for some $w \in W$. Then $W \setminus \{w\} \cup \{w'\}$ is also a minimal τ-infrequent itemset.*

Proposition 3.1.2. *For any partition $(L_{A,\tau}, I'_{A,\tau} \setminus L_{A,\tau})$ the following holds: $\mathcal{I}_{A,\tau} = \mathcal{L}_{A,\tau} \cup \bar{\mathcal{L}}_{A,\tau} \cup r_{A,\tau}$, where $\bar{\mathcal{L}}_{A,\tau} = \{I \setminus \{a\} \cup \{b\} : I \in \mathcal{L}_{A,\tau}, a \in I, b \in \bar{L}_{A,\tau}, R_a = R_b\}$, $\mathcal{L}_{A,\tau} \subset 2^{L_{A,\tau}}$ is the set of minimal τ-infrequent itemsets.*

The proofs are similar to those of Propositions 2.1.1 and 2.1.2 respectively.

In light of Proposition 3.1.2, our goal can therefore be simplified to finding all τ-infrequent and minimal k-itemsets of $L_{A,\tau}$, $\tau > 0$ and $1 \leq k \leq k_{max}$.

3.2 Extended Kyiv Algorithm

Recall Definition 2.2.1. Let $L^{<}_{A,\tau}$ be a list of the items in $L_{A,\tau}$ sorted in ascending order. The extended Kyiv algorithm performs a breadth first search of the prefix tree defined by $L^{<}_{A,\tau}$. Branches are pruned using Proposition 2.2.1 – if an itemset I fails the support itemset test in Definition 1.2.7(2) then it must be non-minimal and so the subtree with itemset I at the root can be pruned. The key advantage of the breadth-first approach is that the support row test can be performed extremely efficiently, as discussed in more detail in Section 3.2.1. Pseudo-code for the extended Kyiv algorithm is given in Algorithm 2.

In Algorithm 2 the collection of sets $\{P_i\}_{i=1}^{t}$ holds the vertices of level $k-1$ of the pruned prefix graph, and the vertices of level k are stored in $\{P'_i\}_{i=1}^{t'}$. Note that there is never any need to store more than two levels of the pruned prefix tree. The algorithm visits each vertex in level k and takes one of three actions: (i) finds the vertex is a non-minimal itemset and so prunes it (it is not added to P' and its children are not traversed), (ii) finds that the vertex is a minimal τ-infrequent itemset and so prints it (it is not added to P' and its children are not traversed), (iii) finds that the vertex is not τ-infrequent and its children must be traversed.

In the implementation of Algorithm 2, to hold the prefix tree levels we reuse the recursive data structure from Algorithm 1.

3.2.1 Highly Efficient Support Itemset Testing

One of the key benefits of adopting a breadth-first approach in Algorithm 2 is that the computational cost of the support itemset test at line 23 can be reduced to essentially zero. This is because the itemsets $S \subset W$ of size $|S| = |W| - 1$, together with the associated row sets R_S, have already been pre-calculated and stored in data structure $\mathcal{P} := \{P_i\}_{i=1}^{t}$. Hence, evaluating whether there exists an S such that $|R_S| \leq \tau$ simply involves lookups from $\mathcal{P}$, which can be carried out efficiently using an appropriate data structure for $\mathcal{P}$ such as a hash table.

Algorithm 2 Extended Kyiv

```
1: **Input**: dataset $A$, $\tau$, threshold $k_{max}$
2: **Output**: all minimal $\tau$-infrequent $k$-itemsets, $k \leq k_{max}$
3: compute $I_A = I'_{A,\tau} \cup U_A \cup r_{A,\tau}$
4: compute $L_{A,\tau}$ for chosen partition $(L_{A,\tau}, I'_{A,\tau} \setminus L_{A,\tau})$
5: print $\tau$-infrequent items in $r_{A,\tau}$                          ▷ $k = 1$ case
6: sort $L_{A,\tau}$ to obtain $L^{<}_{A,\tau}$
7: $t \leftarrow 0$, $k \leftarrow 2$
8: foreach $a \in L^{<}_{A,\tau}$ do $t \leftarrow t + 1$, $P_t \leftarrow \{a\}$
9: while $k \leq k_{max}$ do
10:   $t' \leftarrow 0$
11:   foreach $i \in \{1, \ldots, t-1\}$ do
12:     $I \leftarrow P_i$
13:     foreach $j \in \{i+1, \ldots, t\}$ do
14:       $J \leftarrow P_j$
15:       ▷ get the highest order items in $I$ and $J$
16:       $a \leftarrow \max(I),\ b \leftarrow \max(J)$
17:       if $I \setminus \{a\} \neq J \setminus \{b\}$ then
18:         break                         ▷ itemsets do not share a common prefix
19:       ▷ itemsets $I$ and $J$ differ exactly by one item now
20:       $W \leftarrow I \cup J$
21:       if $k > 2$ then
22:         ▷ support itemset test, Definition 1.2.7(2)
23:         if $\exists\, S \subset W, |S| = |W| - 1 : |R_S| \leq \tau$ then
24:           continue                    ▷ non-minimal, prune this branch
25:         if $k = k_{max}$ then
26:           ▷ Lemma 3.2.1 and Corollary 3.2.1
27:           if $|R_I| + |R_J| > |R_{I \setminus \{a\}}| + \tau$ then continue
28:           $c \leftarrow \max(J \setminus \{b\})$
29:           if $\min(|R_{I\setminus\{c\}}| - |R_I|, |R_{J\setminus\{c\}}| - |R_J|) + \tau < |R_{I\setminus\{c\}} \cap R_b|$ then
30:             continue
31:       $R_W \leftarrow R_I \cap R_J$                              ▷ intersect rows
32:       if $|R_W| = 0$ or $|R_W| = \min(|R_I|, |R_J|)$ then
33:         continue                      ▷ skip absent and uniform itemsets
34:       if $|R_W| \leq \tau$ then
35:         print $W$                     ▷ minimal $\tau$-infrequent itemset found
36:         foreach $w \in W$ do                        ▷ apply Proposition 3.1.1
37:           if $\exists\, w' \in I'_{A,\tau} \setminus L_{A,\tau} : R_w = R_{w'}$ then
38:             print $W \setminus \{w\} \cup \{w'\}$
39:       else             ▷ need to store non-$\tau$-infrequent minimal itemset
40:         if $k < k_{max}$ then
41:           $t' \leftarrow t' + 1$, $P'_{t'} \leftarrow W$
42:   foreach $t \in \{1, \ldots, t'\}$ do $P_t \leftarrow P'_t$
43:   $k \leftarrow k + 1$, $t \leftarrow t'$
```

Observe that acceleration of the support itemset test at line 23 is achieved in Algorithm 2 at the cost of increased memory usage to store data structure $\mathcal{P}$. As τ increases, the number of prefix tree vertices decreases and the arrays stored at each vertex occupy less memory. Nevertheless, this memory cost remains potentially significant, particularly when τ is small and in the middle of the prefix tree where the number of vertices in each level of the tree is largest. However, in view of the fact that the amount of RAM available is growing at a much faster rate than CPU clock speed, this trade-off between of increased memory consumption for a much reduced computational burden can be a favourable one.

3.2.2 Reducing Number of Row Intersections

The main computational bottleneck of Algorithm 2 is at line 31. Below we provide with the lemma proof only as the corollary proof is very similar to the one in Corollary 2.4.1.

Lemma 3.2.1. *Let $I \subseteq I_A$ be an itemset and $a, b \in I_A$ any items in I_A. If*

$$|R_I \cap R_a| + |R_I \cap R_b| > |R_I| + \tau \tag{3.1}$$

then $I \cup \{a, b\}$ is not a τ-infrequent itemset.

Proof. We proceed by contradiction. Suppose $|R_I \cap R_a| + |R_I \cap R_b| > |R_I| + \tau$ and itemset $I \cup \{a, b\}$ is τ-infrequent (so $|R_I \cap R_a \cap R_b| \leq \tau$). By the distributivity of set intersection, $R_I \cap (R_a \cup R_b) = (R_I \cap R_a) \cup (R_I \cap R_b)$. Hence,

$$\begin{aligned}
&|R_I \cap (R_a \cup R_b)| \\
&\quad = |(R_I \cap R_a) \cup (R_I \cap R_b)| \\
&\quad = |R_I \cap R_a| + |R_I \cap R_b| - |(R_I \cap R_a) \cap (R_I \cap R_b)| \\
&\quad = |R_I \cap R_a| + |R_I \cap R_b| - |R_I \cap R_a \cap R_b|.
\end{aligned}$$

Now $|R_I| \geq |R_I \cap (R_a \cup R_b)|$ and by assumption $|R_I \cap R_a \cap R_b| \leq \tau$. Hence, $|R_I| \geq |R_I \cap R_a| + |R_I \cap R_b| - \tau$, yielding the desired contradiction. □

Corollary 3.2.1. *Let $a_1, \ldots, a_k \in I_A$ be any items from I_A, with $k > 2$. If*

$$\Gamma_0 > \min\{\Gamma_1, \Gamma_2\} + \tau \tag{3.2}$$

then $\{a_1, \ldots, a_k\}$ is not a τ-infrequent itemset, where

$$\begin{aligned}
\Gamma_0 &:= |\cap_{i=1}^{k-3} R_{a_i} \cap R_{a_{k-1}} \cap R_{a_k}|, \\
\Gamma_1 &:= |\cap_{i=1}^{k-3} R_{a_i} \cap R_{a_{k-1}}| - |\cap_{i=1}^{k-3} R_{a_i} \cap R_{a_{k-2}} \cap R_{a_{k-1}}|, \\
\Gamma_2 &:= |\cap_{i=1}^{k-3} R_{a_i} \cap R_{a_k}| - |\cap_{i=1}^{k-3} R_{a_i} \cap R_{a_{k-2}} \cap R_{a_k}|.
\end{aligned}$$

In the final iteration (when $k = k_{max}$) we can use Lemma 3.2.1 and Corollary 3.2.1 to test for τ-infrequency before carrying out the intersection at line 31. If either test concludes that the itemset is not τ-infrequent, then there is no need to perform the row intersection.

3.2.3 Correctness and Parallelisation

Theorem 3.2.1. *Algorithm 2 terminates in finite time and finds all minimal τ-infrequent itemsets of I_A up to size k_{max}.*

The proof repeats the corresponding proof of the theorem of Algorithm 1.

Algorithm 2 can be readily parallelised using shared-memory threads: at level k within the prefix tree assign all vertices sharing the same parent at level $k-1$ within the prefix tree to the same thread and then in each thread execute the loop starting at line 13 in Algorithm 2. The shared memory allows each thread access to the prefix tree information stored in P_j, $j \in \{i+1, \cdots, t\}$.

When the number of available threads is less than the number of parent vertices at level $k-1$ in the prefix tree, work must be allocated among the threads. The work associated with each parent vertex is dominated by the number of row intersections to be carried out. This number can be accurately estimated based on the number of children of the parent vertex, and so the work associated with each parent vertex estimated in advance. Using these work estimates, load-balanced scheduling of work among the threads can then be efficiently realised. In principle, the parallel form of Algorithm 2 coincides with the one of Algorithm 1.

3.3 Worst Time Complexity

In this section we present several observations on the worst time complexity of Algorithm 2. Of main interest is the while loop at line 9.

Let A be a dataset with the list of items L_A and $k_{max} = |L_A|$, that is an entire prefix tree is to be traversed (recall that $k_{max} \leq m$ and often $m \ll |L_A|$). Suppose there is no pruning possible in order to find all minimal τ-infrequent k-itemsets, $k \leq k_{max}$. Thus the intersection operation at line 31 will occur at each iteration of the while loop. To further simplify the analysis and also maximise the size of the tree, assume that $l = |R_a|\ \forall a \in L_A$: $k_{max} \leq l \leq n$ and $|R_I| = |R_J| = l - k + 1$ for each pair (I, J) at level k of the prefix tree. Hence there are no absent or uniform itemsets to be found. Finally, letting $\tau < l - k_{max} + 1$ (which is very common) means that there are no τ-infrequent itemsets in the tree.

In the following we let the prefix tree be a graph (V, E). Denote (V, E) by (V_s, E_s) where $s = |L_A|$.

Theorem 3.3.1. *The worst time complexity of the while loop in Algorithm 2 is* $O(l \cdot 2^{|L_A|})$.

Proof. We need to estimate the number of prefix tree traversals and multiply it by the average cost of the intersection operation, which is $O(l + l) = O(l)$. Since we traverse the tree in a breadth-first fashion, the worst time complexity is $O(V + E)$.

We need to prove that $V_s = 2^s - 1$ and $E_s = \sum_{i=1}^{s-1}(s - i)2^{i-1}$. When $s = 1$ there is one item in L_A, so $V = 1$, $E = 0$. Similarly $V = 3$, $E = 1$ when $s = 2$; $V = 7$, $E = 4$ when $s = 3$; $V = 15$, $E = 11$ when $s = 4$ and so on. For example, Figure 2.1 represents a prefix tree with $V = 31$ and $E = 26$. Notice the recurrent relation: $V_{s+1} = 2V_s + 1$ and $E_{s+1} = 2E_s + s$. This means that if we build a prefix tree by adding one item b to the prefix tree (V_s, E_s), we will have to replicate (V_s, E_s) and connect b to s items of its copy. In this way the newly created tree doubles the number of vertices V_s and edges E_s and adds 1 vertex with s edges.

Hence

$$\begin{aligned} V_{s+1} &= 2V_s + 1 = 2 \cdot (2V_{s-1} + 1) + 1 \\ &= \underbrace{2 \cdot (2 \cdot (\ldots (2 \cdot (}_{s} \underbrace{2V_0}_{=0} + 1 \underbrace{) + 1) \ldots) + 1) + 1}_{s} \end{aligned}$$

$$\begin{aligned}
&= 1 + 2^1 + 2^2 + \dots + 2^s = 2^{s+1} - 1. \\
E_{s+1} &= 2E_s + s = 2 \cdot (2E_{s-1} + s - 1) + s \\
&= \underbrace{2 \cdot (2 \cdot (\dots (2 \cdot (}_{s-1} \underbrace{2E_1}_{=0} +1 \underbrace{) + 2) \dots) + s - 1) + s}_{s-1} \\
&= 1 \cdot 2^{s-1} + 2 \cdot 2^{s-2} + \dots + (s-2) \cdot 2^2 + (s-1) \cdot 2^1 + s \\
&= \sum_{i=0}^{s-1} (s-i) 2^i.
\end{aligned}$$

The sum $V_s + E_s = 2^s - 1 + \sum_{i=1}^{s-1}(s-i)2^{i-1}$ is dominated by the summand 2^s. Thus $O(V+E) = O(V_s + E_s) = O(2^s) = O(2^{|L_A|})$. Consequently the worst case time complexity of Algorithm 2 is $O(l \cdot 2^{|L_A|})$. □

Note the dependency between l and $|L_A|$: if $l = n$ then $|L_A| = 1$. Moreover, if $I'_A \setminus L_A = \emptyset$ (there is single partition) then the available space in A for items of L_A can be computed as (items a, b in different columns must be shifted to avoid $R_a = R_b$ because $|R_a| = |R_b| = l$):

$$n+(n-1)+(n-2)+\dots+n-(m-1) = nm-(1+2+\dots+m-1) = nm-(m-1)m/2.$$

Thus $|L_A| \cdot l \leq nm - (m-1)m/2$.

3.3.1 Worst Time Complexity vs k_{max}

The main computational complexity is associated with the middle of the prefix tree (V, E). Let us relax the above conditions to $k_{max} \leq \left\lceil \frac{|L_A|}{2} \right\rceil$ and let (V_k, E_k) denote a prefix tree with vertices and edges lying on the first k levels of (V, E), $1 \leq k \leq k_{max}$. If $k_{max} = 1$ then $V_1 = |L_A|$, $E_1 = 0$. Let $s = |L_A|$. If $k_{max} = 2$ then $V_2 = s(s+1)/2$, $E_2 = s(s-1)/2$. Thus $O(V_2 + E_2) = O(s^2) = O(|L_A|^2)$ and the worst time complexity of the while loop in Algorithm 2 is $O(l \cdot |L_A|^2)$. Obviously $V_{k+1} = V_{new} + V_k$, $E_{k+1} = V_{new} + E_k$. That is, to compute the number of vertices (edges) for $k+1$ it is enough to compute the number of vertices at the very last level $k+1$ (which is equal to the number of new edges) and add it to the number of vertices V_k (edges E_k).

If $k_{max} = 3$ then

$$V_{new} = 1 + (1+2) + (1+2+3) + \dots + (1+2+3+\dots+s-2)$$

$$
\begin{aligned}
&= \underbrace{1 + 2\cdot\frac{1+2}{2} + 3\cdot\frac{1+3}{2} + \dots + (s-3)\cdot\frac{s-2}{2} + (s-2)\cdot\frac{s-1}{2}}_{\text{group every two consecutive summands from the beginning}} \\
&= \begin{cases} 2^2 + 4^2 + \dots + (s-4)^2 + (s-2)^2 & \text{if } s \text{ is even} \\ 2^2 + 4^2 + \dots + (s-5)^2 + (s-3)^2 + (s-2)(s-1)/2 & \text{if } s \text{ is odd} \end{cases} \\
&= 4\cdot\begin{cases} 1^2 + 2^2 + \dots + (s/2-1)^2 & \text{if } s \text{ is even} \\ 1^2 + 2^2 + \dots + (\frac{s-3}{2})^2 + (s-2)(s-1)/8 & \text{if } s \text{ is odd} \end{cases} \\
&= 4\cdot\begin{cases} (s/2-1)\cdot(s/2-1+1)\cdot(2(s/2-1)+1)/6 & \text{if } s \text{ is even} \\ (\frac{s-3}{2})\cdot(\frac{s-3}{2}+1)\cdot(s-3+1)/6 + (s-2)(s-1)/8 & \text{if } s \text{ is odd} \end{cases} \\
&= \begin{cases} (s-2)\cdot s\cdot(s-1)/6 & \text{if } s \text{ is even} \\ (s-3)\cdot(s-1)\cdot(s-2)/6 + (s-2)(s-1)/2 & \text{if } s \text{ is odd} \end{cases} \\
&= (s-2)(s-1)s/6 \\
V_3 &= V_{new} + V_2 = (s-2)(s-1)s/6 + s(s+1)/2 = (s^3+5s)/6 \\
E_3 &= V_{new} + E_2 = (s-2)(s-1)s/6 + s(s-1)/2 = (s^3-s)/6.
\end{aligned}
$$

The sum $V_3 + E_3$ is dominated by s^3. Thus $O(V_3 + E_3) = O(s^3) = O(|L_A|^3)$ and the worst time complexity of the while loop in Algorithm 2 is $O(l \cdot |L_A|^3)$.

If $k_{max} = 4$ then

$$
\begin{aligned}
V_{new} &= 1 + (1+1+2) + (1+1+2+1+2+3) + \dots + \Big(1+1+2+\dots \\
&\quad + (1+2+\dots+s-3)\Big) \\
&= \underbrace{(s-3)}_{\text{A}} + \underbrace{(s-4)\cdot 2\cdot\frac{1+2}{2}}_{\text{B}} + \underbrace{(s-5)\cdot 3\cdot\frac{1+3}{2}}_{\text{C}} + \dots \\
&\quad + \underbrace{3\cdot(s-5)\cdot\frac{1+s-5}{2}}_{\text{C}} + \underbrace{2\cdot(s-4)\cdot\frac{1+s-4}{2}}_{\text{B}} + \underbrace{(s-3)\cdot\frac{1+s-3}{2}}_{\text{A}} \\
&= \frac{s}{2}\cdot\Big((s-3) + 2(s-4) + 3(s-5)\Big) + \dots \\
&\quad + \frac{s}{2}\cdot\begin{cases} \frac{s-4}{2}\Big(s - (\frac{s-4}{2}+2)\Big) + (\frac{s-4}{2}+1)\Big(s - (\frac{s-4}{2}+2+1)\Big) & \text{if } s \text{ is even} \\ \frac{s-3}{2}\Big(s - (\frac{s-3}{2}+2)\Big) & \text{if } s \text{ is odd} \end{cases} \\
&< \frac{s}{2}\cdot(s-3)\Big(1+2+3+\dots+\begin{cases} \frac{s-4}{2} + \frac{s-2}{2} & \text{if } s \text{ is even} \\ \frac{s-3}{2} & \text{if } s \text{ is odd} \end{cases}\Big) \\
&= \frac{s}{2}\cdot(s-3)\cdot\begin{cases} (s-2)\cdot s/8 & \text{if } s \text{ is even} \\ (s-3)(s-1)/8 & \text{if } s \text{ is odd} \end{cases} < s^4.
\end{aligned}
$$

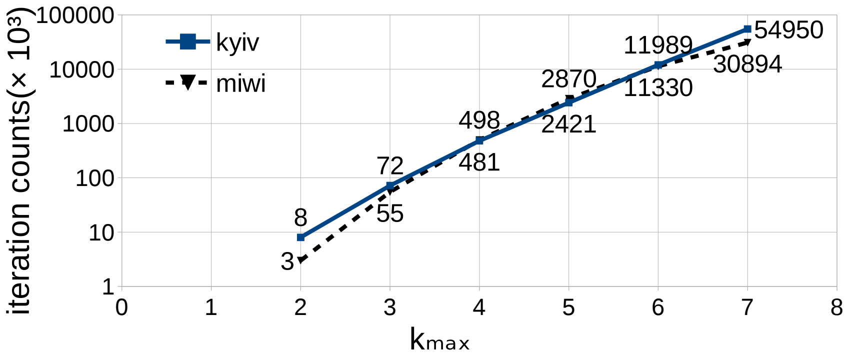

Figure 3.1: Number of recursive calls (MIWI) and intersections (Kyiv) vs k_{max} for Connect dataset, $\tau = 1$.

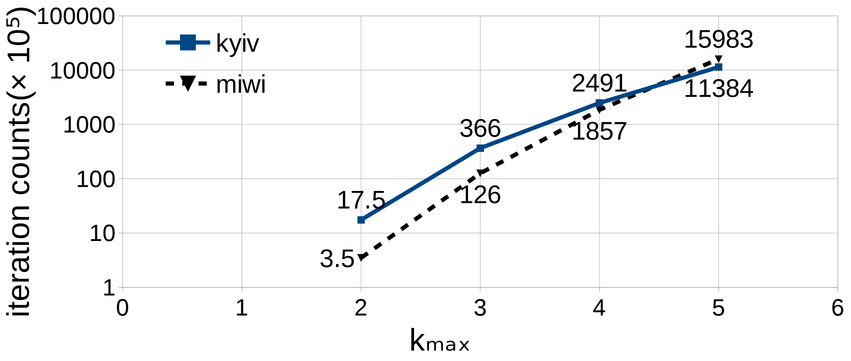

Figure 3.2: Number of recursive calls (MIWI) and intersections (Kyiv) vs k_{max} for Pumsb dataset, $\tau = 1$.

The sum $V_4 + E_4$ has an upper bound of s^4. Thus $O(V_4 + E_4) = O(s^4) = O(|L_A|^4)$ and the worst time complexity of the while loop in Algorithm 2 is $O(l \cdot |L_A|^4)$.

It would be interesting to compute the worst time complexity for $4 < k_{max} \leq \left\lceil \frac{|L_A|}{2} \right\rceil$ since it may be polynomial too, for instance $O(l \cdot |L_A|^{k_{max}})$. Also it is worth expanding our analysis to the variable size of arrays stored at each vertex.

3.3.2 Iteration Counts vs k_{max}

In this section we present measurements of the number of iterations of the main loop of Algorithm 2 and compare them to the number of iterations of the main loop in MIWI, a recursive function each call to which corresponds to a prefix tree vertex, vs k_{max}. MINIT also has a recursive main function but it reports unrealistically small counts so we do not present it here.

Figures 3.1 and 3.2 show the number of iterations vs k_{max} for the Connect and

Pumsb datasets when $\tau = 1$. Although the curves fit together closely, one iteration that corresponds to a given prefix tree vertex may take very different time if computed for either of the algorithms, which is well observed for computationally expensive datasets (the Pumsb or USCensus1990).

3.4 Experimental Results

If not otherwise stated, all experiments in this section were carried out using ascending itemlist order, Lemma 3.2.1 and Corollary 3.2.1.

3.4.1 Hardware and Software Setup

We implemented Algorithm 2 in Java (version 1.7.0_25) using the hppc (version 0.5.2) library, which can be found at `http://labs.carrotsearch.com/hppc.html`. For comparison with the serial version of Algorithm 2, we also implemented a state-of-the-art algorithm MINIT [HM07] in Java (using the C++ implementation kindly provided by the developers of MINIT) and used the C++ implementation of the MIWI algorithm [CG13], kindly provided by its developers.

For testing we used an Amazon cr1.8xlarge instance with an Intel Xeon CPU E5-2670 0 @ 2.60GHz 32 processor (up to 32 hyperthreads), 244Gb of memory, 64-bit Linux operating system (kernel version 3.4.62-53.42. amzn1.x86_64 of Red Hat 4.6.3-2 Linux distribution (Amazon Linux AMI release 2013.09)).

3.4.2 Execution Time vs k_{max}

In this section we reused the datasets defined in Section 2.5.3.1. All measurements in the current section are averaged over three consecutive runs of each algorithm.

The impact of the frequency threshold τ is of greatest importance for infrequent itemset mining. We chose two values of τ (5 and 10) close to the unique case described in Section 2 and one farther – 100.

Figures 3.3, 3.4, 3.5 and 3.6 show similar behaviour of the algorithms to that reported in Figures 2.8, 2.9, 2.10 and 2.11 respectively. Similarly for Figures 3.7, 3.8, 3.9, 3.10 and 3.11, 3.12, 3.13, 3.14. Consequently we may expect the relative performance of the algorithms to be approximately τ-invariant.

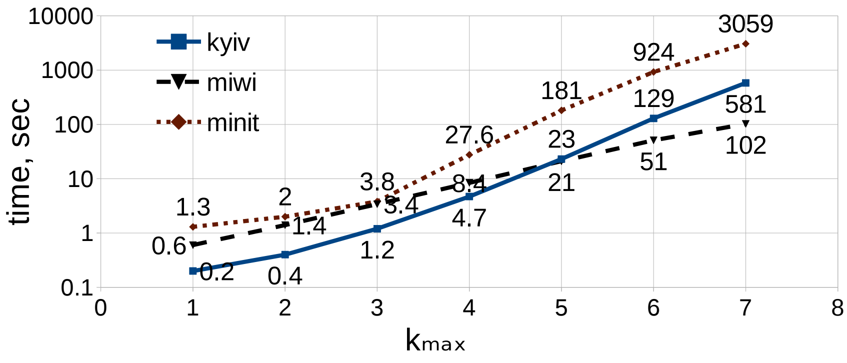

Figure 3.3: Execution time vs k_{max} for Connect dataset, $\tau = 5$.

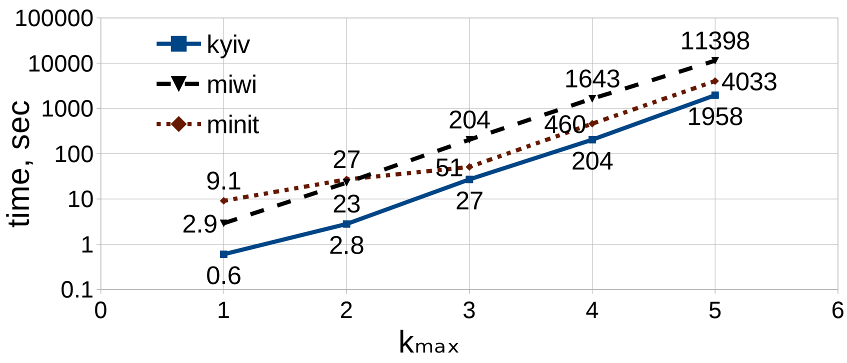

Figure 3.4: Execution time vs k_{max} for Pumsb dataset, $\tau = 5$.

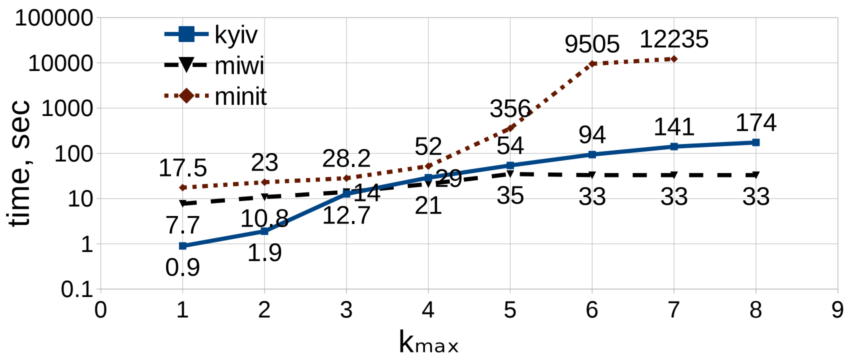

Figure 3.5: Execution time vs k_{max} for Poker dataset, $\tau = 5$.

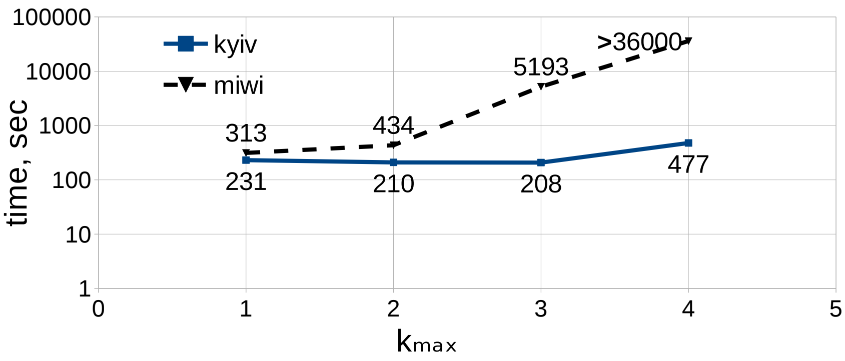

Figure 3.6: Execution time vs k_{max} for USCensus1990 dataset, $\tau = 5$.

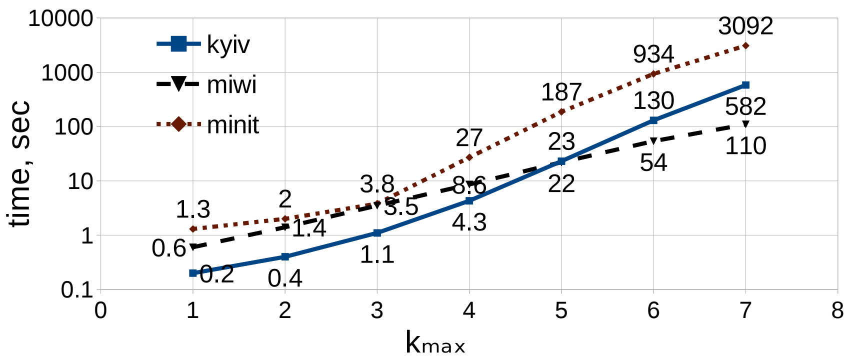

Figure 3.7: Execution time vs k_{max} for Connect dataset, $\tau = 10$.

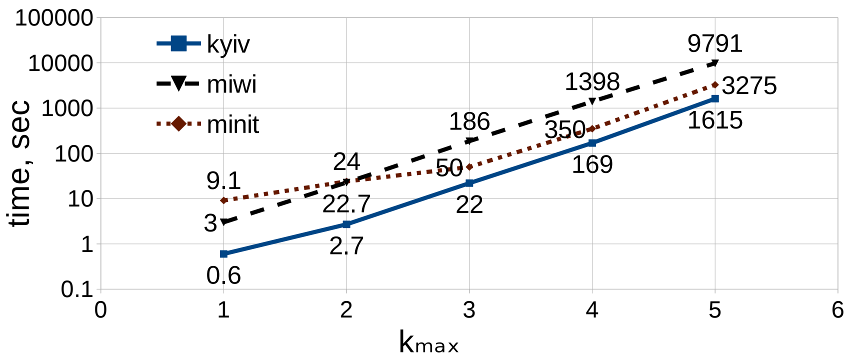

Figure 3.8: Execution time vs k_{max} for Pumsb dataset, $\tau = 10$.

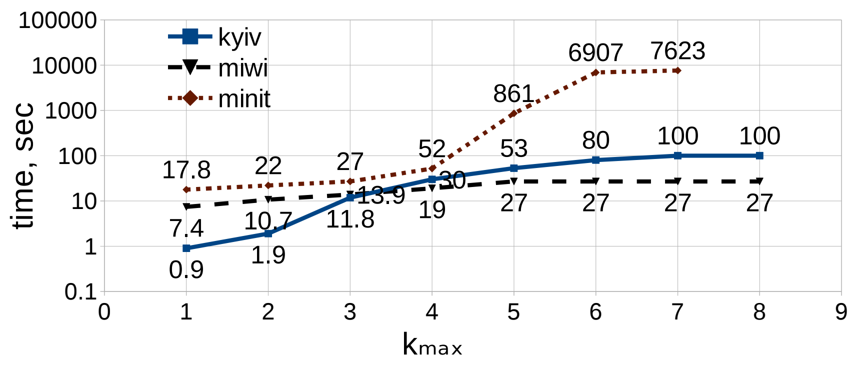

Figure 3.9: Execution time vs k_{max} for Poker dataset, $\tau = 10$.

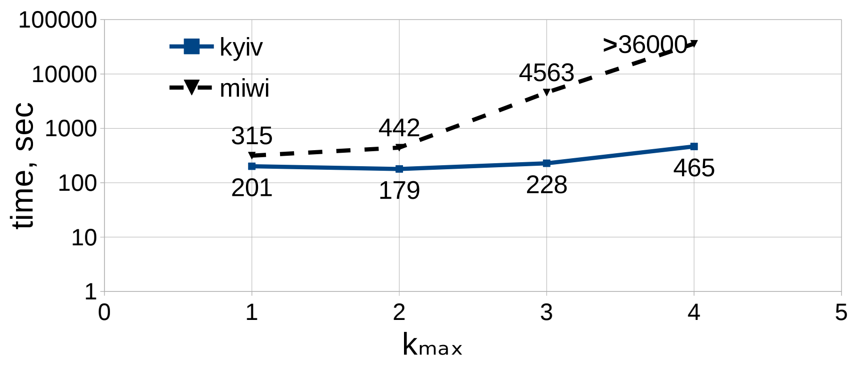

Figure 3.10: Execution time vs k_{max} for USCensus1990 dataset, $\tau = 10$.

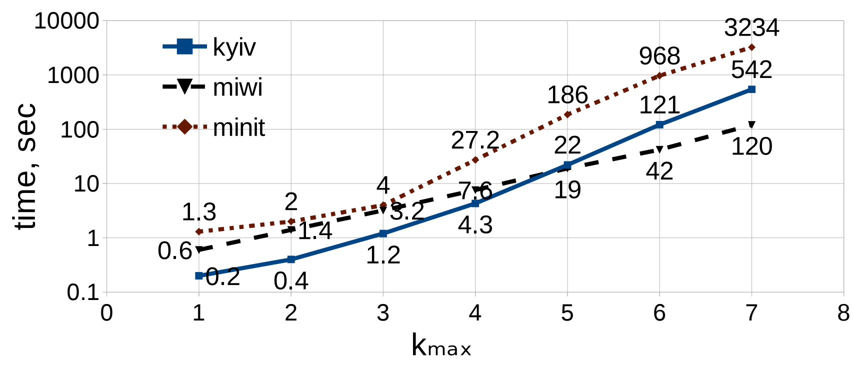

Figure 3.11: Execution time vs k_{max} for Connect dataset, $\tau = 100$.

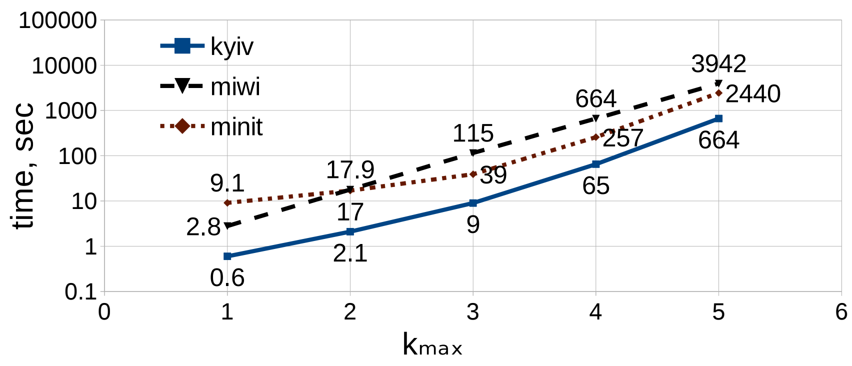

Figure 3.12: Execution time vs k_{max} for Pumsb dataset, $\tau = 100$.

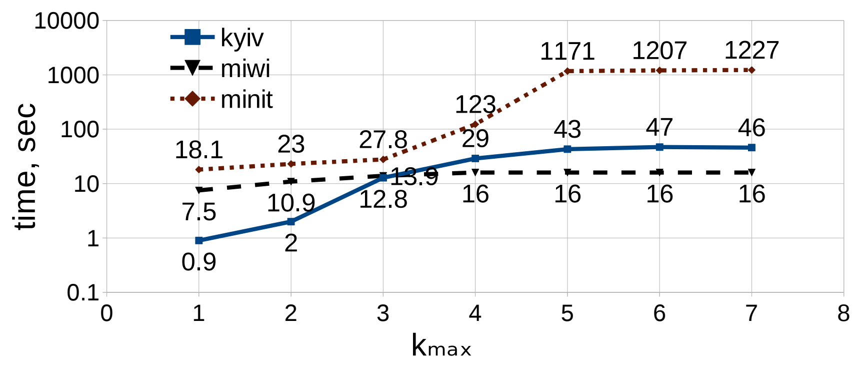

Figure 3.13: Execution time vs k_{max} for Poker dataset, $\tau = 100$.

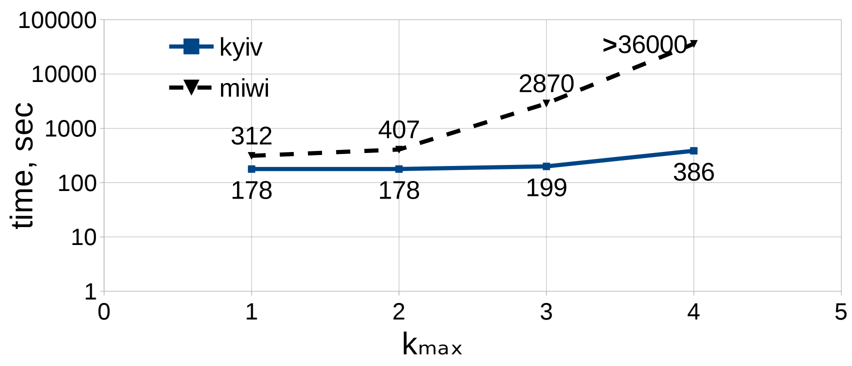

Figure 3.14: Execution time vs k_{max} for USCensus1990 dataset, $\tau = 100$.

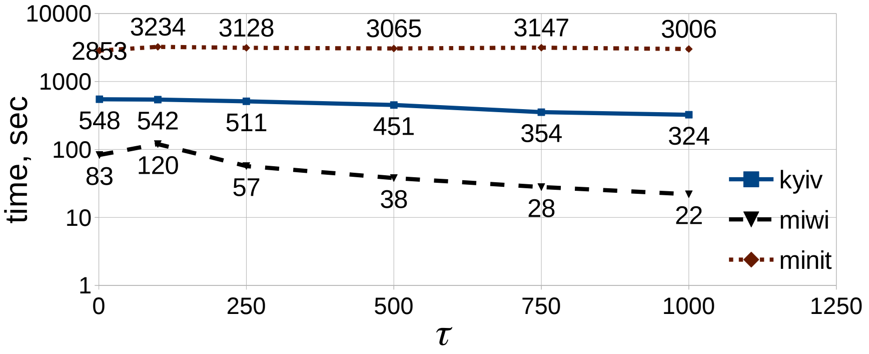

Figure 3.15: Execution time vs τ for Connect dataset, $k_{max} = 7$.

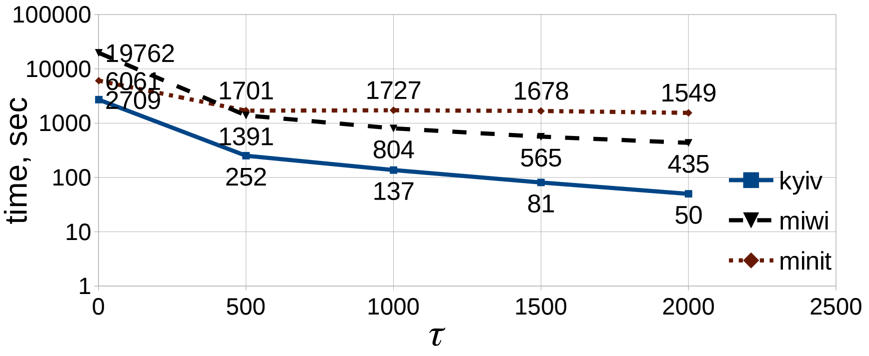

Figure 3.16: Execution time vs τ for Pumsb dataset, $k_{max} = 5$.

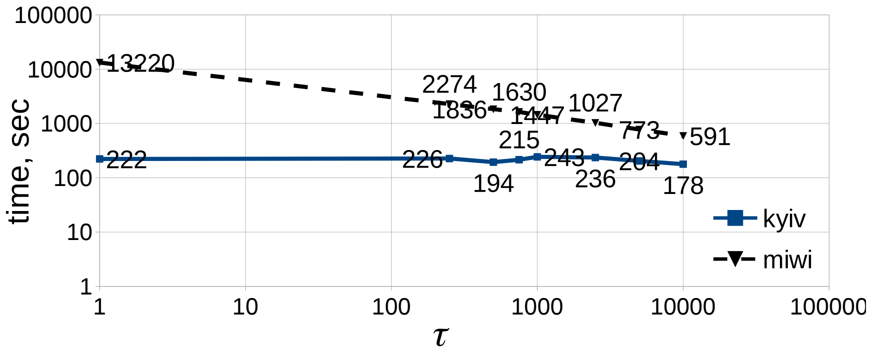

Figure 3.17: Execution time vs τ for USCensus1990 dataset, $k_{max} = 3$, $\tau \in \{1, 250, 500, 750, 1000, 2500, 5000, 10000\}$.

3.4.3 Execution Time vs τ

From Figures 3.3 - 3.14 it can be seen that the execution time of all algorithms tends to fall with increasing τ. That is, finding minimal unique itemsets is more demanding than finding infrequent itemsets, as might be expected. This is studied in more detail in Figures 3.15 - 3.17 which plots the measured execution times vs τ.

It can be seen from Figure 3.15 that MINIT's execution time initially increases with τ (see [HM07] where similar behaviour is reported), and then later falls as τ is increased further. Similarly, the execution time of MIWI also increases initially. We think that these initial increases are caused by the design of the algorithm and not by the dataset complexity since it is not present for Algorithm 2.

For this relatively simple dataset MIWI offers the shortest execution time. However, for the more complex Pumsb and USCensus1990 datasets it can be seen that Algorithm 2 offers the shortest execution time, although the performance gap between MIWI and Algorithm 2 narrows for large τ with the USCensus1990 dataset.

3.5 Summary

In this chapter we introduce an extended algorithm for efficiently finding all of the τ-infrequent and minimal k-itemsets up to a user specified size k_{max}, and frequency threshold $\tau > 0$. To achieve that we show how the set of non-uniform and non-τ-infrequent items can be partitioned into two disjoint subsets of items, the first of which ($L_{A,\tau}$) represents a search space for Algorithm 2. The search space pruning, potential performance bottlenecks, Algorithm 2's pseudo-code, support itemset testing, reduction in the number of row intersections, correctness theorem and parallelisation for Algorithm 2 remain similar to those in Chapter 2 with straightforward changes to extend consideration from uniqueness to τ-infrequency, so we introduce these very briefly. The theoretical discussion concludes by showing that even though the problem has exponential time complexity, it can be computed in polynomial time if k_{max} is set low.

We present experimental results sufficiently evaluating the execution time of the Kyiv, MINIT and MIWI algorithms vs τ and k_{max}. Experiments are presented for the four datasets used in Chapter 2. The tests reveal that Algorithm 2 consistently outperforms MINIT for all datasets and all values of k_{max}, while when compared to MIWI, performs best when the input dataset is computationally expensive (see the results for the Pumsb and USCensus1990 datasets). We also observe that Algorithm 2's execution time tends to decrease with τ and its comparative performance with the MIWI and MINIT algorithms is approximately τ-invariant.

Chapter 4

Conclusions

A new algorithm for finding quasi-identifiers within a data set is introduced, where a quasi-identifier is a subset of attributes that can uniquely identify data set records (or identify that a record lied within a small group of τ records). This algorithm is demonstrated to be substantially faster than the state of the art, to scale well to large data sets and to be amenable to parallelisation with well-balanced thread execution times.

4.1 Future Work

Here we highlight some ideas for improvements and optimisation.

Regarding memory usage, suppose Kyiv that is able to compute the k^*-itemsets by intersecting the $(k^* - 1)$-itemsets but that the algorithm goes out of memory at the $k^* + 1$ level. We might keep intersecting the $(k^* - 1)$-itemsets in order to find not only the k^*-itemsets, but also the $(k^* + \delta)$-itemsets, where $\delta \in \mathbb{N}$ at each consecutive level of the prefix tree. This would allow us to halt growth in memory usage as this is mainly used for itemset storage. Related technical refinements could be to implement the corresponding itemset test using the $(k^* - 1)$-itemsets and to use data compression for the array storage to decrease the memory consumption, albeit at the cost of increased execution time.

Regarding data structures, it would be useful to get a better understanding of the most efficient structures for storing the prefix tree and handling the search space operations. The insights gained might improve the parallel form of the algorithm. One possible direction would be to look at an array implementation of a tree structure representation, e.g. similar to the work in [GZ05].

The main computational bottleneck, the intersection operation, could potentially be improved by making use of the specialised SSE (Streaming SIMD Extensions) instructions available on Intel processors. There exists performance analysis [Kat12] indicating that use of these instructions might produce a 4× speed up.

And of course, more mathematical principles may be obtained to express item relations and so more efficiently prune the algorithm's search space.

Bibliography

[AMS^{+}96] R. Agrawal, H. Mannila, R. Srikant, H. Toivonen, and A. I. Verkamo. Fast discovery of association rules. *Advances in Knowledge Discovery and Data Mining*, 10(5):307–328, 1996.

[BZ06] M. Barbaro and T. Zeller. A face is exposed for AOL searcher No. 4417749. In *New York Times*, August 2006.

[CG07] T. Calders and B. Goethals. Non-derivable itemset mining. *Data Mining and Knowledge Discovery*, 14(1):171–206, 2007.

[CG13] L. Cagliero and P. Garza. Infrequent weighted itemset mining using frequent pattern growth. *Trans. Knowledge and Data Engineering*, 2013.

[DL14] K. Demchuk and D. J. Leith. A fast minimal infrequent itemset mining algorithm. `http://arxiv.org/abs/1403.6985`, 2014.

[DZNJ07] X. Dong, Z. Zheng, Z. Niu, and Q. Jia. Mining infrequent itemsets based on multiple level minimum supports. *Proc. ICICIC*, 2007.

[Ell07] M. Elliot. Using targeted perturbation of microdata to protect against intelligent linkage. In *EUROSTAT Work Session on statistical data confidentiality*, December 2007.

[EMF02] M. J. Elliot, A. M. Manning, and R. W. Ford. A computational algorithm for handling the special uniques problem. *International Journal on Uncertainty, Fuzziness and Knowledge-based Systems*, 10(5):493–509, 2002.

[GGM04] W. Gross, P. Guiblin, and K. Merrett. Risk assessment of the individual sample of anonymised records (SAR) from the 2001 census. In *UK Office of National Statistics*, 2004.

[GMB11] A. Gupta, A. Mittal, and A. Bhattachrya. Minimally infrequent itemset mining using pattern-growth paradigm and residual trees. *Proc. COMAD*, 21:1131–1158, 2011.

[GZ05] G. Grahne and J. Zhu. Fast algorithms for frequent itemset mining using FP-trees. *IEEE Transactions on Knowledge and Data Engineering*, 17(10):1347–1362, 2005.

[HM07] D. J. Haglin and A. M. Manning. On minimal infrequent itemset mining. *Proc. Int. Conf. on Data Mining, DMIN*, pages 141–147, 2007.

[HMM$^+$09] D. J. Haglin, K. R. Mayes, A. M. Manning, J. Feo, J. R. Gurd, M. Elliot, and J. A. Keane. Factors affecting the performance of parallel mining of minimal unique itemsets on diverse architectures. *Concurrency and Computation: Practice and Experience*, 21(9):1131–1158, 2009.

[HPYM04] J. Han, J. Pei, Y. Yin, and R. Mao. Mining frequent patterns without candidate generation: a frequent-pattern tree approach. *Data Mining and Knowledge Discovery*, 8(1):53–87, 2004.

[HSE12] S. Hommes, R. State, and T. Engel. Detecting stealthy backdoors with association rule mining. In *Proc Networking*, volume 7290, pages 161–171, 2012.

[JYT$^+$13] Y. Ji, H. Ying, J. Tran, P. Drews, A. Mansour, and R. M. 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, 2013.

[Kat12] I. Katsov. Fast intersection of sorted lists using SSE instructions. `http://highlyscalable.wordpress.com/2012/06/05/fast-intersection-sorted-lists-sse`, 2012.

[KR05] Y. S. Koh and N. Rountree. Finding sporadic rules using apriori-inverse. In *Proc 9th Pacific-Asia conference on Advances in Knowledge Discovery and Data Mining*, volume 3518, pages 97–106, 2005.

[LDR05] K. LeFevre, D. J. DeWitt, and R. Ramakrishnan. Incognito: efficient full-domain K-anonymity. In *Proc SIGMOD*, pages 49–60, 2005.

[LRRV10] J. M. Luna, A. Ramirez, J. R. Romero, and S. Ventura. An intruder detection approach based on infrequent rating pattern mining. In *Intelligent Systems Design and Applications (ISDA)*, pages 682–688, 2010.

[MH05] A. M. Manning and D. J. Haglin. A new algorithm for finding minimal sample uniques for use in statistical disclosure assessment. *IEEE International Conference on Data Mining (ICDM05)*, pages 290–297, 2005.

[MHK08] A. M. Manning, D. J. Haglin, and J. A. Keane. A recursive search algorithm for statistical disclosure assessment. *Data Mining and Knowledge Discovery*, 16(2):165–196, 2008.

[REA08] A. Rahman, C. I. Ezeife, and A. K. Aggarwal. WiFi miner: an online apriori-infrequent based wireless intrusion detection system. *Proc. Sensor-KDD*, 2008.

[SNV07] L. Szathmary, A. Napoli, and P. Valtchev. Towards rare itemset mining. *Proc. Int. Conf. on Tools with Artificial Intelligence*, pages 305–312, 2007.

[SVN10] L. Szathmary, P. Valtchev, and A. Napoli. Generating rare association rules using the minimal rare itemsets family. *Int. J. Software Informatics*, 4(3):219–238, 2010.

[SVNG12] L. Szathmary, P. Valtchev, A. Napoli, and R. Godin. Efficient vertical mining of minimal rare itemsets. *Proc. Conf. on Concept Lattices and Their Applications*, pages 269–280, 2012.

[Swe02] L. Sweeney. k-Anonymity: a model for protecting privacy. *International Journal on Uncertainty, Fuzziness and Knowledge-based Systems*, 10(5):557–570, 2002.

[TKD11] S. Tsang, Y. S. Koh, and G. Dobbie. RP-tree: rare pattern tree mining. In *Data Warehousing and Knowledge Discovery*, volume 6862, pages 277–288, 2011.

[TKD13] S. Tsang, Y. S. Koh, and G. Dobbie. Finding interesting rare association rules using rare pattern tree. In *Special Issue on Advances in Data Warehousing and Knowledge Discovery*, volume 7790, pages 157–173, 2013.

[TMK13] M. Templ, B. Meindl, and A. Kowarik. IHSN SDC Introduction. `http://ec.europa.eu/eurostat/ramon/statmanuals/files/SDC_Handbook.pdf`, 2013.

[TMK14] M. Templ, B. Meindl, and A. Kowarik. Introduction to Statistical Disclosure Control (SDC). In *CRAN SDCMicro Documentation*, 2014.

[TS13] L. Troiano and G. Scibelli. A time-efficient breadth-first level-wise lattice-traversal algorithm to discover rare itemsets. *Data Mining and Knowlege Discovery*, pages 1–35, 2013.

[TSB09] L. Troiano, G. Scibelli, and C. Birtolo. A fast algorithm for mining rare itemsets. *Proc. Int. Conf. on Intelligent Systems Design and Applications*, 2009.

[ZY07] L. Zhou and S. Yau. Efficient association rule mining among both frequent and infrequent items. *Computers and Mathematics with Applications*, 54(6):737–749, 2007.