The Open University of Israel
Department of Mathematics and Computer Science

Practical Approximation Algorithms for Optimal k-Anonymity

Thesis submitted as partial fulfillment of the requirements
towards an M.Sc. degree in Computer Science
The Open University of Israel
Department of Mathematics and Computer Science

By

Batya Kenig

Prepared under the supervision of Dr. Tamir Tassa

June 2009

I wish to express my gratitude to Dr. Tamir Tassa for his devoted counsel, guidance and patience.

A special thanks to my husband Tal - for his love and encouragement.

Contents

Abstract

k-Anonymity is a privacy preserving method for limiting disclosure of private information in data mining. In a k-anonymized table, every record is indistinguishable from at least $k-1$ other records, whence an adversary who attempts to extract personal information on an individual who is represented in the released table cannot link that individual to less than k records in the table. The process of anonymizing a database table involves generalizing table entries and, consequently, loss of relevant information for the data miner. This motivates the search for anonymization algorithms that achieve the required level of anonymization while incurring a minimal loss of information. The problem of k-anonymity with minimal loss of information is NP-hard. In this study we present several known approximation and heuristic algorithms for the k-anonymization problem. Our main contribution is a practical algorithm that enables solving the k-anonymization problem with an approximation guarantee of $O(\ln k)$. This algorithm improves an algorithm due to Aggarwal et al. [1] that offers an approximation guarantee of $O(k)$, and generalizes that of Park and Shim [15] that was limited to the case of generalization by suppression. Our algorithm uses techniques that we introduce herein for mining generalized frequent itemsets. Experiments show that the new algorithm provides better results than the leading approximation algorithm, as well as known heuristic algorithms.

Chapter 1

Introduction

In recent years, there has been tremendous growth in the amount of personal data that can be collected and analyzed. Data mining tools are increasingly being used to infer trends and patterns. Of particular interest are data containing structured information on individuals. However, the use of data containing personal information has to be restricted in order to protect individual privacy. Although identifying attributes like ID numbers and names are never released for data mining purposes, sensitive information might still leak due to *linking attacks* that are based on the public attributes, a.k.a *quasi-identifiers.* Such attacks may join the quasi-identifiers of a published table with a publicly accessible table like the voters registry, and thus disclose private information of specific individuals. In fact, it was shown in [20] that 87% of the U.S. population may be uniquely identified via the combination of the three quasi-identifiers: birthdate, gender and zipcode. Privacy-preserving data mining [2] has been proposed as a paradigm of exercising data mining while protecting the privacy of individuals.

One of the most well-studied methods of privacy preserving data mining is called k-anonymization, proposed by Samarati and Sweeney [17, 18, 21]. This method suggests to generalize the values of the public attributes, so that each of the released records becomes indistinguishable from at least $k-1$ other records, when projected on the subset of public attributes. The private data is then associated to sets of records of size at least k. The values of the table are modified via the operation of *generalization*, while keeping them consistent with the original ones. A *cost function* is used to measure the amount of information lost by the generalization process. The objective is to modify the table entries so that the table becomes $k-$anonymized and the information loss (or cost function) is minimized. Meyerson and Williams

[13] introduced this problem and studied it under the assumption that the table entries may be either left unchanged or totally suppressed. In that setting, the cost function to be minimized is the total number of suppressed entries in the table. They showed that the problem is NP-hard by showing a reduction from the k-dimensional perfect matching problem. They devised two approximation algorithms; one that runs in time $O(n^{2k})$ and achieves an approximation ratio of $O(k \ln k)$; and another that has a fully polynomial running time (namely, it depends polynomially on both n and k) and guarantees an approximation ratio of $O(k \ln n)$. Aggarwal et al. [1] extended the setting of suppressions-only to generalizations by hierarchical clustering trees, and devised an approximation algorithm with an approximation ratio of $O(k)$. Gionis and Tassa [7] improved the first algorithm of [13] by offering an approximation ratio of $O(\ln k)$, rather than $O(k \ln k)$, and applying it to a wider class of generalization operators (the generalizations in that class are called "proper" and they include generalization by suppression as well as generalizations by hierarchical clustering trees), and a wider class of measures of loss of information. However, the runtime of their algorithm remains $O(n^{2k})$. Finally, Park and Shim [15] devised, independently of [7], an improved and practical version of the algorithm in [7] that also provides an approximation ratio of $O(\ln k)$. However, it applies only to generalizations by suppression.

Another approach to the problem is using heuristical algorithms. The algorithm of choice is called the agglomerative algorithm [6] and it outperforms the approximation algorithm of [1] which is the currently best practical algorithm with a proven approximation guarantee (for the setting of generalization, and not just suppressions).

The main contribution of this thesis is a practical anonymization algorithm that guarantees an approximation ratio of $O(\ln k)$ and applies to all proper generalizations and to a wide class of measures of loss of information. Our algorithm is based on the algorithm of Park and Shim which was restricted to suppressions only. It is also based on techniques that we devise herein for mining generalized frequent itemsets. When comparing the proposed algorithm to the currently best known approximation algorithm (the algorithm of [1]) and to the currently best known heuristical algorithm (the Agglomerative algorithm of [6]) it outperforms both of them in terms of information loss. In terms of runtime, it is a practical algorithm, and in some cases it is even faster than the above mentioned algorithms.

1.1 Overview of the thesis

In Chapter 2 we present the fundamentals of k-anonymization. Then, in Chapter 3 we provide an overview of the common measures of information loss.

In Chapter 4 we present k-anonymization algorithms that have a known approximation guarantee. The first one (Section 4.2) is the $O(\ln k)$-approximation algorithm of [7]. Then we review the closely-related approximation algorithm of [15] that offers the same approximation ratio of $O(\ln k)$, but has better runtime performance, although it is restricted to generalization by suppression (Section 4.2.1). Finally, we discuss in Section 4.3 the $O(k)$-approximation algorithm of [1] which is the best practical and general approximation algorithm for the problem of k-anonymity with minimal loss of information.

In Chapter 5 we turn our attention from approximation algorithms with a known approximation guarantee to heuristic algorithms. We focus on the agglomerative algorithms of [6].

As the main contribution of this thesis borrows ideas from algorithms for mining generalized frequent itemsets, we proceed to discuss algorithms for mining frequent itemsets. In Chapter 6 we give an overview of the known frequent itemset mining algorithms in the standard setting (where all items in the itemset are single-valued, as opposed to generalized values). Then, in Chapter 7, we present the problem of mining generalized frequent itemsets and introduce our novel frequent generalized itemset mining algorithm.

In Chapter 8 we present our practical $O(\ln k)$-approximation algorithm for the problem of k-anonymization with minimal loss of information. That algorithm is based on the algorithms that were presented in Section 4.2 and on the algorithm for mining generalized frequent itemsets that was presented in Chapter 7.

Finally, in Chapter 9 we compare the performance of our proposed algorithm to the performance of the best available approximation and heuristic algorithms. The comparison of the best known approximation algorithm (the algorithm that we devise in Chapter 8) to the best known heuristic (Chapter 5) reveals the following conclusion: The approximation algorithm is preferable not only from a theoretical point of view (as it provides an approximation guarantee) but also from a practical point of view, as it outperforms the heuristic algorithm in terms of the corresponding information loss.

The thesis is concluded in Chapter 10.

Chapter 2

$k-$Anonymization

Here we provide the basic definitions and lemmas related to the k-anonymization problem. These will be used in subsequent chapters of this study.

2.1 Preliminaries

A database table holds information on individuals in some population $U = \{u_1, \ldots, u_n\}$. Each individual is described by a set of r public attributes (a.k.a quasi-identifiers), $A_1, \ldots, A_r$, and s private attributes, $Z_1, \ldots, Z_s$. Each of the attributes consists of several possible values:

$$A_j = \{a_{j,l} : 1 \leq l \leq m_j\}, \ \ 1 \leq j \leq r,$$

and

$$Z_j = \{z_{j,l} : 1 \leq l \leq n_j\}, \ \ 1 \leq j \leq s.$$

We use the same notation, namely A_j or Z_j, to denote both the attribute name and the domain in which it takes values. For example, if A_j is gender, then $A_j = \{M, F\}$, while if it is the age of the individual, it is a bounded non-negative natural number. The public table holds all publicly available information on the individuals in U; it takes the form

$$D = \{R_1, \ldots, R_n\}, \tag{2.1}$$

where $R_i \in A_1 \times \cdots \times A_r,\ 1 \leq i \leq n$. The corresponding private table holds the private information

$$D' = \{S_1, \ldots, S_n\}, \tag{2.2}$$

where $S_i \in Z_1 \times \cdots \times Z_s$, $1 \le i \le n$. The complete table is the concatenation of those two tables, $D||D' = \{R_1||S_1, \ldots R_n||S_n\}$. We refer to the records of R_i and S_i, $1 \le i \le n$, as public and private records, respectively. Each cell in the table will be referred to as a table entry. The jth component of the record R_i (the (i,j)th entry in the table D) will be denoted by $R_i(j)$. It should be noted that the sets in (2.1) and (2.2) may be multisets, that is they may include repeated records.

2.2 Generalization

Generalization is the act of replacing the values that appear in the table with subsets of values, so that an entry $R_i(j) \in A_j$, $1 \le i \le n$, $1 \le j \le r$, is replaced by a subset of A_j that includes the value of this entry.

Definition 2.2.1. *Let A_j, $1 \le j \le r$, be finite sets and let $\overline{A}_j \subseteq \mathcal{P}(A_j)$ be a collection of subsets of A_j. A mapping $g : A_1 \times \cdots \times A_r \rightarrow \overline{A}_1 \times \cdots \times \overline{A}_r$ is called a generalization if for every $(b_1, \cdots, b_r) \in A_1 \times \cdots \times A_r$ and $g(b_1, \cdots, b_r) = (B_1, \cdots B_r)$, it holds that $b_j \in B_j$, $1 \le j \le r$.*

2.2.1 Generalization types

In Definition 2.2.1, each attribute A_j, $1 \le j \le r$, is assigned a collection of subsets $\overline{A}_j \subseteq \mathcal{P}(A_j)$. According to the choice of $\overline{A}_j$ we get different types of generalizations. Let us consider three examples:

- *Generalization by suppression.* This refers to a mapping g that either leaves entries unchanged or replaces them by the entire set of attribute values. That is, $\overline{A}_j = A_j \cup \{A_j\}$ for all $1 \le j \le r$. Namely, $g(b_1, \ldots, b_r) = (\bar{b}_1, \ldots, \bar{b}_r)$ where $\bar{b}_j \in \{b_j, A_j\}$, $1 \le j \le r$.

- *Generalization by hierarchical clustering trees.* In [1], Aggarwal et al. considered a setting in which for every attribute A_j there is a corresponding balanced tree, $T(A_j)$, that describes a hierarchical clustering of A_j. Each node of $T(A_j)$ represents a subset of A_j, the root of the tree is the entire set A_j, the descendants of each node represent a partition of the subset that corresponds to the father node, and the leaves are all singleton subsets. Given such a balanced tree, the generalization operators may replace an entry $R_i(j)$ with any of its ancestors in $T(A_j)$. Generalization by suppression is a special case of generalization by clustering trees where all trees are of height 2.

- *Unrestricted generalization.* In this case $\overline{A}_j = \mathcal{P}(A_j)$. Each entry may be replaced by any subset of A_j that includes it.

Example 2.2.1. *Consider a table with two attributes,* age (A_1) *and* gender (A_2). *An example of a valid generalization by suppression of a record* $R = (28, M) \in D$ *is*

$$g(R) = (28, *) = (28, \{M, F\}) .$$

An example of a generalization which is not restricted to suppressions only is

$$g(R) = ([20-29], *) = ([20-29], \{M, F\}) .$$

Finally, an example of an unrestricted generalization is

$$g(R) = (\{21, 28, 33\}, M) .$$

We will assume hereinafter that the collections of subsets used for generalization, $\overline{A}_j$, $1 \le j \le r$, satisfy the following property [7].

Definition 2.2.2. *Given an attribute* $A = \{a_1, \ldots, a_m\}$, *a corresponding collection of subsets* $\overline{A}$ *is called proper if it includes all singleton subsets* $\{a_i\}$, $1 \le i \le m$, *it includes the entire set* A, *and it is laminar in the sense that* $B_1 \cap B_2 \in \{\emptyset, B_1, B_2\}$ *for all* $B_1, B_2 \in \overline{A}$.

It is shown in [7, Lemma 3.3] that the class of proper generalization coincides with the class of generalizations by possibly unbalanced hierarchical clustering trees. (Such a clustering tree, or a taxonomy is illustrated in Figure 2.1). Hence, our framework in this study extends the framework that was considered in [1] (i.e., balanced hierarchical clustering trees) and, in particular, the framework of generalization by suppression [13, 15].

In this setting, an item may be generalized to any of its ancestors; for example, in the taxonomy in Figure 2.1, the item *Sport-Jacket* may be be left unchanged, or generalized to one of the following: *Outdoors*, *Clothes* or *Clothing*.

We will now define generalizations for an entire tables.

Definition 2.2.3. *Let* $D = \{R_1, \ldots, R_n\}$ *be a table with public attributes* $A_1, \ldots, A_r$, *let* $\overline{A}_1, \ldots, \overline{A}_r$ *be a corresponding collection of subsets, and* $g_i : A_1 \times \cdots \times A_r \to \overline{A}_1 \times \cdots \times \overline{A}_r$ *be corresponding generalization operators,* $1 \le i \le n$. *Let* $\overline{R}_i := g_i(R_i)$ *be the generalization of record* R_i, $1 \le i \le n$. *Then* $g(D) = \{\overline{R}_1, \ldots, \overline{R}_n\}$ *is a generalization of* D.

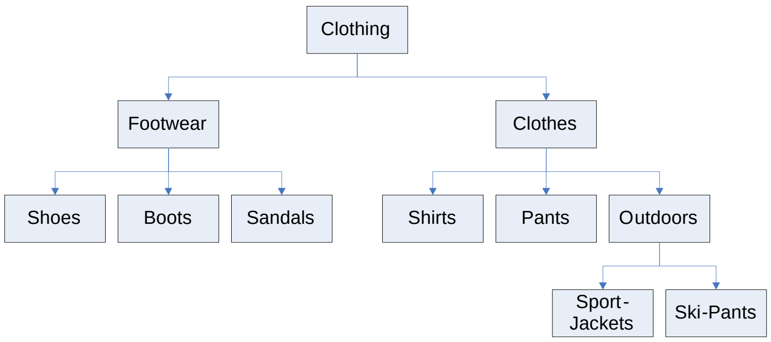


Figure 2.1: A Proper Taxonomy

There are two main models of generalization. In *global recording,* each collection of subsets $\overline{A_j}$ is a clustering of the set A_j and then every entry in the jth column of the table is mapped to the unique subset in $\overline{A_j}$ that contains it. As a consequence, every single value $a \in A_j$ is always generalized in the same manner. In *local recording,* the collection of subsets $\overline{A_j}$ covers the set A_j but it is not a clustering. In that case, each entry in the table's jth column is generalized independently to one of the subsets in $\overline{A_j}$ which includes it. In such a model, if the age 34 appears in the table in several records, it may be left unchanged in some, generalized to 30 - 39, or totally suppressed in other records. Clearly, local recording is more flexible and might enable k-anonymity with a smaller loss of information.

Definition 2.2.4. *A relation $\sqsubseteq$ is defined on $\overline{A_1} \times \cdots \times \overline{A_r}$ as follows: If $R, R' \in \overline{A_1} \times \cdots \times \overline{A_r}$, then $R \sqsubseteq R'$ if and only if $R(j) \subseteq R'(j)$ for all $1 \leq j \leq r$. In that case, we say that R specifies R', or equivalently, that R' generalizes R.*

It is easy to see that $\sqsubseteq$ defines a partial order on $\overline{A_1} \times \cdots \times \overline{A_r}$. We may use this partial order to define a partial order on the set of all generalizations of a given table.

Definition 2.2.5. *Let $D = \{R_1, \ldots, R_n\}$ be a table, and let $g(D) = \{\overline{R}_1, \ldots, \overline{R}_n\}$ and $g'(D) = \{\overline{R}'_1, \ldots, \overline{R}'_n\}$ be two generalizations of D. Then, $g(D) \sqsubseteq g'(D)$ if $R_i \sqsubseteq R'_i$ for all $1 \leq i \leq n$.*

2.3 The k-anonymization problem

The k-anonymization optimization problem is defined as follows [1, 7, 13]

Definition 2.3.1. *A k-anonymization of table $D = \{R_1, \ldots, R_n\}$ is a generalization $g(D) = \{\overline{R}_1, \ldots, \overline{R}_n\}$ where for all $1 \le i \le n$, there exist indices $1 \le i_1 < i_2 < \cdots < i_{k-1} \le n$, all of which are different from i, such that $\overline{R}_i = \overline{R}_{i_1} = \cdots = \overline{R}_{i_{k-1}}$.*

Let $\Pi = \Pi(D, g(D))$ be a measure of the information loss incurred during the generalization process. Then the k-anonymization optimization problem is as follows:

Definition 2.3.2. *Let $D = \{R_1, \ldots, R_n\}$ be a table with public attributes A_j, $1 \le j \le r$. Given collections $\overline{A}_j \subseteq \mathcal{P}(A_j)$, and a measure of information loss Π, find a corresponding k-anonymization, $g(D) = \{\overline{R}_1, \ldots, \overline{R}_n\}$, where $\overline{R}_i \in \overline{A}_1 \times \cdots \times \overline{A}_r$, that minimizes $\Pi(D, g(D))$.*

Chapter 3

Measures of Information Loss

In this chapter we survey common measures of information loss that are used in order to assess the utility of anonymized tables. We begin with the description of basic cost-measures (Section 3.1), and then continue to discuss entropy-based measures Section 3.2). Monotonicity, which is an important property of information-loss measures, is discussed in Section 3.3.

3.1 Basic measures

Let D be the original public table and $g(D)$ be a generalization of D. A critical question in the context of k-anonymity is how to define $\Pi(D, g(D))$ – the distance between D and $g(D)$ in the sense of the amount of information lost during the generalization process. Meyerson and Williams [13] considered the case of generalization by suppression, and their measure counted the number of suppressed entries in the generalized table. Aggarwal et. al . [1] used a more general model of generalization by hierarchical clustering, and proposed the tree measure. According to this measure, generalizing an exact table entry, belonging to the jth attribute, to the rth level in the corresponding hierarchical clustering tree, incurs a cost of $\frac{r}{l_j}$, where l_j is the height of the tree and $0 \leq r \leq l_j$.

The Loss Metric LM [11, 14] is a more precise and more general version of the above defined measure. According to the LM measure, the cost per each table entry is a number between 0 (no generalization at all) and 1 (total suppression) that penalizes the generalization that was made in that entry according to the size of the generalized subset. The overall cost is the average cost per table entry:

$$\Pi_{LM}(D, g(D)) = \frac{1}{nr} \cdot \sum_{i=1}^{n} \sum_{j=1}^{r} \frac{|\overline{R}_i(j)| - 1}{|A_j| - 1} . \tag{3.1}$$

The Ambiguity Metric AM [14] is the average size of the Cartesian products of all generalized entries in each generalized record in the table. This measure represents the number of (theoretically) possible combinations of original records that a generalized record can stand for:

$$\Pi_{AM}(D, g(D)) = \frac{1}{n} \cdot \sum_{i=1}^{n} \prod_{j=1}^{r} |\overline{R}_i(j)| .$$

An immediate drawback of the AM cost measure is that it counts also combinations of attribute values that do not appear in the original table. For example, let $D = \{(1, 1), (1, 2), (2, 3)\}$ be the original table. Let $\overline{R} = (\{1, 2\}, \{1, 2\})$ be a generalized record in $g(D)$.The AM cost of $\overline{R}$ equals $2 \cdot 2 = 4$, which says that there are four combinations that $\overline{R}$ can stand for, while in practice only two of these possibilities exist in D.

The Discernability Metric DM [5] defines the cost of each generalized record $\overline{R}_i$ as the number of generalized records in the anonymized table that are indistinguishable from it. A suppressed record is penalized by the size of the entire table, $|D|$. Therefore the total cost of the DM measure is the sum of squares of the sizes of all non-suppressed clusters, plus the number of totally suppressed clusters multiplied by $|D|$. Since our k-anonymization algorithms usually produce clusters with sizes very close to k, all such anonymizations have approximately the same DM cost, what makes this measure less useful.

While all of the measures that were described above consider only the values of the public attributes of the table, the next measure takes into account also the private attributes. The Classification Metric CM [11] defines a possible penalty for each generalized record of the table, based on a chosen classified attribute, called *the class label.* A record $\overline{R}_i$ is penalized either if its class label differs from the majority class label of its cluster $S(\overline{R}_i)$, or if $\overline{R}_i$ is totally suppressed. Formally:

$$penalty(\overline{R}_i) = \begin{cases} 1 & if\ \overline{R}_i\ is\ suppressed \\ 1 & if\ class(\overline{R}_i) \neq majority(S(\overline{R}_i)) \\ 0 & otherwise \end{cases}$$

The CM measure is then defined as the average penalty of all rows:

$$\Pi_{CM}(D, g(D)) = \frac{1}{n} \cdot \sum_{i=1}^{n} penalty(\overline{R}_i)\,.$$

The rationale behind the classification metric is that homogeneous clusters have more utility than heterogeneous clusters, because they indicate a stronger association between the public attribute values of the cluster and the trained classified attribute.

3.2 Entropy based measures

The measures described in the following sections are based on Shannon's entropy definition. Those were first suggested by Tassa and Gionis in [7].

Definition 3.2.1. *Let X be a discrete random variable on a finite set $\chi = \{x_1, \ldots, x_n\}$, with probability distribution $p(x) = Pr(X = x)$. The entropy $H(X)$ of X is defined as $H(X) = -\sum_{x\in\chi} p(x)\log p(x) = -E(\log p(X))$, where* $\log = \log_2$.

The entropy is essentially a measure of the amount of information that is delivered by revealing the value of a random sample of X. According to this definition, the quantity $-\log p(x)$ is the amount of information conveyed by the value $x \in \chi$.

The public table $D = \{R_1, \ldots, R_n\}$ induces a probability distribution for each of the public attributes. Let X_j, $1 \le j \le r$, denote the value of the attribute A_j in a randomly selected record from D. Then

$$Pr(X_j = a) = \frac{\#\{1 \le i \le n \,:\, R_i(j) = a\}}{n}\,.$$

The corresponding entropy measure is:

$$H(X_j) = -\sum_{a\in A_j} Pr(X_j = a)\log Pr(X_j = a)\,.$$

Let B_j be a subset of A_j. Then, the conditional entropy $H(X_j|B_j)$ is defined as

$$H(X_j|B_j) = -\sum_{b\in B_j} Pr(X_j = b|X_j \in B_j)\log Pr(X_j = b|X_j \in B_j)$$

where

$$Pr(X_j = b | X_j \in B_j) = \frac{\#\{1 \le i \le n \,:\, R_i(j) = b\}}{\#\{1 \le i \le n \,:\, R_i(j) \in B_j\}}\,,\; b \in B_j\,.$$

When $B_j = A_j$, then $H(X_j|B_j) = H(X_j)$, while in the other extreme, where B_j consists of one element, $H(X_j|B_j) = 0$ (0 uncertainty). This allows us to define the following cost function of a generalization operator:

Definition 3.2.2. *Let $D = \{R_1, \dots, R_n\}$ be a table having public attributes $A_1, \dots, A_r$, and let X_j be the random variable that equals the value of the jth attribute A_j, $1 \le j \le r$, in a randomly selected record from D. Then if $g(D) = \{\overline{R}_1, \dots, \overline{R}_n\}$ is a generalization of D then*

$$\Pi_e(D, g(D)) = \frac{1}{nr} \cdot \sum_{i=1}^{n} \sum_{j=1}^{r} H(X_j | \overline{R}_i(j))$$

is the entropy measure of the loss of information caused by generalizing D to $g(D)$.

The entropy measure is the only measure, presented so far, that uses the well accepted information measure of Shannon's entropy in order to capture information loss due to generalization. Therefore it is more accurate than the other measures. The entropy measure enables distinguishing between "simple" attributes (such as gender) and attributes that convey more information (such as age or zip-code). It will prefer generalizing or concealing the less informative attributes. For example ([7]), in the setting of generalization by suppression, if we have a table that contains two attributes, gender and zip-code.

$$D = \begin{bmatrix} M & 41278 \\ M & 98705 \\ F & 41278 \\ F & 98705 \end{bmatrix}$$

The following two tables are 2-anonymizations of D:

$$g_1(D) = \begin{bmatrix} * & 41278 \\ * & 98705 \\ * & 41278 \\ * & 98705 \end{bmatrix}, \quad g_2(D) = \begin{bmatrix} M & * \\ M & * \\ F & * \\ F & * \end{bmatrix}$$

The LM metric will consider the two generalizations as equally distant from D in terms of information loss. The entropy measure, on the other hand, will favor $g_1(D)$ since the entropy of the gender attribute is smaller that that of the zip-code attribute. The generalized table $g_1(D)$ conceals the less-informative attribute and leaves in-tact the attribute that may be of better use for data-mining purposes.

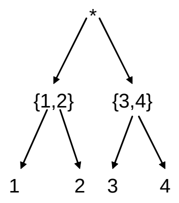
3.3 Monotonicity of cost measures

A natural property that one might expect from any measure of loss of information is monotonicity:

Definition 3.3.1. *Let D be a table, let $g(D)$ and $g'(D)$ be two generalizations of D, and let Π be any measure of information loss. Then, Π is called monotone if $\Pi(D, g(D)) \leq \Pi(D, g'(D))$ whenever $g(D) \sqsubseteq g'(D)$.*

The tree measure and LM measure are monotone. The entropy measure, on the other hand, is not always monotone, as we proceed to exemplify:

Example 3.3.1. *Consider a table with one attribute that may receive values $\{1, 2, 3, 4\}$ with probabilities $\{1 - 3\varepsilon, \varepsilon, \varepsilon, \varepsilon\}$ respectively, where $\varepsilon \ll 1$. The entropy of this attribute is $h(1-3\varepsilon, \varepsilon, \varepsilon, \varepsilon) \approx 0$. Next, assume that the values of this attribute are arranged in the following hierarchy tree:*



Entries with the value 4 may be generalized to $\{3, 4\}$ or be suppressed. While the first possibility incurs a cost of 1 bit (since $\{3,4\}$ contains two values with an equal probability of ε), the second option entails a cost which equals the attribute's entropy, $h(1-3\varepsilon, \varepsilon, \varepsilon, \varepsilon) \approx 0$. Consequently, the entropy measure favors the suppression of the value 4 over its generalization to the partial subset $\{3, 4\}$.

From a data mining point of view, monotonicity is essential. However, the non-monotonicity of the entropy measure is not a severe issue. First, anomalies such as the one described in the above example are very rare in practice. Secondly, the non-monotonicity of the entropy measure may be rectified. Specifically, given any collection of subsets of a given attribute, $\overline{A}$, it is always possible to find a partial collection $\hat{A} \subseteq \overline{A}$ so that the entropy measure on the new collection is monotone. In [7], the process for finding such a partial collection is outlined.

Chapter 4

Approximation Algorithms for k-Anonymization

In this section we describe three approximation algorithms for the problem of k-anonymization with minimal loss of information. The first algorithm was proposed by Gionis and Tassa [7], and is based on the algorithm of Meyerson and Williams [13], which runs in time $O(n^{2k})$. This algorithm is based on the idea of using the set-cover approximation algorithm in order to obtain an approximation algorithm for the k-anonymization problem. Park and Shim [15] also use the set-cover approximation algorithm in order to devise an algorithm for the k-anonymization problem, when restricted to the case of generalization by suppression. They introduced the idea of using data-mining techniques in order to significantly reduce the algorithm's runtime. The third algorithm is the forest algorithm, that was introduced in [1]. It runs in fully polynomial time and guarantees an approximation of $O(k)$.

Section 4.1 provides definitions and lemmas which will be used in subsequent sections. Section 4.2 provides the general framework for generating an approximate solution to the k-anonymization problem using the set-cover approximation algorithm. We also describe how frequent itemsets algorithms are used in order to reduce the running time of the set-cover approximation algorithm for the k-anonymization problem. Finally, section 4.3 describes the Forest algorithm.

4.1 Preliminaries

Any k-anonymization of D defines a partition of D into disjoint clusters, where each cluster consists of all records that were replaced by the same generalized record. All records in the same cluster are replaced with the minimal generalized record that generalizes all of them.

Definition 4.1.1. *Let $A_1, \ldots, A_r$ be attributes with corresponding collections of subsets $\overline{A}_1, \ldots, \overline{A}_r$ that are proper (Definition 2.2.2). Then given $M \subseteq A_1 \times \ldots \times A_r$, its closure is defined as*

$$\overline{M} = \min_{\sqsubseteq} \left\{ C \in \overline{A}_1 \times \ldots \times \overline{A}_r : R \sqsubseteq C \quad \textit{for all } R \in M \right\} .$$

We now use the notion of closure to define the generalization cost of a subset of records.

Definition 4.1.2. *Let D be a table and let $M \subseteq D$ be a subset of records. Then the generalization cost of M, denoted $d(M)$, is defined as the generalization cost of its closure, $\overline{M}$, according to the used measure of loss of information, Π. Furthermore, its anonymization cost is*

$$ANON(M) = \sum_{R \in M} d(M) = |M| \cdot d(M) .$$

For example, for the LM measure:

$$d(M) = d_{LM}(\overline{M}) = \sum_{j=1}^{r} \frac{|\overline{M}_j| - 1}{|A_j| - 1} .$$

The generalization cost of M is the amount of information that we lose for each record $R \in M$ if we replace it by $\overline{M}$, the closure of M. Assuming monotonicity, $\overline{M}$ is the best generalized record that may replace all elements in M. The closely related anonymization cost,

$$ANON(M) = |M| \cdot d(M) = |M| \cdot \sum_{j=1}^{r} \frac{|\overline{M}_j| - 1}{|A_j| - 1}$$

is the overall generalization cost if we replace each record $R \in M$ with the generalized record $\overline{M}$.

The following lemma introduces the property of sub-additivity, which will be used later in our analysis of the algorithms.

Lemma 4.1.1. *Assume that all collections of subsets, $\overline{A}_j$, $1 \le j \le r$, are proper (Definition 2.2.2). Then the generalization cost of any cost measure, is sub-additive in the sense that for all $S, T \subseteq A_1 \times \ldots \times A_r$,*

$$S \cap T \neq \emptyset \Longrightarrow d(S \cup T) \le d(S) + d(T)\,.$$

Proof. Denote $U = S \cup T$, and let S_j, T_j, U_j be the sets of values of the jth attribute, $1 \le j \le r$, that appear in S, T, U respectively. Let $\overline{S}_j, \overline{T}_j, \overline{U}_j$ be the jth components of the closures $\overline{S}, \overline{T}, \overline{U}$, respectively. Since $S \cap T \neq \emptyset$, also $S_j \cap T_j \neq \emptyset$. But since $\overline{A_j}$ is proper, we have that $\overline{S}_j \subseteq \overline{T}_j$ or $\overline{T}_j \subseteq \overline{S}_j$, so either $\overline{U}_j = \overline{S}_j$ or $\overline{U}_j = \overline{T}_j$. This implies

$$d(\overline{U}_j) = \begin{cases} d(\overline{S}_j), & \overline{T}_j \subseteq \overline{S}_j \\ d(\overline{T}_j), & \overline{S}_j \subseteq \overline{T}_j \end{cases} \le d(\overline{S}_j) + d(\overline{T}_j)\,.$$

□

4.2 k-anonymization using set cover

Given a k-anonymization of a table, we can assume that all of its induced clusters are of sizes between k and $2k-1$; otherwise, each cluster of size at least $2k$ could be arbitrarily split into two clusters of size at least k; assuming monotonicity, the amount of information loss would not increase by such a split.

Definition 4.2.1. *An $[\ell, m]$-cover ($\ell \le m$) is a cover γ of D by subsets $S \subset D$ where all subsets $S \in \gamma$ are of size $\ell \le |S| \le m$. An $[\ell, m]$-clustering is an $[\ell, m]$-cover where all subsets are disjoint.*

Let $P_{[k,2k-1]}$ be the set of all $[k, 2k-1]$-clusterings of a given table D. Clearly, any optimal $k-$anonymization corresponds to a member of this set. Let $\Gamma_{[k,2k-1]}$ be the set of all $[k, 2k-1]$-covers of D. Given a $[k, 2k-1]$-cover $\gamma \in \Gamma_{[k,2k-1]}$ of the table D, we define its generalization cost as:

$$d(\gamma) = \sum_{S \in \gamma} d(S)\,. \tag{4.1}$$

Furthermore, if $\gamma \in P_{[k,2k-1]}$ we define its anonymization cost as

$$ANON(\gamma) = \sum_{S \in \gamma} |S| \cdot d(S)\,. \tag{4.2}$$

If $g(D)$ is the k-anonymization that corresponds to the $[k, 2k-1]$-clustering γ, then $ANON(\gamma) = \Pi(D, g(D))$.

Given a table D and a positive integer k, the $[k, 2k-1]$-minimum clustering problem is the problem of finding $\gamma \in P_{[k,2k-1]}$ that minimizes $d(\gamma)$. Note that the problem of finding $\gamma \in P_{[k,2k-1]}$ that minimizes $ANON(\gamma)$ is the k-anonymization problem.

The following lemma describes the relation between the two cost functions.

Lemma 4.2.1. *For all $\gamma \in P_{[k,2k-1]}$,*

$$k \cdot d(\gamma) \leq ANON(\gamma) \leq (2k-1) \cdot d(\gamma) \tag{4.3}$$

Proof. As we have

$$k \leq |S| \leq 2k-1 \quad \text{for all } S \in \gamma \tag{4.4}$$

and

$$ANON(\gamma) = \sum_{S \in p} |S| \cdot d(S) \tag{4.5}$$

inequality (4.3) follows from (4.5), (4.4) and (4.1). □

The following theorem [13, 7] asserts that given an α-approximation algorithm for the $[k, 2k-1]-$minimum clustering problem, the k-anonymization problem can be approximated within a factor of 2α.

Theorem 4.2.2. *Let $\alpha \geq 1$, and let γ be a $[k, 2k-1]$-clustering with cost at most α times that of an optimal solution to the $[k, 2k-1]$-minimum clustering problem. Then the algorithm that anonymizes each $S \in \gamma$ by replacing each $R \in S$ with $\overline{S}$ (the closure of S with respect to generalization, Definition 4.1.1) is a $2\alpha-$approximation algorithm to the problem of optimal k- anonymization.*

Proof. Let γ_{OPT} be an optimal solution to the $[k, 2k-1]$-minimum clustering problem. Let γ be another $[k, 2k-1]-$clustering with generalization cost $d(\cdot)$ no greater than α times that of γ_{OPT}. In addition, let γ^* be an optimal $k-$anonymization and $OPT(D) = ANON(\gamma^*)$, its anonymization cost. Since, as argued earlier, γ^* may be assumed to be a $[k, 2k-1]$-clustering, we may argue as follows:

$$\begin{array}{ll}
ANON(\gamma) = \sum_{S\in\gamma} |S| \cdot d(S) & \\
\leq \sum_{S\in\gamma} (2k-1) \cdot d(S) & \text{(since } |S| \leq 2k-1 \text{ for all } S \in \gamma\text{)} \\
= (2k-1) \cdot d(\gamma) & \text{(by the definition of the function } d\text{, (4.1))} \\
\leq \alpha \cdot (2k-1) \cdot d(\gamma_{OPT}) & \text{(since } d(\gamma) \leq \alpha \cdot d(\gamma_{OPT})\text{)} \\
\leq \alpha \cdot (2k-1) \cdot d(\gamma^*) & \text{(since } d(\gamma_{OPT}) = \min_\gamma d(\gamma) \leq d(\gamma^*)\text{)} \\
\leq \alpha \frac{2k-1}{k} \cdot OPT(D) & \text{(by Lemma 4.2.1, } OPT(D) = ANON(\gamma^*) \geq k \cdot d(\gamma^*)\text{)} \\
< 2\alpha \cdot OPT(D) &
\end{array}$$

□

We now proceed to describe the approximation algorithm of [7] to the problem of k-anonymization with minimal loss of information. The two main procedures in that algorithm are described in Algorithms 1 and 2. Algorithm 1, which will be referred to hereinafter as Gen-Cover, is the well-known greedy algorithm for approximating the weighted set cover problem. It receives as input a table D and a collection C of subsets of D. It outputs a cover of D with subsets from C, that approximates an optimal cover to within $O(\ln \kappa(C))$, where $\kappa(C) := \max\{|S|, S \in C\}$. The approximation algorithm for the optimal k-anonymity problem, Algorithm 3, starts by invoking Gen-Cover with the special collection

$$C = F_{[k,2k-1]} := \{S \subset D : k \leq |S| \leq 2k-1\}\,. \tag{4.6}$$

Consequently, the resulting cover, γ, is an $O(\ln k)$-approximation to the problem of optimal $[k, 2k-1]$-cover. In the second phase, Algorithm 3 invokes Algorithm 2 which translates the cover γ to a $[k, 2k-1]$-clustering γ^0. Finally, that clustering is translated into the corresponding k-anonymization of D.

Theorem 4.2.3. *The $k-$anonymization g, that is produced by Algorithm 3, satisfies*

$$\Pi(D, g(D)) \leq 2(1 + \ln 2k) \cdot OPT(D), \tag{4.7}$$

where $OPT(D)$ is the cost of an optimal $k-$anonymization.

Proof. Let $\gamma \in \Gamma_{[k,2k-1]}$ be the cover that is produced by Algorithm 1 (Gen-Cover), and let γ_{OPT} be the optimal cover (one that minimizes $d(\cdot)$ in $\Gamma_{[k,2k-1]}$). As the greedy algorithm approximates the optimal solution of the set-cover problem to within a factor of $(1 + \ln 2k)$, we have $d(\gamma) \leq (1 + \ln 2k) \cdot d(\gamma_{OPT})$.

Algorithm 1: k-anonymization via set-cover: Phase 1

```
Input: Table $D$, integer $k$, collection of subsets $C \subseteq \mathcal{P}(D)$.
Output: Cover $\gamma$ of $D$ with subsets from $C$
1: Set $\gamma = \emptyset$ and $E = \emptyset$.
2: while $E \neq D$ do
3:   for all $S \in C$ do
4:     compute the ratio $\rho(S) = \frac{d(S)}{|S \cap (D \setminus E)|}$.
5:   end for
6:   Choose $S$ that minimizes $\rho(S)$.
7:   $E = E \cup S$, $\gamma = \gamma \cup \{S\}$, $C = C \setminus \{S\}$.
8: end while
```

Algorithm 2: k-anonymization via set-cover: Phase 2

```
Input: Cover $\gamma = \{S_1, \ldots, S_t\}$ of $D = \{R_1, \ldots, R_n\}$.
Output: A $[k, 2k-1]$-clustering, $\gamma^0$ of $D$.
1: Set $\gamma^0 = \gamma$.
2: while $\gamma^0$ has intersecting subsets do
3:   Let $S_j, S_l \in \gamma^0$ be such that $S_j \cap S_l \neq \emptyset$ and let $R \in S_j \cap S_l$
4:   if $|S_j| > k$ then
5:     $S_j = S_j \setminus \{R\}$
6:   else if $|S_l| > k$ then
7:     $S_l = S_l \setminus \{R\}$
8:   else $\{(|S_j| = |S_l| = k)\}$
9:     remove $S_l$ and $S_j$ from $\gamma^0$ and replace them with $S_j \cup S_l$.
10:  end if
11: end while
```

Algorithm 3: k-anonymization via set-cover

```
Input: Table $D$, integer $k$.
Output: Table $g(D)$ that satisfies $k-$anonymity
1: Invoke Algorithm 1 with $C = F_{[k,2k-1]}$, (4.6).
2: Convert the resulting $[k, 2k-1]$-cover $\gamma$ into a $[k, 2k-1]$-clustering, $\gamma^0$,
   by invoking Algorithm 2.
3: Output the $k$-anonymization $g(D)$ of $D$ that corresponds to $\gamma^0$.
```

Now, let γ^0 be the clustering that is achieved by Algorithm 2. In each iteration of the algorithm, it performs one of two possible actions - either the deletion of a record from a subset of the cover, or the unification of two intersecting subsets. As implied by our monotonicity assumption (Definition 3.3.1) and by Lemma 4.1.1, neither of these operations increases the cost of the cover, so we have $d(\gamma^0) \leq d(\gamma)$. To summarize, we get that $d(\gamma^0) \leq (1 + \ln 2k) \cdot d(\gamma_{OPT})$. Finally, by Theorem 4.2.2, the resulting $k-$anonymization satisfies (4.7). $\square$

Meyerson and Williams [13] used the same algorithm for finding a $k-$anonymization that approximates the optimal one, in the case of generalization by suppression. However, their choice of cost measure led them to an approximation factor of $3k \cdot (1 + \ln 2k)$. Their cost measure was

$$d(S) = max_{R,R' \in S} d(\{R, R'\}),$$

where

$$d(\{R, R'\}) = |\{1 \leq j \leq r : R(j) \neq R'(j)\}|,$$

as opposed to the choice made in [7] that was

$$d(S) = |\{1 \leq j \leq r : \exists R, R' \in S \text{ for which } R(j) \neq R'(j)\}|;$$

it is the latter choice of cost function that enabled the improvement of the approximation ratio by a factor of $O(k)$.

Algorithm 3 has an impractical runtime because of its first phase, Gen-Cover. The runtime of Gen-Cover is $O(|C||D|)$ where C is the input collection of subsets from which a cover is to be selected. Since Algorithm 3 invokes Algorithm 1 with an input collection of size $|C| = O(n^{2k-1})$, we end up with an impractical runtime of $O(n^{2k})$.

4.2.1 k-anonymity using frequent itemsets

To improve the running time of Algorithm 3, Park and Shim [15] introduced the usage of frequent itemsets. Their algorithm runs faster by restricting the size of the collection C used in Gen-Cover (Algorithm 1), while still guaranteeing the approximation ratio of $2(1 + \ln 2k)$. They considered the case of generalization by suppression only. One of the contributions of this thesis is the extension of their algorithm for more general types of generalization.

Definition 4.2.2. *Let S be a set of records from table D. We define $d(S)$ as the number of attributes with multiple distinct values in the record set S, namely:*

$$d(S) = \left|\{1 \leq j \leq r : \exists R, R' \in S, R(j) \neq R'(j)\}\right|,$$

where $R(j)$, $R'(j)$ are the values of the jth attribute of the records R, R' respectively.

Let $D = \{R_1, \ldots, R_n\}$ be a table with r quasi-identifiers. Let S be a subset of records from D. If the number of suppression attributes of $S \subseteq D$ is p (i.e $d(S) = p$), then all records in S coincide in the other $(r-p)$ attributes. Without loss of generality, we can assume that the attributes $A_1, \ldots, A_p$ are the suppression attributes of S. The *representative* of S is defined as the set consisting of only the values in the other $(r-p)$ attributes $A_{p+1}, \ldots, A_r$ in S. There is a close relationship between the representative of S and frequent itemsets in association rule mining. An itemset in this setting is the set of values belonging to the representative of a group of records S. The *support* of an itemset is defined as the subset of records in D that contain the same values for the attributes appearing in the itemset. Given the anonymity parameter k, an itemset is called frequent if its support size is at least k. Thus, if $S \subseteq D$ has size of at least k, then its representative r_S is a frequent itemset. Frequent itemsets with a minimum support size of k are desirable candidates for C, given as input to Gen-Cover (Algorithm 1). Let f denote a frequent itemset in D with a minimum support size of k. Let $S(f)$ be the subset of records in D that contain f (the support of f). Let F_{FQ} be the collection consisting of all $S(f)$s (the supports of all frequent itemsets) together with the whole record set, D (that may be viewed as the support of $f = \emptyset$). F_{FQ} may be a multiset because two frequent itesets may have the same support set. Supplying F_{FQ} as an input to Gen-Cover, instead of the collection C of all subsets of sizes between k and $2k-1$, reduces Gen-Cover's complexity from $O(|C||D|) = O(n^{2k})$ to $O(|F_{FQ}||D|)$. Park and Shim [15] go on to prove that the possible solutions obtained by passing F_{FQ} to Gen-Cover are always possible solutions in the original version of Gen-Cover where the input collection of subsets consists of all subsets of records whose size is in the range $[k, 2k-1]$. They thus conclude that optimal k-anonymization may be approximated to within $2(1 + \ln 2k)$ in time $O(|F_{FQ}||D|)$.

Since the number of frequent itemsets may be very large, some proposals have been made to generate only a concise representation of frequent itemsets from which all frequent itemsets can be produced.

Definition 4.2.3. *A frequent itemset is called closed if there exists no proper super-set with the same support.*

In practice, a lot of frequent itemsets are not closed. Namely, if we let F_{CF} denote the support sets of all closed frequent itemsets, then usually $|F_{CF}| \ll |F_{FQ}|$. It is shown in [15] that the use of closed frequent itemsets support sets F_{CF}, instead of frequent itemsets support sets F_{FQ} as input to Gen-Cover still guarantees the same approximation ratio, but its running time is reduced to $O(|F_{CF}||D|)$.

4.3 $k-$anonymization via the forest algorithm

Let $D = \{R_1, \ldots, R_n\}$ be a table having public attributes A_j, $1 \le j \le r$, and assume that all collections of subsets $\overline{A}_j$, $1 \le j \le r$ are proper. Such a table may be represented by a graph. The forest algorithm relies on the graph representation of the table D.

Definition 4.3.1. *The graph representation for the table $D = \{R_1, \ldots, R_n\}$ is the complete weighted graph $G = (V, E)$ where $V = D$, $E = \{e_{i,j} = \{R_i, R_j\} : 1 \le i < j \le n\}$, and $w(e_{i,j}) = d(\{R_i, R_j\})$, where $d(\cdot)$ is the generalization cost which corresponds to the measure in use,* Π.

Let D be the table to be anonymized and let $G = (V, E)$ be its graph representation. A spanning forest $\mathcal{F}$ of a graph $G = (V, E)$ is a partition of V to disjoint trees, namely every $v \in V$ belongs to exactly one tree, $\mathcal{T}_v \in \mathcal{F}$. If all trees in $\mathcal{F}$ are of size at least k, it induces a $k-$anonymization of D, where all records that belong to the same tree are anonymized in the same manner by replacing them with the closure of the subset of records in that tree. We proceed to describe the forest algorithm [1, 12] which has an approximation factor of $O(k)$, and consists of two phases. In section 4.3.1 we prove some basic results that we need for the approximation factor proof. Then, in Sections 4.3.2 and 4.3.3 we describe the two phases of the algorithm.

4.3.1 Preliminaries

Let G be a graph representation of a given table D, and let $\mathcal{F} = \{\mathcal{T}_1, \ldots, \mathcal{T}_s\}$ be a spanning forest of G. In the case that $k \le |\mathcal{T}_i|$ for each $1 \le i \le s$ then the forest induces a $k-$anonymization. The cost of this $k-$anonymization $g(D)$ that corresponds to $\mathcal{F}$ is:

$$\Pi(D, g(D)) = \sum_{j=1}^{s} |\mathcal{T}_j| \cdot d(\mathcal{T}_j), \tag{4.8}$$

where $d(\cdot)$ is the generalization cost function as described in Definition 4.1.2.

Lemma 4.3.1. *Let $\mathcal{F}$ be a spanning forest of G, which corresponds to some anonymization of table D. Then for each $\mathcal{T} \in \mathcal{F}$,*

$$d(\mathcal{T}) \leq w(\mathcal{T}) = \sum_{e \in \mathcal{T}} w(e).$$

Proof. We prove the claim by induction on the size of $\mathcal{T}$. If $|\mathcal{T}| = 1$ then $d(\mathcal{T}) = w(\mathcal{T}) = 0$. If $|\mathcal{T}| = 2$ then $\mathcal{T}$ has exactly one edge, therefore by Definition 4.3.1, $d(\mathcal{T}) = w(\mathcal{T})$. Now let $|\mathcal{T}| = n > 2$, and assume that the claim holds for all trees of size smaller than n. Then there exist two subtrees $\mathcal{T}_1, \mathcal{T}_2 \subset \mathcal{T}$, whose union is $\mathcal{T}$ and whose intersection contains exactly one node. By sub-additivity, Lemma 4.1.1,

$$d(\mathcal{T}) = d(\mathcal{T}_1 \cup \mathcal{T}_2) \leq d(\mathcal{T}_1) + d(\mathcal{T}_2)\,.$$

By the induction hypothesis for both $\mathcal{T}_1$ and $\mathcal{T}_2$,

$$d(\mathcal{T}_1) + d(\mathcal{T}_2) \leq w(\mathcal{T}_1) + w(\mathcal{T}_2)\,.$$

Finally, since $\mathcal{T}_1$ and $\mathcal{T}_2$ have no edges in common,

$$w(\mathcal{T}_1) + w(\mathcal{T}_2) = w(\mathcal{T}),$$

thus completing the proof. □

Theorem 4.3.2. *Let OPT be the cost of an optimal $k-$anonymization of D with respect to some measure of loss of information Π, and let L be an integer such that $L \geq k$. Let $\mathcal{F} = \{\mathcal{T}_1, \ldots, \mathcal{T}_s\}$ be a spanning forest of G such that:*

(a) $|\mathcal{T}_j| \geq k$, $1 \leq j \leq s$,

(b) The total weight of $\mathcal{F}$ is at most OPT,

(c) $|\mathcal{T}_j| \leq L$, $1 \leq j \leq s$.

Then the $k-$anonymization $g_{\mathcal{F}}$ which is induced by the forest $\mathcal{F}$, is an $L-$approximation for the optimal $k-$anonymization, i.e.,

$$\Pi(D, g_{\mathcal{F}}(D)) \leq L \cdot OPT.$$

Proof. By (4.8) and Lemma 4.3.1, and since all trees are of size at most L, we get

$$\Pi(D, g_{\mathcal{F}}(D)) = \sum_{j=1}^{s} |\mathcal{T}_j| \cdot d(\mathcal{T}_j) \leq \sum_{j=1}^{s} L \cdot w(\mathcal{T}_j) \leq L \cdot OPT,$$

as required. □

In the first phase of the forest algorithm we find a spanning forest $\mathcal{F}$ of G, whose total weight is at most OPT, and in which all trees are of size at least k. Then, in the second phase we decompose all components of the forest of size greater than $L = 3(k-1)$ into smaller components of size at least k, without increasing the total weight of the forest. By repeating this procedure sufficiently many times, we arrive at a forest that satisfies the three conditions in Theorem 4.3.2. That forest induces a $k-$anonymization that approximates the optimal one to within a factor of $L = 3(k-1)$.

4.3.2 Phase 1: Creating the initial forest

The first phase is described in Algorithm 4. That algorithm constructs a directed forest in which all trees are directed towards the roots. In each iteration we pick a node $R \in \mathcal{T}$ whose out-degree is 0, that belongs to a tree whose size is smaller than k (i.e.,$|\mathcal{T}| < k$). We connect R to one of its $k-1$ closest neighbors in G, say R', with the restriction that $R' \notin \mathcal{T}$ (otherwise, $\mathcal{T}$ would no longer be a directed tree). Since at this stage $|\mathcal{T}\backslash\{R\}| \leq k-2$, such a node R' must exist. Therefore, for each edge (R, R') in the resulting forest, it is guaranteed that R' is one of the $k-1$ closest neighbors of R.

Algorithm 4: Forest: Forest construction

```
Input: Table $D$, integer $k$.
Output: A forest $\mathcal{F}$ that satisfies conditions (a),(b) in Theorem
4.3.2.
1: Initialize $\mathcal{F} = (V, E)$ where $V = D$ and $E = \emptyset$.
2: while there exists a component $\mathcal{T}$ of $\mathcal{F}$ of size less than $k$ do
3:    Find a node $R \in \mathcal{T}$ without any outgoing edges.
4:    Find a node $R'$ outside $\mathcal{T}$ that is one of the $k-1$ nearest neighbors
      of $R$.
5:    Add the directed edge $(R, R')$ to $E$.
6: end while
```

Lemma 4.3.3. *Algorithm 4 terminates with a forest $\mathcal{F}$, which has a minimum tree size of k and a weight at most OPT .*

Proof. The algorithm starts up with the trivial forest which contains $|V|$ singleton trees and no edges. In every iteration a single edge is added that connects two of the trees. Therefore, the forest structure is preserved throughout the algorithm. It is evident from the algorithm description that each component of the forest it produces has at least k vertices. In order to prove that $\mathcal{F}$ has a total weight at most OPT, we first note that each node $R \in \mathcal{F}$ has at most one outgoing edge. If we show that the weight of each such edge, denoted (R, R'), is at most the generalization cost of the record R in the optimal $k-$anonymization, then by summing up for all edges in $\mathcal{F}$ we get that $w(\mathcal{F}) \leq OPT$. Let S be the cluster of R in the optimal $k-$anonymization, and let $R^{''}$ be the $(k-1)$th closest record to R in S. Since R' is, in the worst case, the $(k-1)$th closest record to R in the whole table D, we have $w(e_{(R,R')}) \leq w(e_{(R,R'')})$. By monotonicity, $w(e_{(R,R')}) = d(\{R, R'\}) \leq w(e_{(R,R'')}) = d(\{R, R''\}) \leq d(S)$, (because $S \supseteq \{R, R''\}$), leading to the desired inequality $w(e_{(R,R')}) \leq d(S)$. □

4.3.3 Phase 2: Decomposing large trees

The second phase of the $k-$anonymization algorithm turns the forest which was constructed in Algorithm 4 into a forest that satisfies also condition (c) in Theorem 4.3.2, for $L = 3(k-1)$. To that end, we define the following terms.

A component $H \subseteq V$ in the graph $G = (V, E)$ will be called

- legal, if $|H| \geq k$;
- large, if $|H| > L$;
- small, if $k \leq |H| \leq L$.

Let $\mathcal{T}$ be a large tree in G. Algorithm 5 breaks it into two legal components, which may take one of the following forms:

- (F1) Two legal trees.
- (F2) A small forest and a legal tree.

A large tree which is produced by the algorithm will be subjected to it recursively, until we are left with only small components.

Lemma 4.3.4. *Let j be the index that is chosen in Step 3 of Algorithm 5. If $j = 1$ then S_j is a legal tree and so is $\mathcal{T} \backslash S_j$ (form (F1) above). Otherwise it is a small forest and $\mathcal{T} \backslash S_j$ is a legal tree (form (F2) above).*

Proof. If $j = 1$ then $S_1 = \mathcal{T}_1$ (namely, it is a tree) and, by the definition of j, we have $|S_1| = s_1 \geq k$ (namely, it is legal). Since Step 3 is activated only when $s - s_1 \geq k$, we infer that in this case also $\mathcal{T} \backslash S_1$ is a legal tree.

Algorithm 5: Forest: Decompose trees

```
Input: A large tree $\mathcal{T}$ of size $s$, an integer $k$.
Output: Decomposition of $\mathcal{T}$ to one of the forms (F1) or (F2).
1: Pick a vertex $u \in \mathcal{T}$ and root the tree at that vertex. Let $l$ be the
   number of children of $u$; let $\mathcal{T}_1, \ldots, \mathcal{T}_l$ be the subtrees rooted in those
   children, and $s_i = |\mathcal{T}_i|$, $1 \le i \le l$. Without loss of generality, assume
   that $s_1 = max\{s_1, \ldots, s_l\}$.
2: if $s - s_1 \ge k$ then
3:    output the following trees: $S_j$ and $(\mathcal{T} \backslash S_j)$, where $S_j = \bigcup_{i=1}^{j} \mathcal{T}_i$ and $j$
      is the minimal index such that $\sum_{i=1}^{j} s_i = k$.
4: else
5:    replace $u$ with the root of $\mathcal{T}_1$ and return to Step 1.
6: end if
```

If $j > 1$, we infer that $s_i < k$ for all $1 \le i \le l$. As j is set to be the minimal index for which $\sum_{i=1}^{j} s_i \ge k$ (i.e , the minimal j for which S_j is legal), we conclude that $k \le |S_j| = \sum_{i=1}^{j} s_i \le 2k - 2$. Hence, S_j is legal and small. As S_j is a unification of subtrees, it is an unconnected forest. Finally, $\mathcal{T} \backslash S_j$ is a legal tree, since $\mathcal{T}$ is a large tree, therefore $|\mathcal{T} \backslash S_j| \ge (L+1) - |S_j| = [3(k-1)+1] - (2k-2) = k$ □

Lemma 4.3.5. *Algorithm 5 terminates.*

Proof. We prove the statement by showing that in a subsequent iteration of the loop in the subroutine, either we reach a stage in which it stops (i.e., the stopping criterion in Step 2 is satisfied), or the size of the largest subtree, s_1, decreases.

Assume that the loop did not stop when the root of the tree was u. Let v be the root of the largest subtree $\mathcal{T}_1$. Then $s - s_1 \le k - 1$. In the next iteration, when v becomes the root of the whole tree, and u becomes a child of v, there are two possibilities:

Case 1: The subtree rooted at u is the largest subtree under v. The size of that subtree is, by assumption, at most $k - 1$. In that case, as $s \ge L + 1 = 3(k-1) + 1 = 3k - 2$, then the stopping criterion in Step 2 is satisfied since $3(k-1) + 1 - (k-1) = 2k - 1 \ge k$.

Case 2: The largest subtree under v is one of the subtrees that were under v in the previous loop. In that case the value of s_1 must be smaller than its value in the previous iteration (in which the largest subtree was rooted at v). □

Let $\mathcal{F}$ be the forest that is created by Algorithm 4. We have shown that by activating Algorithm 5 repeatedly on all of the large trees in $\mathcal{F}$, we get a partition into legal and small components, $(\bigcup \mathcal{A}_i) \cup (\bigcup \mathcal{B}_i)$, where the components $\mathcal{A}_i$ are trees, and the components $\mathcal{B}_i$ are forests. Each of the forests consists of trees that were connected through a common parent in $\mathcal{F}$. Let $\hat{\mathcal{B}}_i$ be the tree that is obtained from $\mathcal{B}_i$ by connecting all trees in $\mathcal{B}_i$ through a common parent, in the same way that they were connected in $\mathcal{F}$, but instead of the original parent u (which is not part $\mathcal{B}_i$), we will use a Steiner vertex u'. This Steiner vertex is, essentially, a dummy vertex, which neither contributes to the size of its component nor to the weight of its component due to the added edges. By applying that to all forests $\hat{\mathcal{B}}_i$, we get the forest $\mathcal{F}' = (\bigcup \mathcal{A}_i) \cup (\bigcup \hat{\mathcal{B}}_i)$. Since the set of edges of $\mathcal{F}'$ is contained in the set of edges of $\mathcal{F}$ we have $w(\mathcal{F}') \leq w(\mathcal{F})$.

Thus, the conditions in Theorem 4.3.2 are all satisfied by the forest $\mathcal{F}'$, implying that the $k-$anonymization induced by $\mathcal{F}'$ is a $3(k-1)$-approximation for the optimal $k-$anonymization.

The overall running time of the forest algorithm may be shown to be $O(kn^2)$.

Chapter 5

Heuristic Algorithms for $k-$Anonymization

In this chapter we present a heuristic algorithm for the $k-$anonymization problem, called the agglomerative algorithm [6]. Experiments show that this algorithm outperforms, in practice, the Forest algorithm from Section 4.3.

Given a table $D = \{R_1, \ldots, R_n\}$ and an integer $k > 1$, we compute a clustering of D, $\gamma = \{S_1, \ldots, S_m\}$, such that $|S_i| \geq k$ for all $1 \leq i \leq m$. The algorithm assumes a distance function, $dist(\cdot, \cdot)$, between subsets of D, i.e., $dist : \mathcal{P}(D) \times \mathcal{P}(D) \to \mathbb{R}$.

The basic agglomerative algorithm, Algorithm 6, starts with singleton clusters and then keeps unifying the two closest clusters until they mature into clusters of size at least k. As it may produce clusters of size greater than k, while it is preferable to have clusters of size k or close to k in order to reduce the clustering anonymization cost, another variant, called the modified agglomerative algorithm, is proposed in [6]. Algorithm 7 describes how to replace line 12 of Algorithm 6 in order to achieve that goal. Essentially, before moving a "ripe" cluster $\hat{S}$ to the final clustering γ, we shrink it to a sub-cluster of size k.

Finally, the clustering of D that is produced by either of the above agglomerative algorithms is translated into a corresponding $k-$anonymization $g(D)$, by replacing every record $R_i \in D$ with the closure of the cluster to which R_i belongs.

A key ingredient in the agglomerative algorithms is the definition of distance between clusters. It is natural to define the distance so it best fits the cost function of the $k-$anonymization. Since all records in a given cluster are replaced by the same generalized record, we have

Algorithm 6: Basic Agglomerative algorithm

```
Input: Table $D$, integer $k$.
Output: Table $g(D)$ that satisfies $k-$anonymity
 1: for all $R_i \in D$ do
 2:   create a singleton cluster $\hat{S}_i = \{R_i\}$ and let $\hat{\gamma} = \{\hat{S}_1, \ldots, \hat{S}_n\}$.
 3: end for
 4: Initialize the output clustering $\gamma$ to $\emptyset$.
 5: while $|\hat{\gamma}| > 1$ do
 6:   Find the "closest" two clusters in $\hat{\gamma}$, namely, the two clusters
      $\hat{S}_i, \hat{S}_j \in \hat{\gamma}$ that minimize $dist(\hat{S}_i, \hat{S}_j)$ .
 7:   Set $\hat{S} = \hat{S}_i \cup \hat{S}_j$.
 8:   Remove $\hat{S}_i$ and $\hat{S}_j$ from $\hat{\gamma}$.
 9:   if $|\hat{S}| < k$ then
10:     add $\hat{S}$ to $\hat{\gamma}$.
11:   else
12:     add $\hat{S}$ to $\gamma$.
13:   end if
14: end while [At this stage, $\hat{\gamma}$ has at most one cluster,
    $\hat{S} = \{R_{i_1}, \ldots, R_{i_l}\}$, the size of which is $l < k$]
15: For each record $R_{i_j}$, $1 \le j \le l$, add that record to the cluster $S \in \gamma$
    that minimizes $dist(\{R_{i_j}\}, S)$.
```

Algorithm 7: Modification of line 12 of Algorithm 6

```
Input: $\hat{S} = \{\hat{R}_1, \ldots, \hat{R}_l\}$ where $l \ge k$
Output: Sub-cluster of $\hat{S}$ of size $k$.
1: while $|\hat{S}| > k$ do
2:   for all $1 \le i \le l$ do
3:     compute $d_i = dist(\hat{S}, (\hat{S} \backslash \{\hat{R}_i\}))$.
4:     Find the record $\hat{R}_i$ that maximizes $d_i$.
5:     Remove $\hat{R}_i$ from $\hat{S}$ and add the corresponding singleton cluster
       $\{\hat{R}_i\}$ to $\hat{\gamma}$.
6:   end for
7: end while
8: Place the shrunk cluster $\hat{S}$ in the final clustering $\gamma$.
```

$$\Pi(D, g(D)) = \sum_{S \in \gamma} |S| \cdot d(S)\,. \tag{5.1}$$

Several distance functions are discussed and compared in [6]. We used the first function that was proposed there, which is

$$dist(A, B) = |A \cup B| \cdot d(A \cup B) - |A| \cdot d(A) - |B| \cdot d(B)\,. \tag{5.2}$$

In view of (5.1), the quantity $dist(A, B)$ expresses the change in the overall distance $\Pi(D, g(D))$ that results from the unification of the two clusters A and B.

Chapter 6

Frequent Itemset Mining Algorithms

Efficient algorithms for mining frequent itemsets are crucial for mining association rules as well as for many other data mining tasks. Frequent itemset mining leads to the discovery of associations and correlations among items in large transactional or relational data sets. The discovery of interesting correlations among huge amounts of business transaction records can help in many business decision-making processes.

6.1 The frequent itemset mining problem

Let $M = \{I_1, I_2, \ldots, I_m\}$ be a set of items. Let $D = \left\langle T_1, T_2 \cdots T_n \right\rangle$ be a table of transactions, where $T_i = (tid, I)$ is a transaction with identifier *tid* which contains a set of items I such that $I \subseteq M$. The support of a set of items, $I \subseteq M$, is the set of transactions in D that contain I.

I is frequent if I's support size is no less than a minimum support threshold *min_sup*. Given a transaction table D and a minimum support threshold *min_sup*, the problem of finding the complete set of frequent itemsets (FI) is called the frequent itemset mining problem.

Frequent itemsets are *downward closed* in the itemset lattice, meaning that any subset of a frequent itemset is frequent.

A major challenge in mining frequent itemsets from a large or dense dataset, is the fact that such mining often generates a huge number of frequent itemsets satisfying the minimum support threshold, especially when *min_sup* is set low.

For example, if there is a frequent itemset with size l, then all $2^l - 1$ nonempty subsets of that itemset are also frequent, and hence have to be generated. This can potentially be too huge a number for any computer to compute or store. To overcome this difficulty, we introduce the concepts of *closed frequent itemsets* and *maximal frequent itemsets* [8].

A frequent itemset X is called *maximal* if there does not exist a frequent itemset Y such that $Y \supset X$. Due to the downward closure property of frequent itemsets, all subsets of maximal frequent itemsets (MFIs) are frequent, while all supersets of such items are infrequent. Therefore, mining frequent itemsets can be reduced to mining the MFIs, which may be viewed as the "boundary" of the itemset lattice. However, mining only MFIs has the following deficiency: Given an MFI and its support size s, we know that all its subsets are frequent and the support of any of its subsets is of size at least s; however, we do not know exactly what is that support. For several purposes, one of which is the generation of association rules, we do need to know the support of all frequent itemsets.

To solve this problem, another type of frequent itemset, called *closed frequent itemset* (CFI), is proposed. A frequent itemset X is closed if none of its proper supersets have the same support. For any frequent itemset, there exists a single closed frequent itemset that includes it and has the same support. CFIs are lossless in the sense that they uniquely determine the set of all frequent itemsets and their exact support. At the same time, the set of CFIs can be orders of magnitude smaller than the set of FIs, especially in dense tables. The relationship among these groups is:

$$FI \supseteq CFI \supseteq MFI\,.$$

The problem of mining frequent itemsets was first introduced by Agrawal et al. [3]. They proposed an algorithm called Apriori for mining frequent itemsets. Apriori employs a bottom-up, breadth-first search that enumerates every single frequent itemset. Apriori uses the downward closure property of itemsets in order to prune the search space. Thus, only the frequent $k-$itemsets are used to construct candidate $(k+1)-$itemsets. The algorithm starts by finding all itemsets of size 1 that are frequent, and then generates itemsets of size 2 that are candidate to being frequent. It then scans the table in order to detect which of the candidates is indeed a frequent itemset. Frequent itemsets of size 2 are joined in order to create candidate itemsets of size 3. Another table scan is issued in order to find the frequent itemsets among the candidates. This procedure is repeated until no more frequent itemsets are found.

Apriori uses hash-trees to store frequent itemsets and candidate frequent itemsets. These hash-trees are also used for frequent itemset generation and subset testing.

An Apriori-like algorithm may suffer from the following two non-trivial costs:

- It is costly to handle a huge number of candidate sets. For example, if there are 10^4 frequent 1-itemsets, the Apriori algorithm will need to generate more than 10^7 length-2 candidates and test their support sizes. Moreover, to discover a frequent pattern of size 100, it must generate and examine $2^{100} - 2 \approx 10^{30}$ candidates in total.
- When the size of the largest frequent itemset is l, Apriori will need l table scans.

In [9], Han et al. introduced a novel algorithm, known as the FP-Growth method for mining frequent itemsets. The FP-Growth method is a depth-first search algorithm. In that method, a data structure called the FP-tree is used for storing frequency information of the original table in a compressed form. Only two table scans are needed for the algorithm and no candidate generation is required. This makes the FP-Growth method much faster than Apriori. We proceed to describe it.

6.2 The FP-growth algorithm

The FP-Growth algorithm utilizes a structure that is called the FP-tree (frequent pattern tree), which is a compact representation of all relevant frequency information in the table. Every branch of the FP-tree represents a frequent itemset and every node along those branches represents an item in the itemset. The nodes along the branches are stored in decreasing order of frequency of the corresponding items, where the leaves represent the least frequent items. Compression is achieved by building the tree such that overlapping itemsets share prefixes of the corresponding branches.

6.2.1 FP-tree construction

The FP-Growth method performs two table scans in order to construct the FP-tree (as opposed to Apriori and its variants that need as many table scans as the length of the longest frequent itemset). After constructing the FP-tree, the algorithm uses it in order to mine the frequent itemsets.

Transaction ID	Items in transaction
1	a, c, d, e, f
2	a, b, e
3	c, e, f
4	a, c, d, f
5	c, e, f

Table 6.1: Example transaction DB

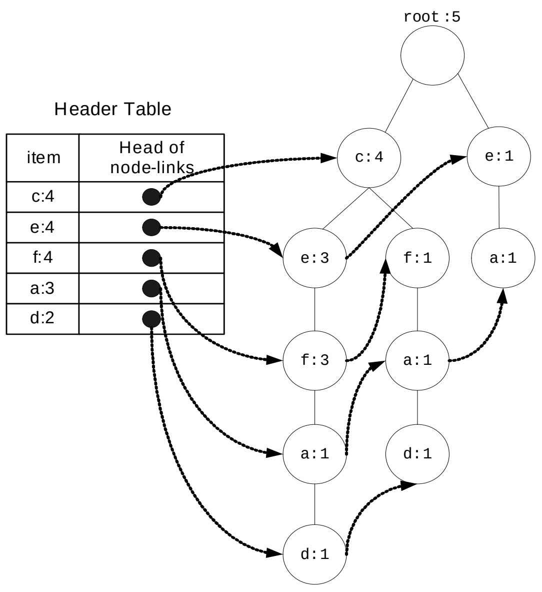
The first table scan finds all frequent items. These items are inserted into an item header table in decreasing order of their support sizes. In the second scan of the table, as each transaction is scanned, the frequent items in it are sorted according to their order in the header table (namely, according to their support size), while the infrequent items are removed. Then, the items in the reduced and sorted transaction are inserted into the FP-tree as a branch. If the new itemset shares a prefix with an itemset that already exists in the tree, the part of the branch that corresponds to that prefix will be shared by those itemsets. In addition, there is a counter that is associated with each node of the tree. The counter stores the number of transactions that contain the itemset that is represented by the path from the root to that particular node. The counter is updated during the second scan.

To facilitate tree traversal, the item header table is built such that each item points to its occurrence in the tree via a *head of node-link*. Nodes with the same item-name are linked in sequence via such node-links.

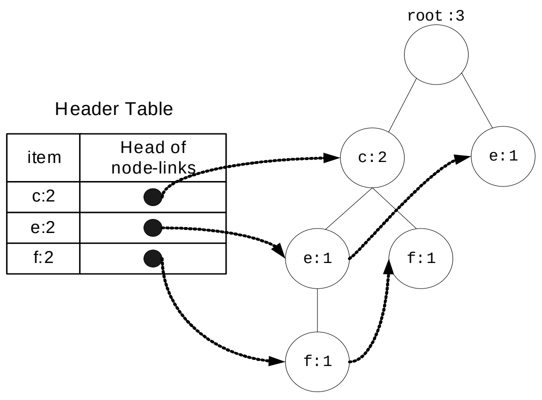
To summarize, the first scan finds frequent items and their support sizes and uses them to fill the header table. The second scan fills the tree with the relevant frequency information which enables, at a later stage, to mine the frequent itemsets. Table 6.1 displays an example transaction table and Figure 6.1(a) illustrates the constructed FP-tree for that table, with $min\_sup = 2$.

6.2.2 FP-Growth, mining FIs from the FP-tree

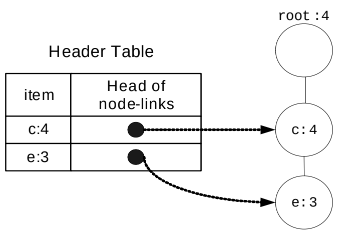
At the completion of the first stage, we have the FP-tree that contains all frequency information about the table. In the second stage, we use that structure in order to mine the frequent itemsets, without needing to scan the table ever again. This is done by a method that is called The FP-Growth. The FP-Growth method relies on the following principle, called the pattern growth property : If X and Y are two itemsets, the count of itemset $X \cup Y$ in the table is exactly that of Y in the restriction of the table



(a) $T_\emptyset$



(b) T_a



(c) T_f

Figure 6.1: Example FP-tree

to those transactions that contain X.

Given an item i in an FP-tree's Header Table, $T.header$, one may visit all branches that contain item i by following the linked list that starts at i in $T.header$. In each branch encountered, the set of nodes on the path from the node representing the item i to the root, form an itemset. The support size of the discovered itemset is i's support size as available in the node representing i in the branch.

The set of itemsets collected in this manner is called the conditional table of itemset $\{i\}$, and is denoted $DB_{\{i\}}$. The FP-tree constructed from the conditional table is called $\{i\}'$s conditional FP-tree, and denoted $T_{\{i\}}$.

The items in the header table are processed from the item with least support size to the item with the highest support size. After an item has been processed, it is disregarded when constructing the conditional tables and trees for the items which come after it in the header table (with higher support size).

For example, in $T_\emptyset$ (Figure 6.1(a)), if we follow item f's linked list, the first branch we encounter is $c : 4, e : 3$ (notice that we disregarded items a and d), therefore the first itemset in DB_f is $c, e : 3$. Moving on to the next node in the linked list, we encounter itemset $c : 4$, therefore the second itemset in DB_f is $c : 1$.

In order to construct the FP-tree $T_{\{i\}}$, two scans as described above need to be conducted. The first scan extracts the itemsets in the conditional table, removes infrequent items, and sorts the rest of the items in support-size descending order. This set of actions constructs $T_{\{i\}}$'s header table. In the example above, we found that the conditional table DB_f contains two itemsets, $c, e : 3$ and $c : 1$, therefore the header table will contain items $c : 4, e : 3$.

Now, that the infrequent items were removed, and the order of the items is established, a second scan of the FP-tree can take place. This time, each itemset found in the FP-tree is processed by removing infrequent items, and sorting the rest of the items according to their order in the header table (initialized in the first scan). After this, the itemset is inserted into the $T_{\{i\}}$ FP-tree. In our example, itemset $c, e : 3$ will be inserted into the tree in that order, and after that, itemset $c : 1$ will be inserted into the tree. The resulting tree, T_f can be seen in Figure 6.1(c).

The FP-tree constructed from the original table is denoted by $T_\emptyset$, and it may be viewed as the conditional FP-tree that corresponds to the empty itemset. The FP-tree constructed from the original table is illustrated in Figure 6.1(a). Figure 6.1(b) displays the conditional FP-tree $T_{\{a\}}$.

The above procedure is applied recursively, and it stops when the resulting new FP-tree contains only a single branch (According to [9], the complete set of frequent itemsets can be generated from all single-branch FP-trees), or when the tree is empty.

Algorithm 8 summarizes the steps for mining the full set of FIs for a given table D. It is a recursive algorithm that accepts two inputs – an itemset X and its corresponding FP-Tree T. We call it with the empty itemset $X = \emptyset$ and the corresponding tree $T = T_\emptyset$ which holds information on the entire table D. The algorithm may call itself with other, nonempty itemsets X, and their corresponding FP-trees. The output of the algorithm is the list of all frequent itemsets in D together with their count.

Algorithm 8: Mining frequent patterns with FP-tree by pattern fragment growth

```
Procedure FPGrowth(itemset $X$, FP-tree $T$)
Input: An itemset $X$ and its corresponding conditional FP-tree $T$
Output: The complete set of all FIs corresponding to $T$
1: if $T$ only contains a single branch $B$ then
2:   for all $Y \subseteq B$ do
3:     Output itemset $Y \cup X$ with count = minimum support-size of
       nodes in $Y$.
4:   end for
5: else
6:   for all $i$ in $T.header$ do
7:     Output $Y = X \cup \{i\}$ together with $i$'s count.
8:     Construct $Y$'s conditional table, and $Y$'s conditional FP_tree, $T_Y$.
9:     if $T_Y \neq \emptyset$ then
10:      call FPGrowth$(Y, T_Y)$
11:    end if
12:  end for
13: end if
```

6.3 Mining closed frequent itemsets

An itemset X is closed if none of the proper supersets of X have the same support. As explained earlier, mining closed frequent itemsets (CFIs) has numerous advantages over mining frequent itemsets. In this section we will survey algorithms designed for mining CFIs. The first algorithm, CLOSET [16] mines CFIs using the FP-tree. The second algorithm surveyed, CHARM

[10], uses a different approach.

6.3.1 Theoretic foundations for mining CFIs from FP-trees

The optimizations applied in algorithm CLOSET [16] aim at identifying CFIs quickly. They are based on the following lemmas which enable to prune the search space as well as to reduce the size of the FP-tree being processed. The proofs of these lemmas are available in [16]. These lemmas and optimizations can be applied to any algorithm for mining CFIs from FP-trees. The use of the optimizations resulting from these lemmas is later illustrated in Example 6.3.1. The CLOSET algorithm is specified in Algorithms 9 and 10.

Lemma 6.3.1. *If X is a closed frequent itemset, then there is no item that appears in every transaction in the X-conditional table.*

Lemma 6.3.2. *If an itemset Y is the maximal set of items appearing in every transaction in the $X-$conditional table, and $X \cup Y$ is not subsumed by some already found closed frequent itemset with identical support size, then $X \cup Y$ is a closed frequent itemset.*

The optimization corresponding to Lemma 6.3.2 can be applied at the item counting phase, after only a single scan of the FP-tree that is being processed. The items in itemset Y should be excluded from both the header table and the conditional table of the conditional FP-tree being constructed. From this point, the conditional table and header table are associated with itemset $X \cup Y$ (as opposed to itemset X). This optimization has the following benefits:

1. It reduces the size of the FP-tree because the conditional table contains itemsets with a smaller number of items after the extraction.

2. It reduces the level of recursions since it combines a few items into one itemset.

Definition 6.3.1. *Let i be a frequent item in the X-conditional table. Assume that the following three conditions are met:*

1. *There is only one node N labeled i in the corresponding FP-tree.*

2. *Every ancestor of N has only one child.*

3. *N has either no children, or one child with a support-size smaller than that of N, or more than one child.*

*Then the i-*single segment itemset *is defined as the union of itemset X and the set of items including i (represented by node N), and the items represented by N's ancestors.*

Lemma 6.3.3. *The $i-$single segment itemset Y is a closed frequent itemset if the support of i within the conditional table passes the given threshold and Y is not a proper subset of any closed frequent itemset already found.*

The above lemma enables to identify CFIs quickly. It reduces the size of the remaining FP-tree to be processed, and reduces the level of recursion since it combines multiple items into one itemset.

Lemma 6.3.4. *Let X and Y be two frequent itemsets with the same support size. If $X \subset Y$, then there exists no closed frequent itemset containing X but not $Y - X$.*

Example 6.3.1 illustrates how the above optimizations are applied. The itemset X is represented as an attribute of T_X, T_X*.base*. T_X*.header* will denote the header table of the FP-tree T_X. The items in T_X*.header* are sorted in descending order of their support size.

Example 6.3.1. *The CLOSET example will be run on transaction table illustrated in table 6.1. After scanning the table, we derive the set of frequent items (or f_list), $f\_list = \langle c:4, e:4, f:4, a:3, d:2 \rangle$. We then continue to construct the FP-tree $T_\emptyset$ for the table (see FP-tree construction procedure in sub-section 6.2.1), the resulting FP-tree can be seen in Figure 6.1(a).*

We now create the conditional table of item $d:2$. According to the FP-tree $T_\emptyset$ in Figure 6.1, $f\_list_d = \{c:2, f:2, a:2\}$. According to Lemma 6.3.2, since c, f and a appear in every transaction in the d-conditional table, then $cfad:2$ is a CFI. After the extraction of these items from f_list_d, it is empty. Therefore, there is no need to perform another scan of $T_\emptyset$ in order to compute DB_d or construct T_d.

We now move on to explore $a:3$'s conditional table, after the first scan of $T_\emptyset$ we have $f\_list_a = \{c:2, e:2, f:2\}$. Since a's support is 3, and there is no item appearing in every transaction in DB_a, and $a:3$ is not subsumed by any CFI with the same support size, then $a:3$ is a CFI. We now perform another scan of $T_\emptyset$ in order to construct $DB_a = \{cef:1, cf:1, e:1\}$. To find the rest of the CFIs containing a, we need to explore a's conditional FP-tree, Figure 6.1(b).

We recursively continue to mine T_a. Therefore we explore CFIs which contain $fa:2$. We can see that since fa is a subset of formerly found CFI

$cfad$, and they both have the same support size, then according to Lemma 6.3.4, there is no need to build or process the fa-conditional table, DB_{fa}.

We move on to to explore CFIs containing $ae : 2$. Since f_list_{ae} is empty, the only potential CFI is $ae : 2$, and since it isn't subsumed by any other CFI with the same support size, it is added the set of CFIs.

For itemset $ca : 2$, we again have that ca is subsumed by $cfad$, and has the same support size. Therefore, ca is not closed, and according to lemma 6.3.4, can be pruned.

Now, we move on to exploring $f : 4$'s conditional table and tree, see Figure 6.1(c). After scanning $T_\emptyset$ we have that $f\_list_f = \{c : 4, e : 3\}$. Since c appears in every transaction that f appears in, f cannot be closed according to Lemma 6.3.1. Itemset $cf : 4$ is a CFI as it is not subsumed by any other CFI with the same support size. At this stage, we can extract item $c : 4$ from f_list_f. Furthermore, we can combine items f and c into one for the rest of the mining process. Therefore, we now have $f\_list_{cf} = \{e : 3\}$. After performing the second scan of $T_\emptyset$ in order to compute DB_{cf}, we have that $DB_{cf} = \{e : 3\}$. Now the tree contains only a single node $e : 3$, and we can easily verify that $cef : 3$ is a CFI.

For item $e : 4$, $f\_list_e = \{c : 3\}$ and $DB_e = \{c : 3\}$. Therefore $e : 4$ is a CFI. Itemset $ec : 3$ is also a potential CFI, but it is not a CFI as it is subsumed by formerly found CFI $cef : 3$, which has the same support size.

Finally we are left with the last itemset $c : 4$, which is not closed because it is subsumed by $cf : 4$.

The full set of CFIs mined throughout this example is

$$cfad : 2, a : 3, ae : 2, cf : 4, cef : 3, e : 4.$$

Algorithm 9 summarizes the steps for mining the full set of CFIs for a given table D. It calls the recursive Closet procedure (procedure 10) that accepts three inputs – an itemset X, its corresponding FP-Tree T and the set of closed frequent itemsets to be updated. We call it with the empty itemset $X = \emptyset$ and the corresponding tree $T = T_\emptyset$ which holds information on the entire table D. The set of closed frequent itemsets is initialized to the empty set. The algorithm may call itself with other, nonempty itemsets X, and their corresponding FP-trees. The output of the algorithm is the list of all closed frequent itemsets in D.

Algorithm 9: CLOSET Algorithm

```
Input:
    1. Transaction table $TDB$.
    2. Support threshold $min\_sup$.
Output: The complete set of closed frequent itemsets.
1: Initialization. Let $FCI$ be the set of closed frequent itemsets.
   Initialize $FCI \leftarrow \emptyset$.
2: Find frequent items. Scan the transaction table $TDB$, find the
   frequent items. Insert the items in support size descending order into a
   list, $f\_list$.
3: Construct the conditional FP-tree for $\emptyset$, $T_\emptyset$.
4: Mine closed frequent itemsets recursively. Call
   $CLOSET(\emptyset, T_\emptyset, FCI)$.
```

Algorithm 10: CLOSET Procedure

Procedure CLOSET(X,T,FCI) **Input**:

1. An itemset, X;
2. The corresponding FP-tree, T;
3. The set of closed frequent itemsets that was already mined, FCI.

Output: Updated FCI

```
1: for all items $i \in T.header$, starting from the one with smallest
   support-size do
2:   if there is only a single node $N$ labeled $i$ in $T$ and the branch $B$
     which contains it abides to the conditions of Lemma 6.3.3 then
3:     if $X \cup B$ is not subsumed by any itemset in $FCI$ with the same
       support size then
4:       $FCI \leftarrow FCI \cup (X \cup B)$
5:     end if
6:     Remove the item $i$ from $T$ and $T.header$.
7:   end if
8: end for
9: for all remaining items $i \in T.header$, starting from the one with
   smallest support-size do
10:  if $X \cup \{i\}$ is not subsumed by any item in $FCI$ with the same
     support size then
11:    Scan $T$ in order to find the list of frequent items which co-occur
       with $i$, $f\_list_i$ .
12:    Sort the items in $f\_list_i$ in support-size descending order.
13:    Let $Y$ be the set of items in $f\_list_i$ such that
       $support\_size(Y) = support\_size(i)$.
14:    $Z \leftarrow X \cup \{i\} \cup Y$.
15:    if $Z$ is not subsumed by any itemset in $FCI$ with the same
       support-size then
16:      $FCI \leftarrow FCI \cup Z$ {Lemma 6.3.2}
17:    end if
18:    Exclude the set of items in $Y$ from the list $f\_list_i$.
19:    Build the conditional table for itemset $Z$, $DB_Z$.
20:    Construct FP-tree $T_Z$ from $DB_Z$.
21:    Call CLOSET($Z, T_Z, FCI$)
22:  end if
23: end for
```

6.3.2 The Charm algorithm

CHARM [10] is also a depth-first CFI mining algorithm, but it does not depend on the FP-tree. The CHARM Algorithm introduces the following ideas:

1. It simultaneously explores both the itemset space and the transaction space. In contrast, most previous methods exploit only the itemset search space.
2. It uses a method that enables to prune the search space of itemsets which cannot lead to undiscovered closed frequent itemsets.
3. It uses a hash-based approach to eliminate non-closed itemsets during subsumption checking.
4. It utilizes a vertical data representation called *diffset* for fast support computations. Diffsets keep track of differences in the support of candidate itemsets from their prefix itemset. They cut down the size of memory required to store the supports of the itemsets.

The set of all transaction ids (tids) in the table will be denoted as $\mathcal{T}$. A set $Y \subseteq \mathcal{T}$ will be referred to as a tidset. For an itemset X, we denote its corresponding tidset as $t(X)$, i.e., the set of all tids that contain X. For a tidset Y, we denote its corresponding itemset as $i(Y)$, i.e., the set of items common to all tids in Y. We can see that $t(X) = \cap_{x \in X} t(x)$, and $i(Y) = \cap_{y \in Y} i(y)$. The notation $X \times t(X)$ is used to refer to an itemset-tidset pair, and it is called an *IT-pair*.

Itemset-TidSet search tree and equivalence classes

Let I be a set of items. Define a function $p(X, k) = X[1 : k]$ as the k-length prefix of X, and a *prefix-based* equivalence relation θ_k on itemsets as follows:

$$\forall X, Y \in I,\ X \equiv_{\theta_k} Y \Longleftrightarrow p(X, k) = p(Y, k).$$

That is, two itemsets are in the same class if they share a common k-prefix.

CHARM performs a search for CFIs over an IT-tree search space, as shown in Figure 6.2 (that corresponds to the table in Table 6.1). Each node in the IT-tree is an itemset-tidset pair $X \times t(X)$, which is also a prefix-based class. All the children of a given node X belong to its equivalence

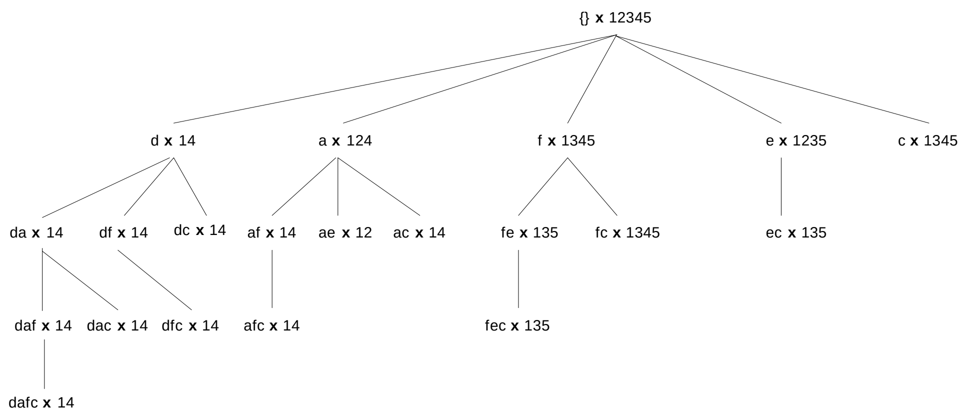


Figure 6.2: IT-Tree:Itemset-tidSet Search tree

class since they share the same prefix X. An equivalence class is denoted by $[P] = \{l_1, \ldots, l_n\}$, where P is the parent node (the prefix), and each l_i is a single item, representing the node $Pl_i \times t(Pl_i)$.

A class is the set of items that the prefix can be extended with to obtain a new itemset-tidset node. Clearly, no subtree of an infrequent prefix needs to be processed. For any subtree rooted at node X, one can treat it as a completely new problem; one can enumerate the patterns under it and simply prefix them with the item X and so on.

Frequent pattern enumeration is done as follows in the IT-tree framework. For a given node, one can perform intersections of the tidsets of all pairs of elements in a class and check if the support threshold (min_sup) is met. Each resulting frequent itemset is represented by a node in the IT-tree, with its own elements, that will be recursively expanded. Algorithm 11 gives a pseudo-code description of a depth-first exploration of the IT-tree for all frequent itemsets. The IT-tree for the frequent items mined from Table 6.1 is illustrated in Figure 6.2.

Mining CFIs using the Itemset-TidSet search tree

The closure of an itemset X, denoted by $c(X)$, is the smallest CFI that contains X. To find the closure of an itemset X we first compute its support, $t(X)$. We next map $t(X)$ to its image in the itemset space, i.e., $i(t(X))$. X is a CFI if and only if $c(X) = i(t(X)) = X$. For example, itemset $\{c, e, f\}$ is closed because $i(t(\{c, e, f\})) = i(\{1, 3, 5\}) = \{c, e, f\}$. The support of an itemset also equals the support of its closure. There are four basic properties

Algorithm 11: Frequent Pattern enumeration using the IT-tree

```
Procedure Enumerate-Frequent
Input: $[P]$ the node, or class to be enumerated
Output: Enumeration of the IT-tree with prefix represented by $[P]$
1: for all $l_i \in [P]$ do
2:   $[P_i] = \emptyset$
3:   for all $l_j \in [P]$ s.t $j > i$ do
4:     $I = Pl_il_j$
5:     $T = t(Pl_i) \cap t(Pl_j)$
6:     if $|T| \geq min\_sup$ then
7:       $[P_i] \leftarrow [P_i] \cup \{I \times T\}$ {new Itemset-TidSet node}
8:     end if
9:   end for
10:  Call Enumerate-Frequent($[P_i]$)
11: end for
```

of IT-Pairs that CHARM leverages for fast exploration of CFIs. Assume that we are currently processing a node $P \times t(P)$ where $[P] = \{l_1, \ldots, l_n\}$ is the prefix class. Let X_i denote the itemset Pl_i, then each member of $[P]$ is an IT-Pair $X_i \times t(X_i)$.

Theorem 6.3.5. *Let $X_i \times t(X_i)$ and $X_j \times t(X_j)$ be any two members of a class $[P]$, with $|t(X_i)| \leq |t(X_j)|$. The following four properties hold:*

1. *If $t(X_i) = t(X_j)$, then $c(X_i) = c(X_j) = c(X_i \cup X_j)$*
2. *If $t(X_i) \subset t(X_j)$, then $c(X_i) \neq c(X_j)$, but $c(X_i) = c(X_i \cup X_j)$*
3. *If $t(X_i) \supset t(X_j)$, then $c(X_i) \neq c(X_j)$, but $c(X_j) = c(X_i \cup X_j)$*
4. *If $t(X_i) \neq t(X_j)$, then $c(X_i) \neq c(X_j)$ and $c(X_i), c(X_j) \neq c(X_i \cup X_j)$*

Proof. We prove each of the four properties:

1. If $t(X_i) = t(X_j)$ then obviously $c(X_i) = c(X_j)$. Furthermore, $t(X_i) = t(X_j)$ implies that $t(X_i \cup X_j) = t(X_i) \cap t(X_j) = t(X_i) = t(X_j)$, therefore $c(X_i \cup X_j) = c(X_i) = c(X_j)$. This property implies that neither X_i or X_j are closed, and that every occurence of X_i can be replaced with $X_i \cup X_j$, and we can prune itemset X_j since its closure is identical to the closure of $X_i \cup X_j$.

2. If $t(X_i) \subset t(X_j)$ then $t(X_i \cup X_j) = t(X_i) \cap t(X_j) = t(X_i)$, giving us $c(X_i \cup X_j) = c(X_i)$. Therefore, X_i is not closed, and instead of processing it, we can directly process $X_i \cup X_j$. But since $t(X_i) \subset t(X_j) \Longrightarrow c(X_i) \neq c(X_j)$, X_j still needs to be processed, and cannot be pruned.

3. Case 3 is symmetrical to the case above.

4. If $t(X_i) \neq t(X_j)$, then clearly $t(X_i \cup X_j) = t(X_i) \cap t(X_j) \neq t(X_i)$ and also $t(X_i \cup X_j) = t(X_i) \cap t(X_j) \neq t(X_j)$, giving us $c(X_i \cup X_j) \neq c(X_i)$ and also $c(X_i \cup X_j) \neq c(X_j)$, therefore no element in the class can be pruned, as both X_i and X_j lead to different closures.

□

The pseudo-code for CHARM appears in Algorithm 12.

The algorithm starts by initializing the prefix class $[P]$, of nodes to be examined, to the frequent items and their tidsets in Line 1. The items are sorted in increasing order of their support sizes in Line 2, this increases the opportunity for pruning elements from a class $[P]$ according to properties 1 and 2 in Theorem 6.3.5.

The main computation is performed in CHARM-Extend, Algorithm 13, which returns the set of closed frequent itemsets. CHARM-Extend is a recursive procedure that is responsible for considering each combination of IT-pairs appearing in the prefix class of $[P]$. For each IT-pair $X_i \times t(X_i)$, it combines it with other IT-pairs $X_j \times t(X_j)$ that come after it (i.e., $|t(X_j)| \geq |t(X_i|)$.

Each X_i generates a new prefix class $[P_i]$ that is initially empty (Line 2). Line 4 tests which of the four IT-pair properties, discussed in theorem 6.3.5, can be applied by calling CHARM-Property (Algorithm 14). This routine will modify the current class $[P]$ by deleting IT-Pairs that have no chance of representing CFIs. It also inserts the newly generated IT-Pairs in the new class $[P_i]$. Once all X_j have been processed, we recursively explore the new class $[P_i]$ in a depth-first manner (Line 8).

The itemset X_i is inserted into the set of closed itemsets (Line 11), provided that X_i is not subsumed by a previously found closed itemset. At this stage any CFIs containing X_i have already been generated.

In order to check whether an itemset is subsumed by an itemset with the same support size, CHARM uses a hash table. It computes a hash function on the itemset's tidset and stores in the hash table the itemset along with its support size (the actual tidset cannot be stored in the hashtable as space requirements would be prohibitive). Before adding an itemset X to the set of closed itemsets C, CHARM retrieves from the hash table all entries with the same hash key. For each matching closed itemset c, it then checks whether $|t(c)| = |t(X)|$. If yes, then it checks whether $c \supset X$. If yes, then X cannot be a CFI, otherwise it is inserted into the hash table.

Algorithm 12: The CHARM Algorithm

```
Input:
    1. Transaction table $D$.
    2. Minimum support threshold $min\_sup$.
Output: The set of CFIs, $C$
Procedure CHARM($D, min\_sup$):
1: $[P] = \{X_i \times t(X_i) : X_i \in \mathcal{I}\ and\ |t(X_i)| \geq min\_sup\}$
2: Sort the items in $P$ in increasing order of their support size.
3: Call CHARM-Extend($[P]$,$C = \emptyset$).
4: return $C$ {the set of CFIs}
```

Algorithm 13: CHARM-Extend

Input:

1. Equivalence class $[P] = \{X_1 \times t(X_1), \ldots, X_n \times t(X_n)\}$.
2. CFI set C to be updated.

Output: Updated set C

```
1: for all $X_i \times t(X_i)$ in $[P]$ do
2:    $[P_i] \leftarrow \emptyset$
3:    for all $X_j \times t(X_j)$ in $[P]$, such that $j > i$ do
4:       Call CHARM-Property($[P], [P_i], X_i \times t(X_i), X_j \times t(X_j)$)
5:    end for
6:    if $[P_i] \neq \emptyset$ then
7:       Sort the nodes in $[P_i]$ in increasing order of support size.
8:       CHARM-Extend($[P_i], C$)
9:    end if
10:   delete $[P_i]$
11:   $C \leftarrow C \cup X_i$ {if $X_i$ is not subsumed by another itemset with the
      same support size}
12: end for
```

Algorithm 14: CHARM-Property

Input:

1. Equivalence class $[P] = \{X_1 \times t(X_1), \ldots, X_n \times t(X_n)\}$.
2. Equivalence class $[P_i]$.
3. TID-pairs to examine $X \times t(X)$, $Y \times t(Y)$ s.t $|t(X)| \leq |t(Y)|$.

Output:

1. Updated class $[P]$.
2. Updated class $[P_i]$.

```
1: if $|t(X)| \geq min\_sup$ then
2:   if $t(X) = t(Y)$ then
3:     Remove $Y$ from $[P]$
4:     Replace $X$ with $X \cup Y$ in $[P]$ while retaining the increasing
       support_size sort order
5:   else if $t(X) \subset t(Y)$ then
6:     Replace $X$ with $X \cup Y$ in $[P]$ while retaining the increasing
       support_size sort order
7:   else if $t(X) \neq t(Y)$ then
8:     if $|t(X) \cap t(Y)| \geq min\_sup$ then
9:       Add $X \cup Y \times t(X) \cap t(Y)$ to $[P_i]$
10:    end if
11:  end if
12: end if
```

Chapter 7

Generalized Frequent Itemset Mining Algorithms

For many applications, it is difficult to find strong associations among data items at low levels of abstraction due to the sparsity of data at those levels. Furthermore, associations discovered at higher levels of abstraction may reveal important trends, which may not be discovered by rules containing only items from the lower levels of abstractions. Therefore, data mining systems should provide capabilities for mining association rules at multiple levels of abstraction, with sufficient flexibility for easy traversal among different abstraction levels. This chapter begins by describing the problem of mining frequent itemsets from various levels of abstraction (Section 7.1). In Sections 7.2 and 7.3 we describe algorithms for mining frequent itemsets which contain items from different levels of abstractions. Then, in Section 7.4, we introduce a novel algorithm for mining generalized frequent itemsets for the setting of structured tables (see Section 2.1), as is the case in the k-anonymity problem. That algorithm will be used later in our proposed approximation algorithm to the k-anonymization problem.

7.1 Problem definition

Let $M = \{a_1, a_2, \cdots a_m\}$ be a set of items, and let $D = \left\langle T_1, T_2 \cdots T_n \right\rangle$ be a transaction table where T_i is a transaction that contains a subset of items in M. Let Tax be a directed acyclic graph on the items (Figure 7.1). An edge in Tax represents an *is-a* relationship, and Tax represents a set of taxonomies. We will call $\hat{x}$ an ancestor of x (and x a descendant of $\hat{x}$) if there is an edge from $\hat{x}$ to x in the transitive closure of Tax. We say that

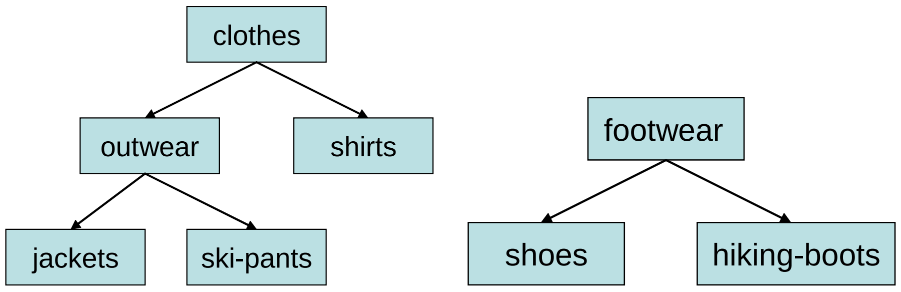


Figure 7.1: Taxonomy

transaction T supports an item $x \in M$ if x is in T or x is an ancestor of some item in T. Given a transaction table D and a minimum support threshold σ, we would like to be able to mine frequent itemsets whose items range from all parts of the taxonomy Tax. An important note is that the size of the support for an item in the taxonomy does not equal the sum of the support sizes of its descendants, since several of the descendants could be present in a single transaction.

7.2 The basic algorithm

This algorithm [19] is a variation on the Apriori algorithm. The change is that for each transaction in the table, all the ancestors of each item in the transaction are added to the transaction (after removing duplicates). Namely, for each transaction $T \in D$, $T' \leftarrow T \cup ancestors(T)$. Thus, an itemset X which may contain items from any place in the taxonomy hierarchy, is supported by the transaction T if X is included in T'.

7.3 The Cumulate algorithm

This algorithm [19] adds several optimizations to the basic algorithm:

1. Instead of traversing the taxonomy graph for each item, we can precompute the ancestors for each item. The list of all ancestors of all items is stored in a table, to which we refer as Tax^*.

2. We do not need to add all ancestors of the items in a transaction to it. Instead, we just add the ancestors that appear in at least one of the candidate itemsets that are being counted in the current pass.

As an example, assume the taxonomy in Figure 7.1 and consider the following table with four transactions:

$$\{\text{shirts, boots}\}, \{\text{jacket, boots}\}, \{\text{shirts, shoes}\}, \{\text{jacket, shoes}\} \ .$$

If the minimum support size is 3 then the following are the frequent itemsets:

$$\{\text{footwear}\}, \{\text{clothes}\}, \{\text{clothes, footwear}\} \ .$$

If the above items are the only ones being currently counted then in any transaction containing "jacket" we can safely add only the "clothes" ancestor, since there is no need to add the "outwear" ancestor which at this point is already known to be not frequent. That is, any ancestor in Tax^* that does not appear in any of the current candidates C_k, is pruned.

3. We can prune itemsets that contain an item and one of its ancestors. The justification for this is that the support of an itemset X that contains both an item x and its ancestor $\hat{x}$ will be the same as the support for the itemset $X \setminus \{\hat{x}\}$ (and that candidate was considered in the former pass over the table). The pruning should be performed for each C_k (and not only for C_2 as originally claimed in [19]).

4. An item in a transaction t can be pruned if it is not present in any candidate in C_k. It is safe to prune such items because if an item is not contained in any candidate in C_k then there is no chance that it will belong to an itemset in F_k and thus to larger candidate sets, C_{k+1}, C_{k+2}, etc. It is important to note that this improvement should be performed only *after* improvement step (2) that was discussed above. If not, then we might prune items which do not belong to C_k because they are not frequent enough, but their ancestors are frequent. So we should add the ancestors before pruning the items. For example, assume that {clothes, shoes} is the only candidate being currently inspected. If a transaction contains "jacket", we need to first add the "clothes" ancestor to the transaction and only then prune the "jacket" item. Otherwise, if we first remove "jacket" for not being frequent, we would miss its ancestor "clothes" that *is* frequent.

The Algorithm is illustrated in Algorithm 15.

Algorithm 15: Algorithm Cumulate

```
Input: Table $D$ of transactions, minimum support threshold $\sigma$.
Output: $F(D,\sigma)$ // frequent generalized itemsets in table $D$ with
support threshold at least $\sigma$
 1: Compute $Tax^*$ the set of ancestors of each item from $Tax$
    {improvement 1}
 2: $F_1$ = {frequent 1-itemsets}
 3: $k = 2$
 4: while ($F_{k-1} \neq \emptyset$) do
 5:    $C_k$={New candidates of size k generated from $F_{k-1}$}
 6:    Delete any candidates in $C_k$ that contain an item and its ancestor
       {improvement 3}
 7:    Delete any ancestors in $Tax^*$ that are not present in any of the
       candidates in $C_k$ {improvement 2}
 8:    for all $T \in D$ do
 9:       for all $x \in T$ do
10:          Add all ancestors of $x$ in $Tax^*$ to the transaction $T$, omitting
             duplicates
11:       end for
12:       Delete any items in $T$ that are not present in any of the
          candidates in $C_k$ {improvement 4}
13:       Increment the count of all candidates in $C_k$ that are contained in $T$
14:    end for
15:    $F_k$= {All candidates in $C_k$ with support size at least $\sigma$}
16:    $k = k + 1$
17: end while
18: return $\bigcup_k F_k$
```

7.4 Proposed generalized itemset mining algorithm

7.4.1 The setting

When dealing with k-anonymity, the table consists of records that contain quasi-identifier information. Those records (as opposed to the more general case of transactions) have a very specific structure. Letting A_j, $1 \leq j \leq r$, denote the r attributes (or quasi-identifiers), each record belongs to $A_1 \times \cdots \times A_r$ (see Section 2.1). Each attribute has a corresponding collection of subsets, $\overline{A_j} \subseteq \mathcal{P}(A_j)$. We assume that this collection is proper (Definition

2.2.2). It is shown in [7, Lemma 3.3] that the class of proper generalization coincides with the class of generalizations by taxonomies. Therefore, the subsets in collection $\overline{A}_j$ form a taxonomy, $tax(j)$. The root of that taxonomy, which will also be denoted as $tax(j)$, corresponds to the entire set A_j; every node in the tree $tax(j)$ corresponds to a subset of values of A_j, and the leaves correspond to the singleton values in A_j.

Definition 7.4.1. *A generalized itemset is a record $\overline{R} \in \overline{A}_1 \times \cdots \times \overline{A}_r$. A generalized itemset is called frequent if its support is of size at least k. A generalized frequent itemset is called closed if none of its proper specifications has the same support.*

Comment. Our notion of a generalized itemset extends the notion of an itemset that was used in [15]; correspondingly, our notion of a closed frequent generalized itemset extends that of a closed frequent itemset. While the itemset notions were suitable for the restricted setting of generalization by suppression only, our generalized itemset notions offer the proper extension for the case when any generalization is allowed.

There is an apparent difference between the definition of closed frequent itemsets with regard to generalization, as defined above, and the standard definition of closed frequent itemsets, as defined in Definition 4.2.3. Definition 7.4.1 considers an itemset as closed if it has no specification that has the same support, while Definition 4.2.3 considers an itemset as closed if it has no superset that has the same support. Both definitions follow the same logic: Specifying a generalized itemset, as well as extending a standard itemset – both are operations that may shrink the support of the itemset. Hence, the two definitions are consistent in the sense that they consider an itemset as closed if all operations that may potentially shrink its support indeed do.

There are several differences between the setting of structured tables and transactional tables:

- All of the frequent itemsets are of fixed size, r, as opposed to the general setting in which the frequent itemsets may be of any length.
- In the general setting, two or more items in a transaction may belong to the same taxonomy hierarchy; for example "jackets" and "ski pants" may appear in the same transaction. Therefore, in such a case the size of the support set of an item in the taxonomy does not equal the sum of the sizes of the supports of its children. But, in tables that are subjected to the k-anonymity transformation, no transaction, namely

- a record in such a table, contains two items which belong to the same quasi-identifier (e.g., more than one age, or more than one zipcode). Therefore, we can say that the distinct values of each quasi-identifier form a partition of the records in the table.

We assume hereinafter that the cost function is monotone, (Definition 3.3.1). Under that assumption, we would be interested only in generalized frequent itemsets that have minimal generalization costs. In other words, if $\overline{R}$ and $\overline{R}'$ are both generalized frequent itemsets, $\overline{R} \sqsubset \overline{R}'$, and they have the same support, we shall not be interested in $\overline{R}'$ since $cost(\overline{R}') \geq cost(\overline{R})$. Hence, we are interested only in generalized frequent itemsets that are closed. Our algorithm will recursively traverse the space of generalized itemsets while pruning branches which cannot lead to generalized closed frequent itemsets. As our solution will only require the closed frequent itemsets, we do not need to accumulate all of the frequent itemsets that we encounter during the search.

The main underlying property exploited by this algorithm is that the support size is monotonically decreasing with respect to specification of an itemset. Namely, if $\overline{R}, \overline{R}'$ are two itemsets and $\overline{R} \sqsubseteq \overline{R}'$, then $|support(\overline{R})| \leq |support(\overline{R}')|$. Therefore, if an itemset is infrequent, then all of its specifications must be infrequent as well.

7.4.2 Overview of the algorithm

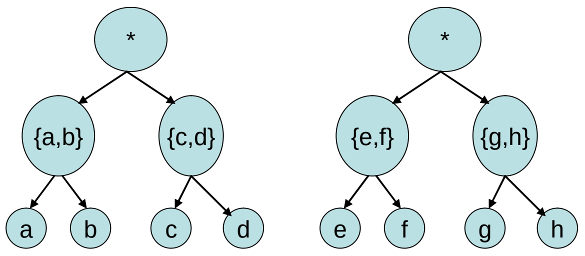
Let $D = \{R_1, R_2 \cdots R_n\}$ be a table with r quasi-identifiers. The values $R_i(j)$, $1 \leq i \leq n$, $1 \leq j \leq r$, belong to the first level of the hierarchy of quasi-identifier j (i.e no generalization). An itemset, $(a_1, \ldots, a_r)$, is an r-length record, in which a_j, $1 \leq j \leq r$, is a node from quasi-identifier j's taxonomy tree, $tax(j)$. (Alternatively, we can say that $(a_1, \ldots, a_r) \in \overline{A}_1 \times \cdots \times \overline{A}_r$.) The support of such an itemset is the set of records ids in D which support it. The itemset is frequent if its support size is of size at least k.

Our algorithm will traverse the search space by beginning with the most generalized itemset, $(A_1, \ldots, A_r)$. That itemset is supported by all records of D since every record in D belongs to $A_1 \times \cdots \times A_r$. Hence, it is frequent. The algorithm will explore the search space in a depth-first manner, pruning infrequent branches according to the downward-closure property (A specification of an infrequent itemset cannot be frequent). Branches which have no chance of leading to closed itemsets will be pruned as well (see Lemma 7.4.1 below).

Figure 7.2 illustrates an example of a table with ten records and two attributes ($n = 10$, $r = 2$) and the accompanying taxonomies for each of

A	B
a	e
d	h
b	h
c	g
d	e
b	g
b	e
c	f
a	e
b	g

(a)



(b)

Figure 7.2: (a) Table (b) Taxonomies

the two attributes. That toy example will be used in order to exemplify the action of the algorithm.

Since we are interested in mining only the closed frequent itemsets, it would be beneficial to prune the search space by removing those frequent itemsets which would not lead to undiscovered closed frequent itemsets. To that end, we will apply Lemma 7.4.1.

Lemma 7.4.1. *Let X be a frequent itemset, and let X' be an itemset such that $X' \sqsubset X$. If support$(X) =$ support(X'), then X is not closed. Moreover, assume that Y is an itemset such that $Y \sqsubseteq X$ and there exists an index $1 \leq j \leq r$ for which $X'(j) \subsetneqq Y(j)$. Then Y is not closed either.*

Proof. Since $X' \sqsubset X$, but the two itemsets have the same support, X is not closed.

Now, consider an itemset $Y = (Y(1), \ldots, (Y(r))$ as described in the second part of the lemma. Since $Y \sqsubseteq X$ then

$$\mathit{support}(Y) \subseteq \mathit{support}(X) = \mathit{support}(X')\,. \tag{7.1}$$

Since the taxonomy structures, $\mathit{tax}(j)$, $1 \leq j \leq r$, are proper (Definition 2.2.2), one of the following holds for each $1 \leq j \leq r$:

$$X'(j) \subseteq Y(j)\ , \quad \text{or} \quad X'(j) \supseteq Y(j)\ , \quad \text{or} \quad X'(j) \cap Y(j) = \emptyset\,.$$

We separate our discussion into two cases.

Case 1. Assume that there exists an index $1 \leq j \leq r$ for which $X'(j) \cap Y(j) = \emptyset$. In this case, $\mathit{support}(X'(j)) \cap \mathit{support}(Y(j)) = \emptyset$, since no record can contain in its jth entry items from both $X'(j)$ and $Y(j)$, (otherwise, there would exist a record whose jth attribute contains more than one value, in contradiction to the assumed table structure). Consequently, $support(X') \cap support(Y) = \emptyset$, because for any itemset Z, $support(Z(j)) \supseteq support(Z)$ for every $1 \le j \le r$. As $support(X') = support(X)$, we infer that $support(X) \cap support(Y) = \emptyset$. Finally, in view of (7.1) we arrive at the conclusion that $support(Y) = \emptyset$.

Consider now the itemset

$$Y' = (Y(1), \dots, Y(j-1), X'(j), Y(j+1), \dots, Y(r))$$

where j is the index for which $X'(j) \subsetneqq Y(j)$. As $Y' \sqsubset Y$ and $support(Y) = \emptyset$, we conclude that $support(Y') = \emptyset$. Hence, Y has a proper specification Y' with the same support. Therefore, Y is not closed. This settles the proof in Case 1.

Case 2. Assume that for all $1 \le j \le r$, $X'(j) \subseteq Y(j)$ or $X'(j) \supseteq Y(j)$. Consider the itemset $Y' = (Y'(1), \dots, Y'(r))$, where:

$$Y'(j) = \begin{cases} X'(j) & X'(j) \subsetneqq Y(j) \\ Y(j) & X'(j) \supseteq Y(j) \end{cases} , \quad 1 \le j \le r .$$

In other words, $Y'(j) = Y(j) \cap X'(j)$ for all $1 \le j \le r$. Therefore,

$$support(Y') = support(Y) \cap support(X') = support(Y) .$$

Hence, we found an itemset $Y' \sqsubset Y$ for which $support(Y') = support(Y)$. Consequently, Y is not closed. This settles the proof in the second case as well. □

Example 7.4.1. *Consider the table in Figure 7.2 and take*

$$X = (\{a, b\}, \{g, h\}) \ , \quad X' = (\{b\}, \{g, h\}) .$$

Clearly, $X' \sqsubset X$, but they both have the same support, namely $\{3, 6, 10\}$. As implied by Lemma 7.4.1, there is no need to process itemsets such as $Y = (\{a, b\}, \{g\})$ (indeed, $Y \sqsubset X$ and $X'(1) \subsetneqq Y(1)$). The reason is that the support of Y equals that of $Y' = (\{b\}, \{g\})$ (the set of records $\{6, 10\}$), whence it is not closed.

7.4.3 Data structures used for counting itemset support

The taxonomy structures illustrated below may be exploited in order to design an efficient algorithm for finding the support of a given generalized itemset. To that end, we begin by passing over the table and constructing augmented taxonomies in which each leaf is augmented by the list of all records which contain the value in that leaf. This way, the union of the lists of all leaves under a given node in the taxonomy will be the support for that node. Figure 7.3 illustrates the augmented taxonomies for the example in Figure 7.2.

Assume next that we wish to compute the support of itemset $(a_1, \ldots, a_r)$, where a_j is a node in taxonomy $tax(j)$. Then all we need to do is to intersect the supports of each of the sets of records that are associated with the r nodes a_j. For example, if we wish to compute the support of the itemset $(\{a, b\}, \{g\})$, we first compute the list of records in which the first entry supports the set $\{a, b\}$, then compute the list of records in which the second entry supports the set $\{g\}$, and then intersect the two lists. The first list is $\{1, 3, 6, 7, 9, 10\}$ (it is the union of the two lists under the node $\{a, b\}$ in the first taxonomy in Figure 7.3), while the second list is $\{4, 6, 10\}$. The support of $(\{a, b\}, \{g\})$ is the corresponding intersection, $\{6, 10\}$.

Computing the itemset support in such a manner is much more efficient than traversing the complete table and inspecting each record. After making one pass over the entire table, the algorithm needs only to mine these compact taxonomy structures, no other table traversals are required.

7.4.4 The algorithm

Algorithm 16 starts by augmenting the taxonomies with the lists of supports for each leaf in each taxonomy (Step 1). It then prunes from $tax(j)$ nodes whose support is of size less than k since no itemset that includes the corresponding generalized value in its jth entry can be frequent (Steps 2-4). Then it starts with the most generalized itemset, $(A_1, \ldots, A_r) = (*, \ldots, *)$ (which is obviously frequent, but not necessarily closed), and starts traversing the search space, which is $tax(1) \times \cdots \times tax(r)$, in a depth-first manner. An itemset will be added to the output list, $F_{CF}(D, k)$, if it is frequent and closed, i.e., none of its direct specifications has the same support as it does. Both checks can be made quickly and easily using the special data structures that were constructed in Step 1. Since this depth-first search may attempt to examine the same itemset more than once, we keep a hashtable (*processed*) of all itemsets that were already examined in order not to check them again. (See Section 7.4.5 for more details regarding the actual implementation.)

Algorithm 16: Proposed algorithm

```
Input:
  • Table $D$ of $n$ records and $r$ quasi-identifiers;
  • Minimum support threshold $k$;
  • Taxonomy trees for quasi-identifiers $1 \leq j \leq r$, $tax(1), \cdots, tax(r)$.
Output: $F_{CF}(D,k)$ – the list of all closed frequent generalized
itemsets of support size at least $k$.
1: Make a single pass over $D$ and augment the trees $tax(1), \cdots, tax(r)$
   with the supporting record ids.
2: for all $1 \leq j \leq r$ do
3:     Remove from $tax(j)$ all nodes that are supported by less than $k$
       records.
4: end for
5: $root \leftarrow (A_1, \ldots, A_r)$ {The most generalized itemset, which is supported
   by all of $D$, consists of the roots of all taxonomy trees}
6: $processed \leftarrow \emptyset$
7: $F_{CF}(D,k) \leftarrow \emptyset$
8: call $GetClosedFrequentItemsets(root, k, processed, F_{CF}(D,k))$
9: return $F_{CF}(D,k)$
```

The main function in this algorithm is called in Step 8: That function, which is implemented in Algorithm 17, receives as an input a root itemset and adds to the list of closed frequent itemsets all the closed frequent itemsets that are specifications of the given root. That function implements the depth-first search, starting from the given root downward.

It starts (Steps 1-3) by checking whether the given root was already processed, or if its support is of size less than k; in either case it returns without going on further. It then continues and adds the root to the list of processed itemsets (Step 4) and marks it as closed (Step 5). The mark of the root as being closed will remain so until we are convinced otherwise.

Next, we start the loop over all quasi-identifiers ($j = 1, \ldots, r$) (Step 6) and for each identifier over all the children of a_j (Step 7). This double loop scans all itemsets that are immediate specifications of the input itemset. (To this end, an itemset X' is an immediate specification of itemset X if X and X' coincide in all entries except for one entry, say the jth entry, in which $X'(j)$ is an immediate child of $X(j)$.) We first compute the support

Algorithm 17: Generalized Closed Itemset Mining Algorithm

Input:

- An itemset $(a_1, \ldots, a_r)$, referred to as *root*, where each a_j is a node in the taxonomy hierarchy of quasi-identifier j, $tax(j)$;
- k, the minimum support for itemsets;
- *processed*, the set of the processed itemsets;
- *closed*, a set of closed frequent itemsets;

Output: Adding to the given list *closed* all the closed frequent itemsets which are specifications of *root*.

```
1:  if (($root \in processed$) $OR$ ($|support(root)| < k$)) then
2:    return
3:  end if
4:  $processed \leftarrow processed \cup \{root\}$
5:  $isRootClosed \leftarrow true$
6:  for $j = 1$ to $r$ do
7:    for all children $taxChild(j)$ of $a_j$ in $tax(j)$ do
8:      $new\_itemset\_support \leftarrow support(taxChild(j)) \cap support(root)$
9:      if ($|new\_itemset\_support| \geq k$) then
10:       if ($|new\_itemset\_support| = |support(root)|$) then
11:         $isRootClosed \leftarrow false$
12:       end if
13:       $new\_itemset \leftarrow (a_1, \ldots, a_{j-1}, taxChild(j), a_{j+1}, \cdots, a_r)$
14:       $GetClosedFrequentItemsets(new\_itemset, k, processed, closed)$;
15:       if ($isRootClosed = false$) then
16:         break loop;
17:       end if
18:     end if
19:   end for
20: end for
21: if ($isRootClosed = true$) then
22:   $closed \leftarrow closed \cup \{root\}$
23: end if
```

of that specification (Step 8). If that support is of size less than k (Step 9) there is no need to further process it, and it does not contradict our current assumption that the input itemset is closed. If the support of that specification equals that of the input itemset then we mark the input itemset as not closed (Steps 10-12). Then we activate the depth-first strategy by calling the function, recursively, with that specification as an input. Finally, if the input itemset has been already established as not closed, we can break the double-loop and not proceed to check the rest of its specifications. To see why, let us assume that the root is $X = (a_1, \ldots, a_r)$ and assume that a_j has in $tax(j)$ the children $b_1, \ldots, b_t$. Assume that while examining the specification $X' = (a_1, \ldots, a_{j-1}, b_1, a_{j+1}, \ldots, a_r)$, we discovered that X and X' have the same support (whence, we set *isRootClosed* to be false). Then that means that the support of $(a_1, \ldots, a_{j-1}, b_i, a_{j+1}, \ldots, a_r)$ for all $i > 1$ is empty and so we can break out of the inner loop on the children. Moreover, there is no need to continue the loop over j since the resulting specifications there cannot be closed and cannot lead to other closed itemsets as implied by Lemma 7.4.1.

Finally, at the end of the loop over all immediate specifications, if the root survived all checks and proved to be closed, we add it to the list of closed frequent itemsets (Steps 21-23).

Comment. We attempted to apply the optimizations that are provided by the Charm Algorithm and specified in Theorem 6.3.5. Although this caused the algorithm to process less frequent itemsets, we observed that it did not reduce the overall running time in our experiments. This is due to the fact that computing these optimizations took more time than the time that was saved by processing less frequent itemsets. It should be noted that most of the frequent itemsets which could not lead to undiscovered CFIs were pruned as a result of the optimization provided by Lemma 7.4.1.

7.4.5 Implementation notes

Algorithms 16 and 17 were implemented in Java. The main issue to overcome was the program's memory consumption. This led to the decision to implement the algorithm in a depth-first manner. Other optimizations that we implemented were as follows:

- We implemented an object pool of frequent itemset objects. This greatly helped in reducing the memory consumption by reusing formerly processed frequent itemset objects.

- The frequent itemsets form a lattice (not a tree), whence it was important not to process the same frequent itemset more than once, in order to avoid much larger running times and memory consumption. For this purpose, we used an integer hash table that contained the hash codes of the already processed itemsets.

- The supports of the nodes were represented as bit vectors. Apart from reducing the memory that was used for that purpose, it enabled efficient calculation of the support sizes of the frequent itemsets due to the fact that the intersection process could be performed quickly.

- The closed frequent itemsets that were found during the execution of the algorithm, were serialized and saved in a database table due to memory limits. The writing to the database table was performed in a separate thread in order to make the processing thread as efficient as possible.

- The immediate children of a certain frequent itemset X in Algorithm 17 (lines 6-7), and their supports, are calculated before anyone of them is recursively processed. The incentive for this is that if X is not closed, then one of its immediate children has the same support as X. By first creating all of X's immediate children (along with their support cardinalities), we ensure that if X has such a child, it would be the only one to be processed (Lemma 7.4.1). In such a case, we skip the processing of the remaining children of X and accelerate the mining process by pruning the maximum number of itemsets.

It is useful to note that the set of closed frequent itemsets that was computed for a certain value of k can be utilized in order to run the anonymization algorithms for all other larger values of k, because the size of the set of closed frequent itemsets decreases monotonically (with respect to inclusion) when the minimum support parameter k increases.

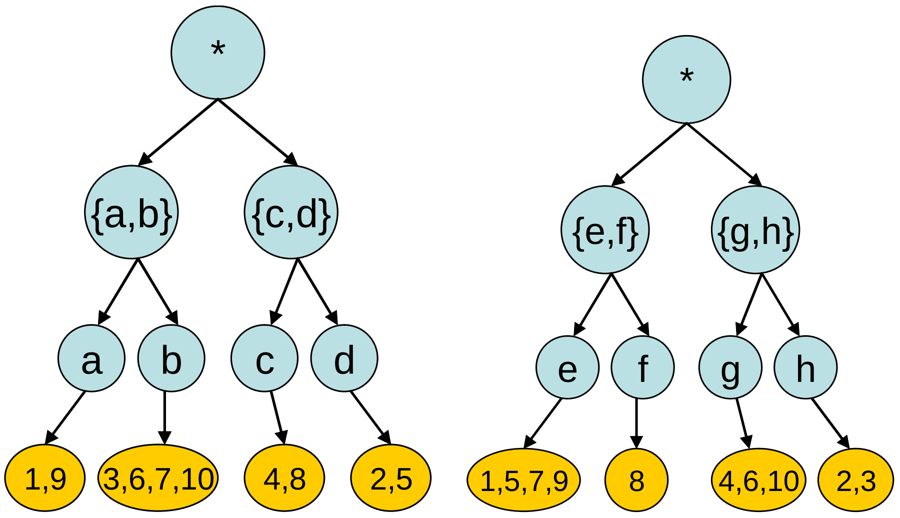


Figure 7.3: Taxonomy structures

Chapter 8

An Improved Approximation Algorithm for k-Anonymity

In this chapter we describe and discuss our improved approximation algorithm for k-anonymity. The algorithm and the accompanying proofs follow the lines of those that were provided in [15] for the case of k-anonymization by suppression. We will adjust these algorithms and proofs to the general case of k-anonymity.

The idea is to use Algorithm 3. In that algorithm there were two phases. In the first phase we invoke Algorithm 1 (Gen-Cover) that outputs a cover of D that approximates the optimal $[k, 2k-1]$-cover of D to within an approximation ratio of $O(\ln k)$. In the second phase we convert that cover into a $[k, 2k-1]$-clustering. As shown in Theorem 4.2.3, that clustering induces a k-anonymization that approximates the optimal k-anonymization within a factor of $O(\ln k)$.

The main disadvantage of Algorithm 3 is the runtime of its first phase (Gen-Cover), which is $O(n^{2k})$. Here, we will present a modification of that algorithm, to which we refer as Gen-Cover-CF (Algorithm 18). That algorithm will also produce a cover of D that approximates the optimal $[k, 2k-1]$-cover of D to within an approximation ratio of $O(\ln k)$. However, its runtime will be a practical one.

Both algorithms – Gen-Cover and Gen-Cover-CF – receive as an input a collection of subsets of $C \subseteq \mathcal{P}(D)$ from which they select the subsets for the cover. The runtime of both algorithms is bounded by $O(|D| \cdot |C|)$. Hence, the key idea is to reduce dramatically the size of the input collection C. In the original algorithm Gen-Cover, the input collection is $C = F := \{S \subset D : k \leq |S| \leq 2k-1\}$. Its size is $O(n^{2k-1})$. In the modified algorithm Gen-

Cover-CF, the input collection is $C = F_{CF}$, where F_{CF} contains only sets of records, S, whose closure, $\overline{S}$, corresponds to a closed frequent generalized itemset; here, a frequent itemset is one whose support is of size at least k. While the runtime of Gen-Cover is $O(|F| \cdot |D|) = O(n^{2k})$, the runtime of Gen-Cover-CF is $O(|F_{CF}| \cdot |D|)$. Generally, the size of F_{CF} is much smaller than that of F. Hence, the runtime of Gen-Cover-CF is much smaller than that of Gen-Cover.

This change in the input collection of subsets requires us also to modify the algorithm itself. In Gen-Cover, it was essential that all subsets in the cover are of size between k and $2k-1$. In Gen-Cover-CF, the input collection of subsets is F_{CF}, and it may include subsets of any size greater than or equal to k. Hence, whenever the greedily selected subset S has a size larger than $2k-1$, we create a subset by randomly selecting up to $2k-1$ records from S, such that the number of uncovered records is maximized, and then insert that subset into the cover γ.

The modified algorithm Gen-Cover-CF is given in Algorithm 18. That algorithm is then used as a procedure in Algorithm 19, which is the approximation algorithm for k-anonymity.

Algorithm 18: Gen-Cover-CF

Input: Table D; a collection of the supports of all closed frequent itemsets, F_{CF}

Output: A cover of D, where each set has size between k and $2k-1$, that has cost at most $O(\ln k)$ times the cost of the minimal cost cover

```
1: $\gamma = \emptyset$ {the current cover}
2: $E = \emptyset$ {currently covered records in $D$}
3: while ($E \neq D$) do
4:    for all $S \in F_{CF}$ do
5:       Compute the ratio $\rho(S) = \frac{d(S)}{min(|S \cap (D-E)|, 2k-1)}$
6:    end for
7:    Choose a set $S$ such that $\rho(S)$ is minimized
8:    if ($|S| \leq 2k-1$) then
9:       $S^R \leftarrow S$ {the set is in the right size}
10:   else if ($|S \cap (D-E)| \geq 2k-1$) then
11:      Choose $S^R \subseteq S \cap (D-E)$ s.t $|S^R| = 2k-1$ {select $2k-1$
         uncovered records}
12:   else {$|S| \geq 2k$ and $|S \cap (D-E)| < 2k-1$}
13:      Choose $S^R \subseteq S$ s.t $S^R \supseteq S \cap (D-E)$ and
         $|S^R| = max(k, |S \cap (D-E)|)$
14:   end if
15:   $E \leftarrow E \cup S^R$
16:   $\gamma \leftarrow \gamma \cup \{S^R\}$
17: end while
18: return $\gamma$
```

Algorithm 19: k-anonymization via set-cover using frequent itemsets

Input: Table D, integer k.

Output: Table $g(D)$ that satisfies $k-$anonymity

```
1: Find all closed generalized itemsets in $D$ whose support size is at least
   $k$ (Algorithm 16).
2: Set $F_{CF}$ to be the set of supports of all the found closed generalized
   frequent itemsets.
3: Produce a cover $\gamma$ of $D$, by using Algorithm 18, with $F_{CF}$ as an input.
4: Convert the resulting $[k, 2k-1]$-cover $\gamma$ into a $[k, 2k-1]$-clustering, $\gamma^0$,
   by invoking Algorithm 2.
5: Output the $k$-anonymization $g(D)$ of $D$ that corresponds to $\gamma^0$.
```

Comment. In Algorithm Gen-Cover-CF we use the same notation ρ as used in Algorithm Gen-Cover. In both algorithms this function has the same meaning, namely, the price paid per uncovered record.

We proceed to show that Gen-Cover-CF is an appropriate substitute to Gen-Cover, in the sense that it produces a cover of D that approximates the optimal $[k, 2k-1]$-cover to within $O(\ln k)$. Once we show that, we may conclude that Algorithm 19 produces a k-anonymization that approximates the optimal one to within $O(\ln k)$ by arguing along the same lines as in Theorem 4.2.3. First of all, it is clear that the cover γ produced by Gen-Cover-CF includes only subsets of size between k and $2k-1$. It remains to show that γ achieves the same approximation ratio as Gen-Cover (Algorithm 1).

Lemma 8.0.2. *If S and S^R are the subsets that are selected in each iteration of Gen-Cover-CF, then $\rho(S^R) \leq \rho(S)$.*

Proof. We will prove this claim for every set S and its corresponding subset S^R (namely, not only for sets S that minimize ρ). Since $S^R \subseteq S$ then, by monotonicity, $d(S^R) \leq d(S)$. Furthermore, it is easy to see that in each of the three cases in Gen-Cover-CF, we have

$$min\{|S \cap (D-E)|, 2k-1\} = min\{|S^R \cap (D-E)|, 2k-1\}\,.$$

This implies that $\rho(S^R) \leq \rho(S)$. □

Gen-Cover and Gen-Cover-CF may produce different solutions for a given input collection of subsets (F in the case of Gen-Cover and F_{CF} in the case of Gen-Cover-CF), since in each iteration there may exist more than one set that minimizes $\rho(\cdot)$, and the algorithms randomly select one of them. Let Sol and Sol_{CF} be the sets of all possible solutions that may be produced by Gen-Cover and Gen-Cover-CF, respectively. We proceed to show that $Sol_{CF} \subseteq Sol$. Namely, every solution that may be produced by Gen-Cover-CF is also a possible solution for Gen-Cover. Since all solutions of Gen-Cover achieve an approximation ratio of $O(\ln k)$ (since it is simply the greedy algorithm for the set-cover problem), we may infer a similar conclusion for all solutions of Gen-Cover-CF.

Lemma 8.0.3. *The possible solutions obtained by Gen-Cover-CF are always possible solutions for Gen-Cover.*

Proof. Let Sol_{CF} and Sol be the set of all possible solutions that may be produced by by Gen-Cover-CF with F_{CF} and Gen-Cover with F respectively.

Consider any solution $\Pi_{CF} \in Sol_{CF}$. We will show that there exists a solution $\Pi \in Sol$ such that $\Pi = \Pi_{CF}$. In other words, $Sol_{CF} \subseteq Sol$.

Let $\Pi_{CF} = \{S_1^{'R}, S_2^{'R}, \cdots, S_m^{'R}\}$ where the subscript denotes the selection order by Gen-Cover-CF. $S_i^{'R}$ is selected in lines 8-14 of Gen-Cover-CF as a subset of S_i' which was selected greedily among the subsets in F_{CF} in lines 4-7. We claim that there exists a solution $\Pi = \{S_1, S_2 \dots S_m\} \in Sol$ where the subscript denotes the selection order of the subsets and $S_i = S_i^{'R}$ for $1 \leq i \leq m$. That will prove that $\Pi_{CF} = \Pi \in Sol$. We will prove this claim by induction.

When $i = 0$, we have $\emptyset = \emptyset$ and thus the base case of our inductive proof holds. Assume that there exists a solution $\Pi \in Sol$ such that $S_i = S_i^{'R}$ for $1 \leq i \leq j-1$, where $1 \leq j \leq m$. We want to show that it holds for $i = j$ too. If $S_j \neq S_j^{'R}$, we will show that there exists a different solution $\Pi' \in Sol$ such that $S_i = S_i^{'R}$ for $1 \leq i \leq j$. Assume, towards contradiction, that there does not exist such a solution. We consider two cases:

When $\rho(S_j) < \rho(S_j^{'R})$: Let $S^{''}$ be the support of $\overline{S_j}$ in D. As $\overline{S''} = \overline{S_j}$, by the definition of $S^{''}$, it follows that $d(S^{''}) = d(S_j)$ (Definition 4.1.2). Since, $S^{''} \supseteq S_j$ we have that $\rho(S^{''}) \leq \rho(S_j)$, this is because we have accomplished that $d(S^{''}) = d(S_j)$ and $S^{''} \supseteq S_j$. So we have that $\rho(S^{''}) \leq \rho(S_j) < \rho(S_j^{'R})$ at the j-th iteration of Gen-Cover-CF. Since S_j' is the greedily selected subset at the j-th iteration of Gen-Cover-CF, we have $\rho(S_j^{'R}) \leq \rho(S_j')$ by Lemma 8.0.2. Therefore, $\rho(S^{''}) < \rho(S_j')$. By the definition of $S^{''}$, there is no other set, X, such that $X \supset S^{''}$ which has the same closure, namely $\overline{S_j}$. This, combined with the fact that $|S^{''}| \geq k$, gives us that $S^{''} \in F_{CF}$. But then, we should have selected $S^{''}$ instead of S_j' at the j-th iteration of Gen-Cover-CF, and this contradicts that $\rho(S_j')$ is minimum at the j-th iteration of Gen-Cover-CF.

When $\rho(S_j) > \rho(S_j^{'R})$: Since $k \leq |S_j^{'R}| \leq 2k-1$ and F is the collection of all subsets of D with cardinalities between k and $2k-1$, we have that $S_j^{'R} \in F$. Then we should have selected $S_j^{'R}$ instead of S_j at the j-th iteration of Gen-Cover. This contradicts that $\rho(S_j)$ is the minimum at the j-th iteration of Gen-Cover.

Thus, we have shown that $\rho(S_j) = \rho(S_j^{'R})$. Since we have that $k \leq |S_j^{'R}| \leq 2k-1$, we must have that $S_j^{'R} \in F$. Then, if Gen-Cover selected $\{S_1, \dots, S_{j-1}\}$ until the $(j-1)$-th iteration, it can select $S_j^{'R}$ instead of S_j at the j-th iteration. It results in another solution, that is, there exists a solution $\Pi' \in Sol$ such that it contains $S_1, \dots, S_{j-1}$ and $S_j^{'R}$. $\square$

Chapter 9

Experiments

We tested our algorithm versus the Forest (Section 4.3) and the modified Agglomerative algorithm (Chapter 5). The tests were conducted on three datasets from the UCI Machine Learning Repository [4]. These include Adult, Nursery and Coil2000.

- Adult: The dataset was extracted from the US census Bureau Data Extraction System. It contains demographic information of a small sample of US population with 14 public attributes such as age, education-level, marital-status, occupation, and native-country. The private information is an indication whether that individual earns more or less that 50 thousand dollars annually. The Adult data contains 30162 records after the records with missing values are removed.

- Nursery: The Nursery Database was derived from a hierarchical decision model that was originally developed to rank applications for nursery schools. The Nursery dataset contains 12960 records after deleting those with missing values. It has 8 quasi-identifier attributes.

- Coil2000: This data set used in the CoIL 2000 Challenge contains information on customers of an insurance company. The data consists of 86 variables and includes product usage data and socio-demographic data derived from zip area codes. The Coil2000 database contains 5822 records after deleting those with missing values. We used a set of 9 quasi-identifiers out of the 86 available.

We ran each of the four algorithms with five values of the anonymity parameter, $k = 50, 75, 100, 150, 200$. The information loss measure that we used in order to compare the algorithms is the LM measure, (3.1). We also verified the results on the entropy measure (Definition 3.2.2).

All of the algorithms that we tested were implemented in Java and run on a core 2 (R) quad (Q6600) CPU 2.4 GHz, 8GB of RAM.

9.1 Adult dataset experiments

The comparison between the various algorithms can be seen in the following figures. Figure 9.1 compares between the algorithms when run using the LM measure. Figure 9.2 compares between the algorithms when run using the entropy measure (EM). Figure 9.3 displays the runtime differences between the algorithms.

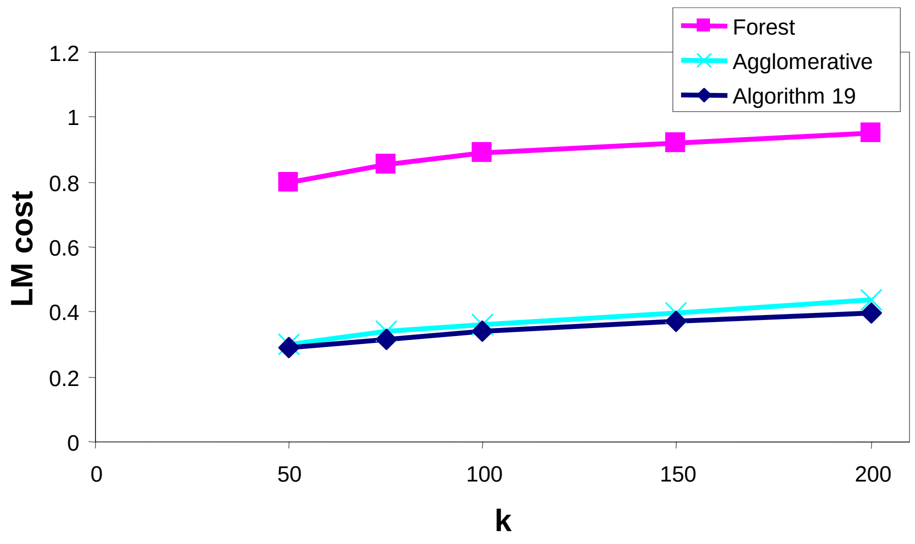


Figure 9.1: Algorithm score comparison (LM) - Adult dataset

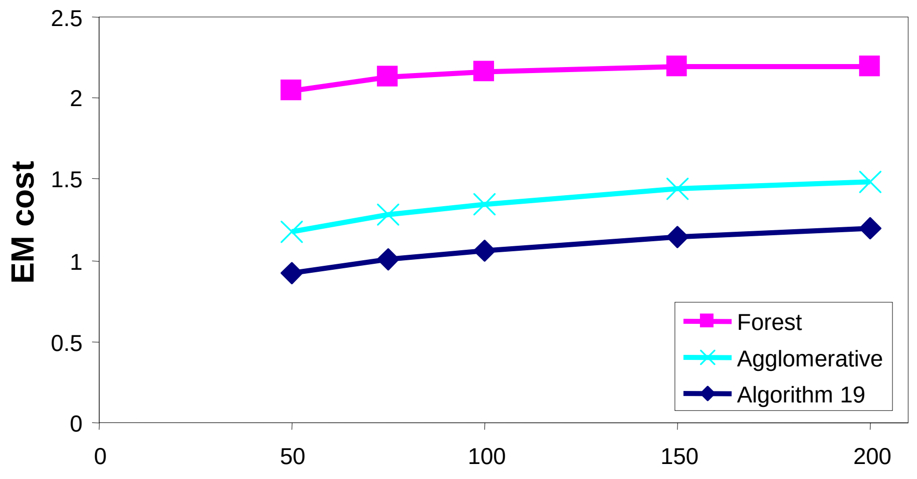


Figure 9.2: Algorithm score comparison (Entropy) - Adult dataset

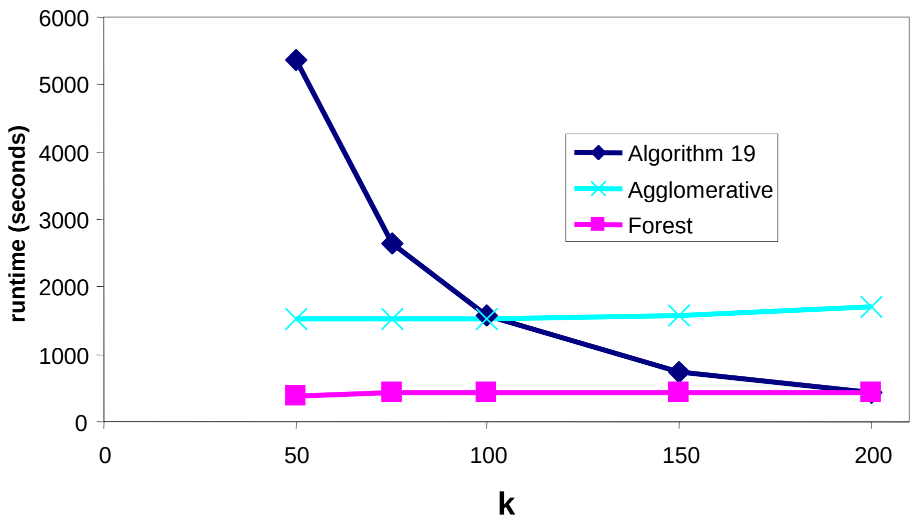


Figure 9.3: Algorithm runtime comparison - Adult dataset

As *k* decreases, the number of closed frequent itemsets increases, as can be seen in Table 9.1. This directly affects the runtime of both the closed frequent itemset mining phase of the algorithm as well as the Gen-Cover phase (see table 9.2).

k	Number of closed frequent itemsets
50	1,199,494
75	819,991
100	612,617
150	390,126
200	278,672

Table 9.1: *k* vs. the number of closed frequent itemsets – Adult dataset

k	Mining Algorithm runtime	Gen-Cover-CF runtime	total runtime
50	971	4389	5360
75	591	2050	2641
100	436	1131	1567
150	261	478	739
200	188	247	435

Table 9.2: *k* vs. mining and Gen-Cover-CF runtimes (seconds) – Adult dataset

We also investigated the relation between the number of public attributes in the table, and the attributes' hierarchy structure, to the runtime of Algorithm 19. To that end, we ran Algorithm 19 on four versions of the Adult dataset. The first was the original dataset, which has 14 public attributes. The second was a reduced version that had 11 public attributes; we removed attributes fnlwgt, capital-gain and capital-loss. The third dataset had 8 public attributes; we further removed attributes education-num, hours-per-week and native-country. The fourth dataset had five public attributes; we further removed marital-status, relationship and race attributes. We also ran the algorithm on the full Adult dataset using "flat" hierarchies, namely, generalization by suppression only. Figure 9.4 displays the total algorithm runtime as a function of k in each of the above described tests. The vast differences in the running times as a function of the number of public attributes is due to the fact that the number of closed frequent itemsets depends exponentially on the number of public attributes.

As we can see, our anonymization algorithm, Algorithm 19, provides much better anonymizations than the Forest algorithm, in consistency with the improvement in the approximation factor from $O(k)$ (for the Forest algorithm) to $O(\ln k)$ (for Algorithm 19). The information loss in the solutions produced by Algorithm 19 are also better than that in the solutions issued by the Agglomerative Algorithm, which is a heuristical algorithm with no approximation guarantee. This better performance in terms of information loss is accompanied by slower (but still practical) runtimes for smaller values of the security parameter k. However, as shown in Figure 9.3, already for $k > 100$ Algorithm 19 runs faster than the Agglomerative Algorithm, which was until now the algorithm of choice in terms of information loss.

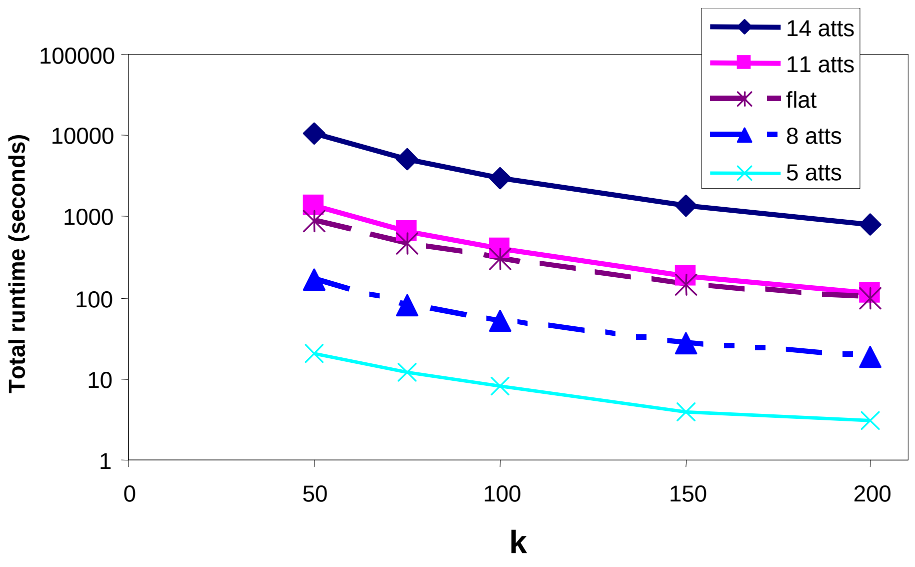


Figure 9.4: k vs. total runtime– Adult dataset

9.2 Nursery dataset experiments

Figure 9.5 displays the LM-scores achieved by the three algorithms on the Nursery dataset. Figure 9.6 displays the runtimes of the three algorithms when run on this dataset. Table 9.3 lists the number of closed frequent itemsets mined as a function of k.

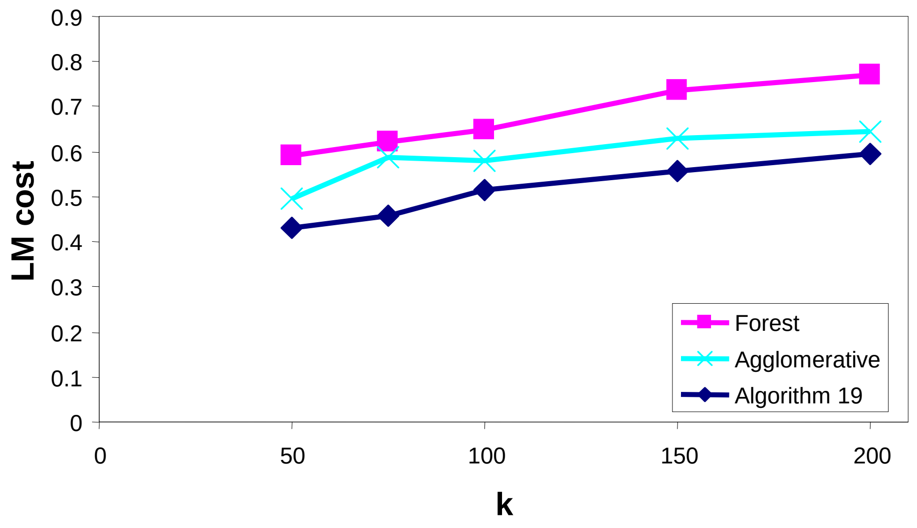


Figure 9.5: Algorithm score comparison (LM) – Nursery dataset

k	Number of closed frequent itemsets
50	28,575
75	17,301
100	13,685
150	7,421
200	5,604

Table 9.3: k vs. the number of closed frequent itemsets – Nursery dataset

In this dataset as well, we see that Algorithm 19 provides better anonymizations than the Forest and the Agglomerative algorithms. Here, it performs much better than the other two algorithms also in terms of runtime. The reason for that is that the Nursery dataset includes only 8 public attributes and, as exemplified earlier on the reduced versions of the Adult dataset, the number of public attributes affects dramatically the overall runtime of Algorithm 19.

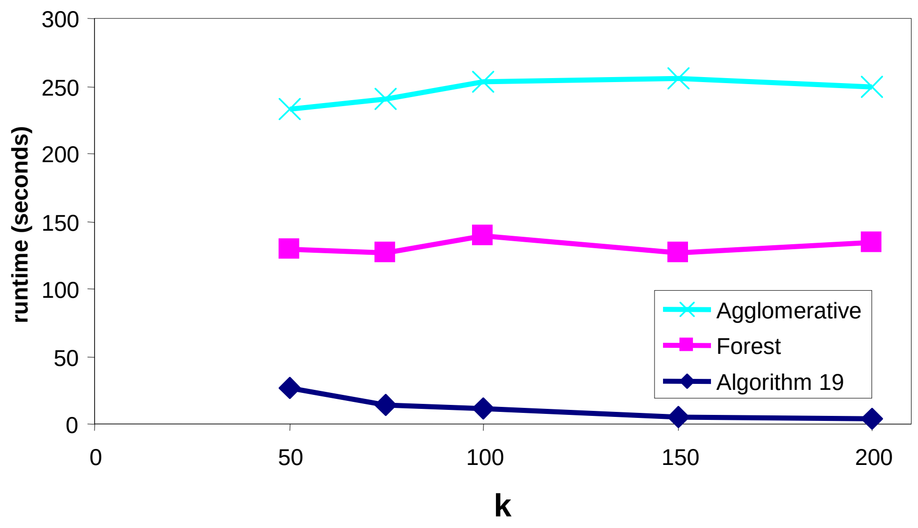


Figure 9.6: Algorithm runtime comparison - Nursery dataset

9.3 Coil2000 dataset experiments

Figure 9.7 displays the LM-scores achieved by the three algorithms on the Coil2000 dataset. Figure 9.8 displays the runtimes of the three algorithms when run on this dataset. Table 9.4 lists the number of closed frequent itemsets mined during the algorithm run.

Here, like in the Nursery dataset, Algorithm 19 performs better than the Forest and Agglomerative algorithms in terms of information loss as well as runtime.

In this experiment it can be seen that the runtime of the Agglomerative algorithm (5) increases with k, as opposed to the behavior of Algorithm 19. The runtime of Algorithm 19 decreases as k increases due to the fact that the number of closed frequent generalized itemsets decreases as k increases, and the runtimes of both phases of Algorithm 19 directly depend the number of closed frequent generalized itemsets. The Agglomerative algorithm, on the other hand, begins with $|D|$ singleton clusters. In each iteration the number of clusters decreases by one. Since the algorithm aims at creating clusters of size close to k, the algorithm terminates when the number of clusters is roughly $\frac{|D|}{k}$. Therefore, the Agglomerative algorithm will go through about $|D| - \frac{|D|}{k} = |D| \cdot (1 - \frac{1}{k})$ iterations, a number which increases as k increases.

k	Number of Closed frequent itemsets
50	14,113
75	8,985
100	6,234
150	3,499
200	2,227

Table 9.4: k vs. the number of closed frequent itemsets- Coil2000 dataset

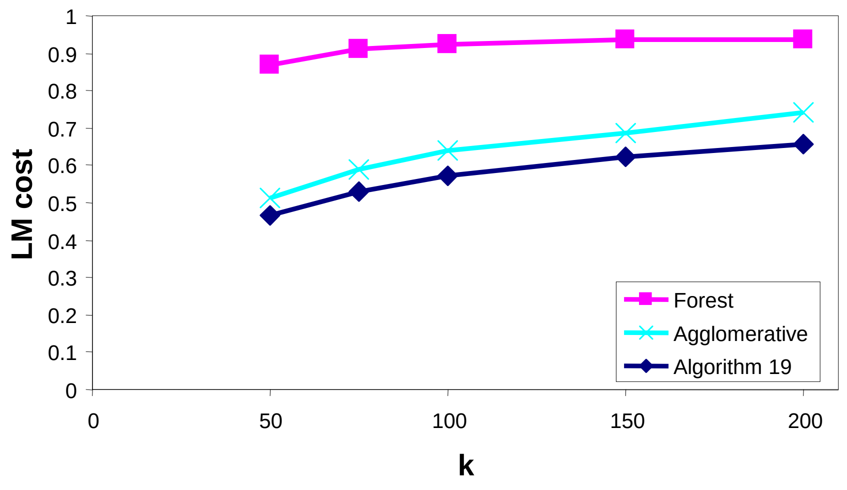


Figure 9.7: Algorithm Comparison - Coil2000 dataset

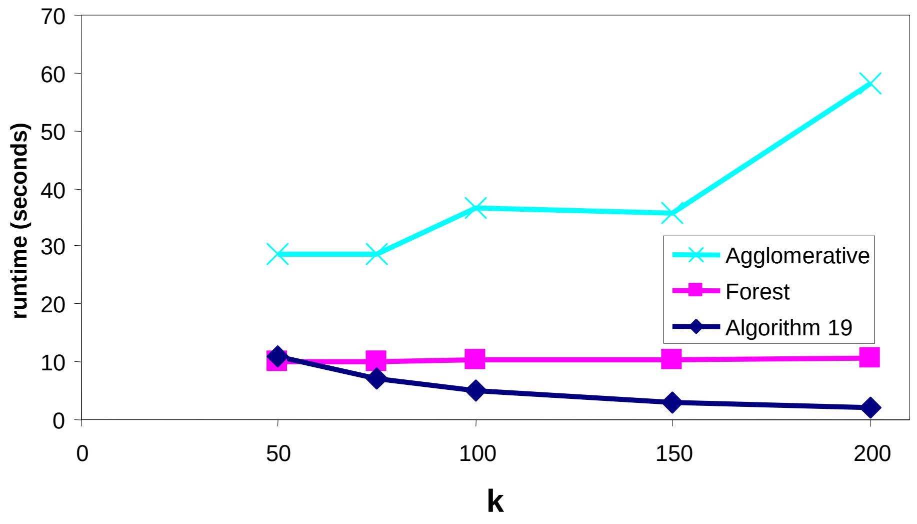


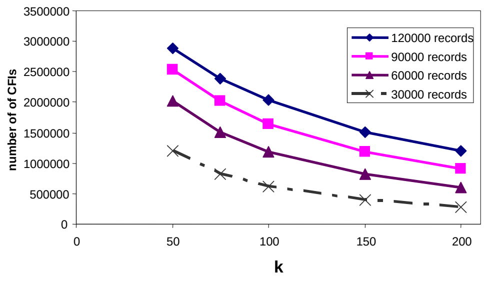
Figure 9.8: Algorithm runtime Comparison - Coil2000 dataset

9.4 Scalability of the proposed mining algorithm

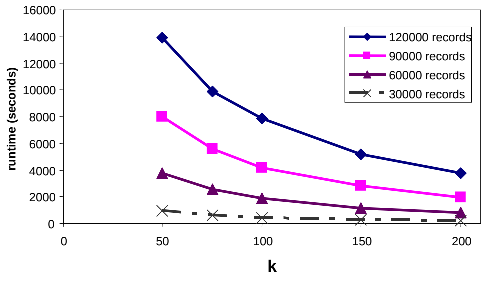
We examined the performance of the the proposed algorithm for mining generalized closed frequent itemsets (Algorithm 16) from two aspects. The first was the dependence on the size of the table (Figure 9.9). For this purpose, we ran the algorithm over four versions of the Adults table; the original table that has roughly 30000 records, and extended versions (that were obtained by means of duplicating records) containing 60000, 90000 and 120000 records. We also tested the algorithm's performance as a function of the number of attributes in the table (Figure 9.10). For this purpose, we ran the algorithm over four versions of the Adults table, containing 14,11,8 and 5 attributes.

According to these experiments we see that the number of generalized closed frequent itemsets grows linearly with the table size (Figure 9.9 (a)). We also observed that the runtime of the algorithm grows quadratically with the table size (Figure 9.9 (b)). This can be explained due to the fact that the algorithm's runtime depends on two factors. First, the number of generalized closed frequent itemsets which are mined. Second, the amount of time taken to perform support computations, which is expected to grow linearly with the size of the table.

On the other hand, there is an exponential dependency between the number of attributes in the table and the number of generalized closed frequent itemsets which can be mined (see Figure 9.10 (a) in which the y axis is scaled logarithmically). Therefore, the runtime of the algorithm depends exponentially on the number of attributes (Figure 9.10 (b)).

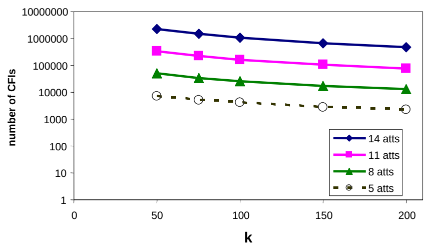


(a) Number of CFIs vs. k

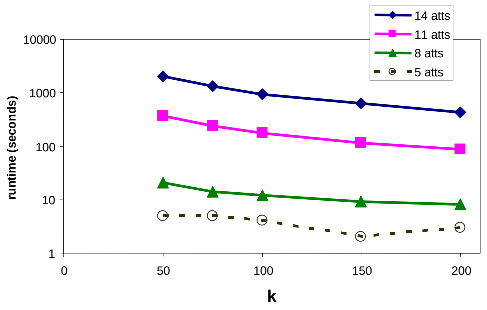


(b) Runtime (seconds) vs. k

Figure 9.9: Table size scaling



(a) Number of CFIs vs. k



(b) Runtime (seconds) vs. k

Figure 9.10: Attribute scaling

Chapter 10

Conclusion

In this study we discussed the concept of k-anonymization as a privacy-preserving method used for publicizing data sources for research and data-mining purposes. In particular, we concentrated on the problem of k-anonymization with minimal information loss. We surveyed several approximation algorithms for that problem. We also described the heuristic Agglomerative algorithm which was shown to provide better anonymizations than the known approximation algorithms.

The main contribution of this study was a practical anonymization algorithm that uses techniques for mining closed generalized frequent itemsets. To that end, we surveyed various algorithms which efficiently mine closed frequent itemsets. Leveraging the ideas of frequent itemset mining algorithms, we introduced an approximation algorithm for the k-anonymization problem that apply for the case of generalization (as opposed to suppressions-only). We went on to show that this approximation algorithm provides smaller information losses than both the best known approximation algorithm as well as the best known heuristic algorithm, and that it is practical in terms of running time.

Bibliography

[1] G. Aggarwal, T. Feder, K. Kenthapadi, R. Motwani, R. Panigrahy, D. Thomas, and A. Zhu, "Approximation algorithms for k-anonymity," in *Proceedings of the International Conference on Database Theory (ICDT)*, 2005.

[2] R. Agrawal and R. Srikant, "Privacy-preserving data mining," in *ACM-SIGMOD Conference on Management of Data*, May 2000, pp. 439–450.

[3] ——, "Fast algorithms for mining association rules," in *Proc. 20th Int. Conf. Very Large Data Bases, VLDB*, 1994, pp. 487–499.

[4] A. Asuncion and D. Newman, "UCI machine learning repository," 2007. [Online]. Available: http://www.ics.uci.edu/∼mlearn/MLRepository.html

[5] R. J. Bayardo and R. Agrawal, "Data privacy through optimal k-anonymization," in *International Conference on Data Engineering (ICDE)*, 2005, pp. 217–228.

[6] A. Gionis, A. Mazza, and T. Tassa, "k-anonymization revisited," in *International Conference on Data Engineering (ICDE)*, 2008, pp. 744–753.

[7] A. Gionis and T. Tassa, "k-anonymization with minimal loss of information," *IEEE Trans. on Knowl. and Data Eng.*, vol. 21, no. 2, pp. 206–219, 2009.

[8] J. Han and M. Kamber, *Data Mining Concepts and Techniques*, 2nd ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2005.

[9] J. Han, J. Pei, Y. Yin, and R. Mao, "Mining frequent patterns without candidate generation: A frequent-pattern tree approach," *Data Min. Knowl. Discov.*, vol. 8, no. 1, pp. 53–87, 2004.

[10] C.-J. Hsiao, “Efficient algorithms for mining closed itemsets and their lattice structure,” *IEEE Trans. on Knowl. and Data Eng.*, vol. 17, no. 4, pp. 462–478, 2005.

[11] V. S. Iyengar, “Transforming data to satisfy privacy constraints,” in *KDD ’02: Proceedings of the eighth ACM SIGKDD international conference on Knowledge discovery and data mining*, 2002, pp. 279–288.

[12] A. Mazza, “Anonymization of databases: New models and algorithms,” Master’s thesis, The Department of Mathematics and Computer Science, The Open University of Israel, 2008.

[13] A. Meyerson and R. Williams, “On the complexity of optimal k-anonymity,” in *PODS ’04: Proceedings of the twenty-third ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, 2004, pp. 223–228.

[14] M. E. Nergiz and C. Clifton, “Thoughts on k-anonymization,” *Data Knowl. Eng.*, vol. 63, no. 3, pp. 622–645, 2007.

[15] H. Park and K. Shim, “Approximate algorithms for k-anonymity,” in *SIGMOD ’07: Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, 2007, pp. 67–78.

[16] J. Pei, J. Han, and R. Mao, “CLOSET: An Efficient Algorithm for Mining Frequent Closed Itemsets,” in *Workshop on Research Issues in Data Mining and Knowledge Discovery, DMKD*, 2000, pp. 21–30.

[17] P. Samarati, “Protecting respondents’ identities in microdata release,” *IEEE Trans. on Knowl. and Data Eng.*, vol. 13, no. 6, pp. 1010–1027, 2001.

[18] P. Samarati and L. Sweeney, “Generalizing data to provide anonymity when disclosing information (abstract),” in *PODS ’98: Proceedings of the seventeenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, 1998, p. 188.

[19] R. Srikant and R. Agrawal, “Mining generalized association rules,” in *VLDB ’95: proceedings of the 21st International Conference on Very Large Data Bases*, 1995, pp. 407–419.

[20] L. Sweeney, “Uniqueness os simple demographics in the u.s. population,” *Laboratory for international Data Privacy (LIDAP-WP4)*, 2000.

[21] ——, “k-anonymity: A model for protecting privacy,” *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems*, vol. 10, no. 5, pp. 557–570, 2002.