Knowledge Compilation for Itemset Mining

Abstract. Mining frequently occurring patterns or itemsets is a fundamental task in datamining. Many ad-hoc itemset mining algorithms have been proposed for enumerating frequent, maximal and closed itemsets. The datamining community has been particularly interested in finding itemsets that satisfy additional constraints, which is a challenging task for existing techniques. In this paper we present a novel approach to itemset mining whereby the set of all itemsets are compiled into a compact form, closely related to binary decision diagrams. While there were previous attempts to utilize decision diagrams for storing the set of frequent itemsets, to the best of our knowledge, this is the first approach that does not rely on backtracking search to generate such a set. Instead, we use a pure knowledge compilation inference mechanism based on pairwise conjunctions of decision diagrams. Our empirical evaluation demonstrates that our approach is complementary to current state-of-the-art approaches. While we do not dominate these approaches, we show that our pure compilation-based method can handle instances that remain out of reach for current state-of-the-art.

Our empirical evaluation demonstrates that our approach is complementary to state-of-the-art itemset miners [19]. While we do not dominate these approaches, we show that our pure compilation-based method can handle instances that remain out of reach for the current state-of-the-art.

The remainder of this paper is organised as follows. In Section 2 we present the formal notation needed throughout this paper. A review of the current state-of-the-art is presented in Section 3. Our compilation approach is developed in Section 4. In Section 5 we describe the experimental evaluation of our scheme. Finally we conclude and outline directions for future work in Section 6.

1 INTRODUCTION

Mining frequent itemsets is a central problem in data mining. It is related to numerous discovery tasks in databases, e.g. association rule mining [1], correlation discovery, sequential pattern mining [8], and many others. Given a database defined as a set of transactions, each of which is defined by a set of items (see Figure 1), the task is to find all sets of items (patterns) that are considered frequent. An itemset is frequent if it occurs at least a specified number of times in the database. However, the set of frequent itemsets might be exponentially large. This has motivated research on more restrictive, but representative, sets such as maximal or closed itemsets. Mining maximal or closed itemsets is often more tractable in practice than frequent itemset mining. However, this tractability comes at the loss of information which might be useful in particular contexts.

Our focus in this paper is on mining frequent itemsets. To help address the scalability issue mentioned above we propose a novel approach that casts itemset mining as a knowledge compilation task. Our compilation-based approach supports a number of powerful queries that involve reasoning about the set of all frequent itemsets in time polynomial in the size of the compiled representation (which might be exponentially large in the worst case). Counting frequent itemsets is one such query. The technical advance is to compile the set of all itemsets into a special form of binary decision diagram (BDD) [5], augmented with counting variables, so that all itemsets of the same frequency are represented by BDD paths that end in the same counting node. This representation is similar to the one used in [18] to compress the database as opposed to the itemsets. It was later abandoned by the same authors in [13]. Our approach not only supports the efficient counting of the number of itemsets for any given frequency threshold, \( \theta \), in real-time, but also supports the efficient answering of a number of advanced queries that involve itemset frequency. The novelty of our approach is that it differs fundamentally from earlier attempts to incorporate knowledge compilation into itemset mining since it does not rely on search at all, but only proceeds by manipulation of BDDs. Therefore, any BDD manipulation package, such as [15], can be easily used with our approach as an itemset mining system.

Our empirical evaluation demonstrates that our approach is complementary to current state-of-the-art itemset miners [19]. While we do not dominate these approaches, we show that our pure compilation-based method can handle instances that remain out of reach for the current state-of-the-art.

Example 1 An example transaction database containing three transactions defined in terms of three items, \( \{I_1, I_2, I_3\} \), is shown in Figure 1. The transactions are \( T_1 = \{I_1, I_2, I_3\} \), \( T_2 = \{I_1, I_3\} \), \( T_3 = \{I_1, I_2\} \) (left). A frequency table, giving for each itemset \( X \) its transaction set and its support is shown on the right of Figure 1.

Binary Decision Diagrams. Binary decision diagrams (BDDs) [3] are rooted directed acyclic graphs \( G = (V, E) \) that represent the set of assignments over a set of Boolean variables, \( \{x_1, \ldots, x_n\} \).

A BDD has a root node, denoted \( r \), and two terminal nodes \( 0 \) and \( 1 \) indicating true and false, respectively. Every nonterminal node is labeled with one of the Boolean variables \( x_i \) and has two outgoing edges, which encode assignments \( x_i = 0 \) and \( x_i = 1 \). This encoding associates every path from the root to terminal \( 1 \), denoted \( p : r \sim 1 \), with a partial assignment \( x(p) \). BDDs are normally ordered unless stated otherwise – variables labeling nodes in a path from the root to the terminal are always in the same order, e.g. \( x_1 \prec \ldots \prec x_n \).
BDD is a divide-and-conquer method and compresses the database into a data structure called an FP-tree (a frequent pattern tree), which is a prefix tree representing the set of transactions. Intermediate databases are compressed into FP-trees as the algorithm branches to facilitate efficient frequency counting. One of the most efficient implementations of this principle is LCM [19]. However, the enumeration of itemsets remains restricted in time and space by the size of the available storage. The exponential growth in the number of patterns can even prevent LCM from dumping the itemsets into storage. To overcome this limitation and represent very large set of itemsets, a method using Zero-suppressed Binary Decision Diagrams (ZBDDs) was proposed [18] and the use of automata was evaluated in [17]. The approach replaces the FP-tree by a ZBDD used both as the internal data structure to represent intermediate databases and as a compact form to maintain the set of solutions. A similar study was performed in [16] and several good variable orderings for the ZBDD to store the itemsets were studied in [14, 12]. It was shown, nonetheless, that performance is not competitive with LCM, especially when the compression rate of the database is not sufficiently high. Those authors, thus, abandoned the use of ZBDDs as the internal data structure and proposed an improved version of LCM in [13] that simply stores the solutions in the ZBDD as they are discovered during search.

To summarize, decision diagrams have, thus far, been used in itemset mining for two main purposes: as a compact form to store the set of solutions and, more recently, to speed up existing backtracking algorithms by replacing their central internal data structure representing conditional databases during search. The work proposed in this paper takes a very different approach by treating the entire itemset mining process as a knowledge compilation task.

### 4 THE COMPILATION APPROACH

Our basic approach is to compile the set of all itemsets into a compact representation so that for each threshold \( \theta \) we can distinguish the set of items whose support is at least \( \theta \). Such a representation would allow us, in real time, to compute a number of properties of frequent itemsets with respect to any threshold.

#### 4.1 The Model

We introduce \( n \) Boolean variables \( x_1, \ldots, x_n \), one for each item. Assignment \( x_j = 1 \) indicates that the item \( I_j \) is in the itemset. Hence, an assignment to \( x_1, \ldots, x_n \) represents an itemset

\[
I(x_1, \ldots, x_n) = \{I_j \mid x_j = 1, j = 1, \ldots, n\}.
\]

We also introduce \( m \) Boolean variables \( t_1, \ldots, t_m \), one for each transaction. An assignment to \( t_1, \ldots, t_m \) represents a set of transactions

\[
T(t_1, \ldots, t_n) = \{T_j \mid t_j = 1, j = 1, \ldots, m\}.
\]

Finally, we introduce \( m \) constraints \( C_1, \ldots, C_m \), one for each transaction \( T_i \):

\[
C_i : \quad t_i \iff \bigwedge_{j \in T_i} \neg x_j, \quad i = 1, \ldots, m
\]

where \( t_i \) is a shorthand for \( t_i = 1 \) and \( \neg x_j \) means \( x_j = 0 \). The constraint \( C_i \) ensures that a transaction \( T_i \) is in a transaction set exactly when none of the items outside the transaction \( (I_j \notin T_i) \) are in the itemset. It is not hard to see that any assignment to \( (x_1, \ldots, x_n, t_1, \ldots, t_m) \) that satisfies all constraints \( C_1, \ldots, C_m \) represents an itemset \( I(x_1, \ldots, x_n) \) and its corresponding transaction set \( T(t_1, \ldots, t_m) \). We refer to these assignments as solutions, and denote the set of all solutions as \( \text{Sol} \).

### 3 PRIOR WORK

Many algorithms have been designed to address the problem of itemset mining. Most of the initial studies focus on what is called an apriori-like approach [1]. These approaches attempt to enumerate the possible itemsets by exploring the lattice of all possible subsets of items \( I \) and using the fundamental idea that if an itemset is discovered to be infrequent, then all supersets are infrequent and can be pruned. However, such an approach is known to be inadequate in situations where there are high cardinality itemsets because it will need to generate all the subsets before discovering them. Many variants of this scheme have been designed to address this issue [9, 6, 4].

Another class of algorithms are those based on backtrack search to avoid generating subsets, which rely on the FP-growth framework [11, 2]. This is a backtracking approach that applies a divide-and-conquer method and compresses the database into a data structure called an FP-tree (a frequent pattern tree), which is a prefix tree representing the set of transactions. Intermediate databases are compressed into FP-trees as the algorithm branches to facilitate efficient frequency counting. One of the most efficient implementations of this principle is LCM [19]. However, the enumeration of itemsets was covered to be infrequent, then all supersets are infrequent and can be pruned. However, such an approach is known to be inadequate in situations where there are high cardinality itemsets because it will need to generate all the subsets before discovering them. Many variants of this scheme have been designed to address this issue [9, 6, 4].

Another class of algorithms are those based on backtrack search to avoid generating subsets, which rely on the FP-growth framework [11, 2]. This is a backtracking approach that applies a divide-and-conquer method and compresses the database into a data structure called an FP-tree (a frequent pattern tree), which is a prefix tree representing the set of transactions. Intermediate databases are compressed into FP-trees as the algorithm branches to facilitate efficient frequency counting. One of the most efficient implementations of this principle is LCM [19]. However, the enumeration of itemsets was covered to be infrequent, then all supersets are infrequent and can be pruned. However, such an approach is known to be inadequate in situations where there are high cardinality itemsets because it will need to generate all the subsets before discovering them. Many variants of this scheme have been designed to address this issue [9, 6, 4].
4.2 The Transaction MTBDDs

The basic idea behind the knowledge compilation is to solve a portion of the problem that does not change frequently offline to be able to quickly process the changing parts of the problem online. In our context, for the same transaction database \( D \), a user might want to get the set of frequent itemsets with respect to different \( \theta \).

This could be achieved by compiling the set of all solutions \( \text{Sol} \) from Section 4.1 into a special form of binary decision diagram, in which each path corresponds to one or more itemsets and which has multiple terminals, one for each encountered transaction set. This corresponds closely to a multi-terminal binary decision diagram (MTBDD), and we refer to it as a transaction MTBDD. The hope is that even though there are many itemsets, many of them would correspond to the same transaction sets and therefore many nodes would become isomorphic. Transaction MTBDDs for our itemset example are shown in Figure 2, for ordering \( x_3 < x_2 < x_1 \). In Figure 2(a) we show the MTBDD where the isomorphic nodes are merged, but redundant nodes are still not eliminated. We can see that despite having 8 paths there are only 4 distinct transaction sets. The reduced version of the same MTBDD is shown in Figure 2(b) where the edges are skipping variable \( x_1 \).

![Figure 2. Transaction BDDs for the itemset example.](image)

Generating such a structure would allow us to extract the itemsets that have the desired minimal support \( \theta \). For example, each terminal node can be associated with the size of its transaction set, and the number of frequent itemsets can be obtained by counting the number of solutions from the root node to those terminals with support at least \( \theta \). Since counting the number of solutions in a BDD is a linear operation \([5]\) in the size of the BDD, we can efficiently count the number of frequent itemsets for any \( \theta \). The corresponding set of itemsets can also be efficiently retrieved in a similar way for any \( \theta \).

4.3 The Count MTBDDs

However, for the purpose of counting and storing frequent itemsets, it is not necessary to distinguish between terminal nodes whenever they do not represent the same transaction sets. Namely, we can treat different terminal nodes as equivalent as long as they correspond to the same number of transactions. Merging terminal nodes based on this criterion might significantly reduce the number of nodes, since for any cardinality there might be exponentially many different transaction sets. We denote such MTBDDs, where each terminal is associated with the size rather than the set, as count MTBDDs. A count MTBDD corresponding to the transaction MTBDD from Figure 2(b) is shown in Figure 3(a). Note how the terminals corresponding to \( \{T_1, T_2\} \) and \( \{T_1, T_3\} \) become isomorphic since they have the same set size. Also note that counting the frequent itemsets for any given \( \theta \) can be efficiently executed over the count MTBDD in exactly the same way as for the transaction MTBDDs.

![Figure 3. Count BDDs for the itemset example.](image)

4.4 Generating Count MTBDDs

The main contribution of this paper is an easy to implement and yet powerful approach to generating count MTBDDs, relying only on the standard BDD operations that are supported by standard BDD packages, such as BuDDy [15].

The first step is to realize that we do not need specialized software to manipulate MTBDDs. We can encode the count MTBDD as a simple BDD by introducing an extra multi-valued counting variable \( y \) using the well known log encoding. For a model with \( m \) transactions \( y \in [0, m] \) and it takes \( k = \lceil \log(m) \rceil \) Boolean variables to encode any value in \([0, m]\) as a number in binary notation. For our itemset example, \( m = 3 \) and therefore we need \( k = 2 \) Boolean variables, denoted as \( b_1, b_2 \). Value \( y = 3 \) corresponds to \( b_1 = 1, b_2 = 1 \), \( y = 2 \) corresponds to \( b_1 = 0, b_2 = 1 \), \( y = 1 \) is equivalent to \( b_1 = 1, b_2 = 0 \) and \( y = 0 \) to \( b_1 = 0, b_2 = 0 \). The BDD that encodes the count MTBDD from Figure 3(a) is shown in Figure 3(b). We will refer to such BDDs as count BDDs. Note that for simplicity of representation we do not draw edges to the terminal \( 0 \).

The second and most important step is to develop a method that generates the count BDDs. The count BDD representation is quite close to existing representations based on zero-suppressed decision diagrams (ZBDDs) that are generated in the datamining community [18]. In an analogous way to the generation of ZBDDs, one could use backtracking search to store the itemsets and their support in the corresponding BDDs. However, we take a different route and generate count BDDs through pairwise conjunctions of atomic BDDs in line with traditional approaches to BDD generation. Pairwise conjunctions are based on the well known dynamic programming scheme.
(the apply operator), and are guaranteed to produce the BDD corresponding to the desired Boolean operator (conjunction, disjunction, implication, etc.) in quadratic time and space (in the number of nodes of the input BDDs). By caching and storing intermediate results of dynamic programming recursion one is achieving an essentially different inference mechanism, that is incomparable to the inference made through backtracking search. For example, it is a well known fact that the BDD-based inference and resolution-based inference are mutually incomparable [10].

The Compilation Algorithm. The statement of the compilation procedure is presented in Algorithm 1. We start by generating a BDD for each constraint $C_i$ (line 1). The count BDD, denoted $B_{Sol}$, is initialized to represent the database consisting of a single transaction $\{T_1\}$ by transforming a BDD for the first constraint $B_1$. The transaction variable $t_1$ is replaced with the counting variable $y$ so that every itemset that is covered by transaction $T_1$ (i.e. where transaction variable $t_1$ has value 1) is now associated with the count $y = 1$. This is achieved in line 2 of the algorithm by conjoining $B_1$ with a BDD representation of the expression $y = t_1$. In addition, transaction variable $t_1$ is no longer necessary and therefore is removed from the counting BDD, through existential quantification $\exists t_1$.

Then, in an iterative fashion we keep updating the count BDD $B_{Sol}$ so that in each step $i$, the count BDD representation of the database $\{T_1, \ldots, T_i\}$ is updated to represent the database $\{T_1, \ldots, T_i, T\}$. After all the iterations are completed, $B_{Sol}$ will represent a count BDD for the entire database $\{T_1, \ldots, T_m\}$.

The update of $B_{Sol}$ with respect to $B_1$ is made in two steps (lines 3 and 4). In the first and the most critical step, we update the frequency counts (represented by variable $y$) of a current count BDD $B_{Sol}$ with respect to the BDD $B_1$. This is achieved symbolically, by using a set of copy variables $y'$ and conjoining $B_{Sol} \land B_1$ with a BDD corresponding to the expression $y' = y + t_i$. This conjunction is a critical computational mechanism, that performs the update of frequency counts for all the itemsets simultaneously. This operation does not depend on the number of itemsets, but on the number of nodes in the corresponding BDDs. After the conjoining we get the BDD whose nodes are labeled with itemset variables $x_1, \ldots, x_n$, variable $t_1$ and variables $y', y$. Since the updated frequency counts are stored in $y'$, we can existentially quantify variables $t_1$ and $y (\exists t_1, \exists y)$. The resulting BDD is a count BDD where $y'$ variables denote the counts. Before the next iteration can start, we need to replace the nodes labeled with $y'$ variables with the equivalent nodes labeled with $y$ variables. This is done in line 4 by calling a bdd_replace operator (supported by BuDDy).

Algorithm 1: BDDCount($C_1, \ldots, C_m$): Compiles count BDD for the database $\{T_1, \ldots, T_m\}$.

1. $B_1 \leftarrow BDD(C_i), i = 1, \ldots, m$;
2. $B_{Sol} \leftarrow \exists t_1(B_1 \land BDD(y = t_1))$;
3. foreach $i = 2, \ldots, m$ do
   4. $B_{Sol} \leftarrow \exists t_1, \ldots, \exists t_m (B_{Sol} \land B_1 \land BDD(y' = y + t_i))$
   5. $B_{Sol} \leftarrow bdd_replace(B_{Sol}, y' \rightarrow y)$;

return $B_{Sol}$.

5 EXPERIMENTS

The performance of Algorithm 1 heavily relies on the ordering of the itemset variables $\{x_1, \ldots, x_n\}$ and finding the best ordering is a well known NP-hard problem. In these experiments, we ordered the items by increasing support (number of transactions in which it appears) as it outperformed other ordering heuristics we tried.

We evaluated our approach on all the datamining instances used in [7] that were collected from the UCI machine learning repository. In total, there are 15 instances involving between 31 and 287 items and between 101 and 8124 transactions. The experiments ran as a single thread on a Dual Quad Core Xeon CPU, 2.66GHz with 12MB of L2 cache per processor and 16GB of RAM overall, running Linux 2.6.25 x86. The baseline used for comparison is the state of the art itemsets miner LCM [19]. We will refer to LCM count as the version of LCM that only counts the number of itemsets without storing them. LCM was extended with ZBDDs to be able to store vast number of solutions leading to the approach LCM over ZBDDs described in [13] where very large sets of itemsets can be handled with little overhead compared to LCM count. LCM over ZBDDs is to our knowledge the very best approaches for itemset mining. We use LCM count as the base line because it is available publicly and constitutes a lower bound on the time needed by LCM over ZBDDs (see [13]) which allows us to draw fair comparisons. Note that our approach also stores the sets of itemsets for any threshold.

Overall Results. Table 1 reports the results. The number of items $n$ and number of transactions $m$ of each instance are indicated and results are reported for two compilation schemes. The column “BDD all 0” reports results for computing count BDDs using Algorithm 1. Column msize shows the number of transactions that were conjoined successfully. This number can be smaller than the value of $m$ in which case we ran out of memory and the compilation was not completed. Column BDD size reports the number of BDD nodes in the resulting BDD (in case of unsuccessful termination, this was the size of BDD at the last successful iteration). The second compilation scheme, “BDD 1%”, attempts to build a BDD restricted to a given threshold of 1% by merging all counting nodes for frequencies greater than $\theta$, and pruning itemsets with count 0 in the final BDD. Finally the last column reports the time needed by LCM count for the same threshold of 1%.

The results are globally positive as 9 out of 15 instances can be solved for all thresholds at once by the general compilation approach. The method fails for memory reasons on 6 instances (indicated with a *) of which 3 can be handled when restricting the compilation to a threshold of 1%. LCM count is only comparable to the restricted scheme “BDD 1%” and proves to be much faster in general. However it fails on Audiology which appears to be easy even for the complete compilation (44s). It highlights that the compilation method can complement traditional approaches based on search and can handle instances that are at the moment out of reach of backtracking based methods – perhaps for the same reasons that make the BDDs and resolution incomparable inference methods [10]. Further investigations are needed to fully understand the reasons that make Audiology so easy for compilation and so hard for backtracking. We believe that this example is a very positive and promising result.

Detailed Analysis. We examine in more details three instances presenting different behaviors: Anneal, Mushroom and Audiology. In Figure 4, for each frequency threshold $\theta \in [1, m]$, the middle and the left plots report on a logarithmic scale the number of solutions as well as the size (number of nodes) of $BDD(\sigma(\theta) \geq \theta)$ and $BDD(\sigma(\theta) = \theta)$, respectively. The plots show that the restricted

1 http://www.cs.kuleuven.ac.be/ dtai/CP4IM/datasets/
Table 1. Comparing the general compilation scheme “BDD all $\theta$” with a restricted one, “BDD 1\%” and LCM count

<table>
<thead>
<tr>
<th>Instance</th>
<th>$n$</th>
<th>$m$</th>
<th>$m_{succ}$</th>
<th>BDD size</th>
<th>Time</th>
<th>$\theta$</th>
<th>$m_{succ}$</th>
<th>BDD size</th>
<th>Time</th>
<th>LCM count 1%</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Zoo-1</td>
<td>36</td>
<td>101</td>
<td>101</td>
<td>6750</td>
<td>0m 0.06s</td>
<td>1</td>
<td>101</td>
<td>1386</td>
<td>0m 0.28s</td>
<td>0m 0.1s</td>
<td></td>
</tr>
<tr>
<td>Vote</td>
<td>48</td>
<td>435</td>
<td>435</td>
<td>258138</td>
<td>0m 24s</td>
<td>4</td>
<td>435</td>
<td>39992</td>
<td>0m 11.5s</td>
<td>0m 0.3s</td>
<td></td>
</tr>
<tr>
<td>Tic-tac-toe</td>
<td>27</td>
<td>958</td>
<td>958</td>
<td>30863</td>
<td>0m 1.2s</td>
<td>10</td>
<td>958</td>
<td>5305</td>
<td>0m 14.3s</td>
<td>0m 0.1s</td>
<td></td>
</tr>
<tr>
<td>*Splice-1</td>
<td>287</td>
<td>3190</td>
<td>161</td>
<td>19247569</td>
<td>16m 20s</td>
<td>32</td>
<td>161</td>
<td>1924673</td>
<td>23m 33s</td>
<td>0m 19.8s</td>
<td></td>
</tr>
<tr>
<td>Soybean</td>
<td>30</td>
<td>630</td>
<td>630</td>
<td>50451</td>
<td>0m 8s</td>
<td>6</td>
<td>630</td>
<td>9694</td>
<td>0m 6.6s</td>
<td>0m 0.1s</td>
<td></td>
</tr>
<tr>
<td>Primary-tumor</td>
<td>31</td>
<td>336</td>
<td>336</td>
<td>46569</td>
<td>0m 2s</td>
<td>3</td>
<td>336</td>
<td>5934</td>
<td>0m 0.9s</td>
<td>0m 0.1s</td>
<td></td>
</tr>
<tr>
<td>Mushroom</td>
<td>119</td>
<td>8124</td>
<td>8124</td>
<td>130413</td>
<td>0m 3s</td>
<td>31</td>
<td>8124</td>
<td>29247</td>
<td>8m 46s</td>
<td>0m 0.3s</td>
<td></td>
</tr>
<tr>
<td>*Re-vs-kp</td>
<td>35</td>
<td>1906</td>
<td>1314</td>
<td>15850148</td>
<td>197m /s</td>
<td>32</td>
<td>1314</td>
<td>19017648</td>
<td>251m 28s</td>
<td>12m 55.6s</td>
<td></td>
</tr>
<tr>
<td>*Hypothyroid</td>
<td>88</td>
<td>3247</td>
<td>2124</td>
<td>13516008</td>
<td>146m 15s</td>
<td>32</td>
<td>2124</td>
<td>1052365</td>
<td>18m 3.2s</td>
<td>18m 3.2s</td>
<td></td>
</tr>
<tr>
<td>Hepatitis</td>
<td>68</td>
<td>137</td>
<td>137</td>
<td>1583304</td>
<td>0m 44s</td>
<td>1</td>
<td>137</td>
<td>18427</td>
<td>0m 6.7s</td>
<td>0m 12.2s</td>
<td></td>
</tr>
<tr>
<td>*Heart-cleveland</td>
<td>95</td>
<td>296</td>
<td>296</td>
<td>21279153</td>
<td>23m 59s</td>
<td>3</td>
<td>296</td>
<td>1405948</td>
<td>29m 35s</td>
<td>9m 24.1s</td>
<td></td>
</tr>
<tr>
<td>*German-credit</td>
<td>112</td>
<td>1000</td>
<td>505</td>
<td>19265912</td>
<td>54m 12s</td>
<td>10</td>
<td>505</td>
<td>19539142</td>
<td>100m 36s</td>
<td>8m 5.3s</td>
<td></td>
</tr>
<tr>
<td>*Australian-credit</td>
<td>135</td>
<td>653</td>
<td>292</td>
<td>na</td>
<td>20m 33s</td>
<td>7</td>
<td>280</td>
<td>24073775</td>
<td>45m 0s</td>
<td>45m 19.7s</td>
<td></td>
</tr>
<tr>
<td>Audiology</td>
<td>148</td>
<td>216</td>
<td>216</td>
<td>3185830</td>
<td>0m 44s</td>
<td>2</td>
<td>216</td>
<td>26265</td>
<td>0m 1.97s</td>
<td>&gt; 10 hours</td>
<td></td>
</tr>
<tr>
<td>Anneal</td>
<td>93</td>
<td>812</td>
<td>812</td>
<td>931935</td>
<td>3m 40s</td>
<td>8</td>
<td>812</td>
<td>23164</td>
<td>0m 14.11s</td>
<td>0m 4.9s</td>
<td></td>
</tr>
</tbody>
</table>

Figure 4. Detailed frequency profile for instances: Anneal, Audiology and Mushroom.
BDD is not necessarily much smaller than the complete BDD. The number of solutions follow a similar trend for the three instances and can reach around $10^{20}$ solutions on Audiology for low thresholds whereas the sizes of the BDDs remain under 10$^7$.

The plots on the right report the time of LCM count for each $\theta$ as well as the time for compiling the general BDD (the straight line on the plots) and the time for querying the BDD to obtain the counts. The three instances show three different scenarios: Audiology is completely intractable for LCM which is reaching the cut off of two hours as soon as $\theta$ falls below 194 (out of 216 transactions). Both the compilation and querying time are thus very advantageous here. Mushroom is the opposite case where the response time of LCM is so low for any threshold that it is comparable to the querying time in the BDD so that compilation is not beneficial. Finally Anneaal is an intermediate case where the compilation is still worthwhile compared to the sum of the times needed by LCM for all thresholds but LCM remains faster if we are only interested in one single threshold. In other words, the counting BDD could pay off in an incremental setting where many queries are performed.

Finally, Figure 5 shows the compilation curve for Audiology. For each conjunction step of Algorithm 1 we report the size of the corresponding count BDD $B_{Sol}$. We can see that the size of the BDD grows gracefully. In particular, it does not exhibit a frequently occurring compilation problem whereby the size of intermediate BDDs is bigger than the final BDD. The same graceful compilation behavior is observed in other instances.

![Figure 5. The compilation curve for Audiology.](image-url)

### 6 CONCLUSIONS AND FUTURE WORK

Pattern discovery is typically an iterative task where queries are refined depending on the answers to previous queries. It is therefore important to develop techniques for incremental mining. The compilation approach presented in this paper addresses this issue by proposing a compact form to allow efficient queries regarding any threshold of the frequency. The resulting method can be surprisingly competitive with well established state-of-the-art itemset miners that benefit from years of maturity. Although it is generally slower, it can succeed where the later fails as shown on the case of the Audiology transaction database. The method seems complementary to the traditional backtracking schemes and can deal efficiently with low thresholds of frequencies. It should be therefore considered as a valid alternative to standard datamining approaches.

The literature dealing with BDDs and itemsets reports that ZBBDs are more compact when representing the set of all itemsets [18, 16]. We therefore plan to investigate whether the same holds for our approach. Improvements could also be made by exploring the hybrid versions of the compilation scheme, where less information is compiled offline and more computational effort is spent online. One example is clustering the transactions and adopting the itemset counting over the set of small BDDs rather than a unique large BDD.

### REFERENCES


