Artificial Intelligence 175 (2011) 1951–1983

Contents lists available at ScienceDirect

Artificial Intelligence

www.elsevier.com/locate/artint

Itemset mining: A constraint programming perspective

Tias Guns *, Siegfried Nijssen, Luc De Raedt

Katholieke Universiteit Leuven, Celestijnenlaan 200A, 3001 Leuven, Belgium

ARTICLE INFO

Article history:
Received 31 May 2010
Received in revised form 5 May 2011
Accepted 6 May 2011
Available online 11 May 2011

Keywords:
Data mining
Itemset mining
Constraint programming

ABSTRACT

The field of data mining has become accustomed to specifying constraints on patterns of interest. A large number of systems and techniques has been developed for solving such constraint-based mining problems, especially for mining itemsets. The approach taken in the field of data mining contrasts with the constraint programming principles developed within the artificial intelligence community. While most data mining research focuses on algorithmic issues and aims at developing highly optimized and scalable implementations that are tailored towards specific tasks, constraint programming employs a more declarative approach. The emphasis lies on developing high-level modeling languages and general solvers that specify *what* the problem is, rather than outlining *how* a solution should be computed, yet are powerful enough to be used across a wide variety of applications and application domains.

This paper contributes a declarative constraint programming approach to data mining. More specifically, we show that it is possible to employ off-the-shelf constraint programming techniques for modeling and solving a wide variety of constraint-based itemset mining tasks, such as frequent, closed, discriminative, and cost-based itemset mining. In particular, we develop a basic constraint programming model for specifying frequent itemsets and show that this model can easily be extended to realize the other settings. This contrasts with typical procedural data mining systems where the underlying procedures need to be modified in order to accommodate new types of constraint, or novel combinations thereof. Even though the performance of state-of-the-art data mining systems outperforms that of the constraint programming approach on some standard tasks, we also show that there exist problems where the constraint programming approach leads to significant performance improvements over state-of-the-art methods in data mining and as well as to new insights into the underlying data mining problems. Many such insights can be obtained by relating the underlying search algorithms of data mining and constraint programming systems to one another. We discuss a number of interesting new research questions and challenges raised by the declarative constraint programming approach to data mining.

1. Introduction

Itemset mining is probably the best studied problem in the data mining literature. Originally applied in a supermarket setting, it involved finding frequent itemsets, that is, sets of items that are frequently bought together in transactions of customers [1]. The introduction of a wide variety of other constraints and a range of algorithms for solving these *constraint-based itemset mining* problems [33,5,41,42,11,31,50,9] has enabled the application of itemset mining to numerous other

* Corresponding author. Tel.: +32 16 32 75 67; fax: +32 16 32 79 96.
E-mail address: tias.guns@cs.kuleuven.be (T. Guns).

0004-3702/$ – see front matter
doi:10.1016/j.artint.2011.05.002

problems, ranging from web mining to bioinformatics [31]; for instance, whereas early itemset mining algorithms focused on finding itemsets in unsupervised, sparse data, nowadays closed itemset mining algorithms enable the application of itemset mining on dense data [40,43], while discriminative itemset mining algorithms allow for their application on supervised data [35,13]. This progress has resulted in many effective and scalable itemset mining systems and algorithms, usually optimized to specific tasks and constraints. This procedural and algorithmic focus can make it non-trivial to extend such systems to accommodate new constraints or combinations thereof. The need to allow user-specified combinations of constraints is recognized in the data mining community, as witnessed by the development of a theoretical framework based on (anti-)*monotonicity* [33,41,11] and systems such as ConQueSt [9], MusicDFS [50] and Molfea [18]. These systems support a predefined number of (anti-)monotonicity based constraints, making them well suited for a number of typical data mining tasks.

These approaches contrast with those of constraint programming. Constraint programming is a general *declarative* methodology for solving constraint satisfaction problems, meaning that constraint programs specify *what* the problem is, rather than outline *how* the solution should be computed; it does not focus on a particular application. Constraint programming systems provide declarative modeling languages in which many types of constraints can be expressed and combined; they often support a much wider range of constraints than more specialized systems such as satisfiability (SAT) and integer linear programming (ILP) solvers [10]. To realize this, the *model* is separated as much as possible from the *solver*. In the past two decades, constraint programming has developed expressive high-level modeling languages as well as solvers that are powerful enough to be used across a wide variety of applications and domains such as scheduling and planning [45].

The question that arises in this context is whether these constraint programming principles can also be applied to itemset mining. As compared to the more traditional constraint-based mining approach, this approach would specify data mining models using general and declarative constraint satisfaction primitives, instead of specialized primitives; this should make it easy to incorporate new constraints and combinations thereof as – in principle – only the model needs to be extended to specify the problem and general purpose solvers can be used for computing solutions.

The contribution of this article is that we answer the above question positively by showing that the general, off-the-shelf constraint programming methodology can indeed be applied to the specific problems of constraint-based itemset mining.[1] We show how a wide variety of itemset mining problems (such as frequent, closed and cost-based) can be modeled in a constraint programming language and that general purpose out-of-the-box constraint programming systems can effectively deal with these problems.

While frequent, closed and cost-based itemset mining are ideal cases, for which the existing constraint programming modeling language used suffices to tackle the problems, this cannot be expected in all cases. Indeed, in our formulation of discriminative itemset mining, we introduce a novel primitive by means of a *global constraint*. This is common practice in constraint programming, and the identification and study of global constraints that can effectively solve specific subproblems has become a branch of research on its own [6]. Here, we have exploited the ability of constraint programming to serve as an integration platform, allowing for the free combination of new primitives with existing ones. This property allows to find closed *discriminative* itemsets effectively, as well as discriminative patterns adhering to any other constraint(s). Furthermore, casting the problem within a constraint programming setting also provides us with new insights in how to solve discriminative pattern mining problems that lead to important performance improvements over state-of-the-art discriminative data mining systems.

A final contribution is that we compare the resulting declarative constraint programming framework to well-known state-of-the-art algorithms in data mining. It should be realized that any such comparison is difficult to perform; this already holds when comparing different data mining (resp. constraint programming) systems to one another. In our comparison we focus on high-level concepts rather than on specific implementation issues. Nevertheless, we demonstrate the feasibility of our approach using our CP4IM implementation that employs the state-of-the-art constraint programming library Gecode [47], which was developed for solving general constraint satisfaction problems. While our analysis reveals some weaknesses when applying this particular library to some itemset mining problem, it also reveals that Gecode can already outperform state-of-the-art data mining systems on some tasks. Although outside the scope of the present paper, it is an interesting topic of ongoing research [37] to optimize constraint programming systems for use in data mining.

The article is organized as follows. Section 2 provides an introduction to the main principles of constraint programming. Section 3 introduces the basic problem of frequent itemset mining and discusses how this problem can be addressed using constraint programming techniques. The following sections then show how alternative itemset mining constraints and problems can be dealt with using constraint programming: Section 4 studies closed itemset mining, Section 5 considers discriminative itemset mining, and Section 6 shows that the typical monotonicity-based problems studied in the literature can also be addressed in the constraint programming framework. We also study in these sections how the search of the constraint programming approach compares to that of the more specialized approaches. The CP4IM approach is then evaluated in Section 7, which provides an overview of the choices made when modeling frequent itemset mining in a concrete constraint programming system and compares the performance of this constraint programming system to specialized data mining systems. Finally, Section 8 concludes.

[1] We studied this problem in two conference papers [16,38] and brought it to the attention of the AI community [17]. This article extends these earlier papers with proofs, experiments and comprehensive comparisons with related work in the literature.

2. Constraint programming

In this section we provide a brief summary of the most common approach to constraint programming. More details can be found in text books [45,3]; we focus on high-level principles and omit implementation issues.

Constraint programming (CP) is a declarative programming paradigm: the user specifies a problem in terms of its constraints, and the system is responsible for finding solutions that adhere to the constraints. The class of problems that constraint programming systems focus on are constraint satisfaction problems.

Definition 1 *(Constraint Satisfaction Problem (CSP)).* A CSP $\mathcal{P} = (\mathcal{V}, D, \mathcal{C})$ is specified by

- a finite set of variables $\mathcal{V}$;
- an initial domain D, which maps every variable $v \in \mathcal{V}$ to a set of possible values $D(v)$;
- a finite set of constraints $\mathcal{C}$.

A variable $x \in \mathcal{V}$ is called *fixed* if $|D(x)| = 1$; a domain D is *fixed* if all its variables are fixed, $\forall x \in \mathcal{V}$: $|D(x)| = 1$. A domain D' is called *stronger* than domain D if $D'(x) \subseteq D(x)$ for all $x \in \mathcal{V}$; a domain is *false* if there exists an $x \in \mathcal{V}$ such that $D(x) = \emptyset$. A *constraint* $C(x_1, \ldots, x_k) \in \mathcal{C}$ is an arbitrary boolean function on variables $\{x_1, \ldots, x_k\} \subseteq \mathcal{V}$.

A solution to a CSP is a fixed domain D' stronger than the initial domain D that satisfies all constraints. Abusing notation for a fixed domain, we must have that $\forall C(x_1, \ldots, x_k) \in \mathcal{C}$: $C(D'(x_1), \ldots, D'(x_k)) = \mathit{true}$.

A distinguishing feature of CP is that it does not focus on a specific set of constraint types. Instead it provides general principles for solving problems with any type of variable or constraint. This sets it apart from satisfiability (SAT) solving, which focuses mainly on boolean formulas, and from integer linear programming (ILP), which focuses on linear constraints on integer variables.

Example 1. Assume we have four people that we want to allocate to two offices, and that every person has a list of other people that he does not want to share an office with. Furthermore, every person has identified rooms he does not want to occupy. We can represent an instance of this problem with four variables which represent the persons, and inequality constraints which encode the room-sharing constraints:

$$D(x_1) = D(x_2) = D(x_3) = D(x_4) = \{1, 2\},$$
$$\mathcal{C} = \{x_1 \neq 2, x_1 \neq x_2, x_3 \neq x_4\}.$$

The simplest algorithm to solve CSPs enumerates all possible fixed domains, and evaluates all constraints on each of these domains; clearly this approach is inefficient. Most CP systems perform a more intelligent type of depth-first search, as given in Algorithm 1 [47,45]:

Algorithm 1 Constraint-Search(D)

```
1: D := propagate(D)
2: if D is a false domain then
3:     return
4: end if
5: if ∃x ∈ V: |D(x)| > 1 then
6:     x := arg min_{x∈V, D(x)>1} f(x)
7:     for all d ∈ D(x) do
8:         Constraint-Search(D ∪ {x ↦ {d}})
9:     end for
10: else
11:     Output solution
12: end if
```

Other search strategies have been investigated as well [53,44], but we restrict ourselves here to the most common case. In each node of the search tree the algorithm branches by restricting the domain of one of the variables not yet fixed (line 7 in Algorithm 1). It backtracks when a violation of a constraint is found (line 2). The search is further optimized by carefully choosing the variable that is fixed next (line 6); here a function $f(x)$ ranks variables, for instance, by determining which variable is involved in the highest number of constraints.

The main concept used to speed up the search is constraint propagation (line 1). Propagation reduces the domains of variables such that the domain remains *locally consistent*. One can define many types of local consistencies, such as node consistency, arc consistency and path consistency; see [45]. In general, in a locally consistent domain, a value d does not occur in the domain of a variable x if it can be determined that there is no solution D' in which $D'(x) = \{d\}$. The main

motivation for maintaining local consistencies is to ensure that the backtracking search does not unnecessarily branch over such values, thereby significantly speeding up the search.

To maintain local consistencies, *propagators* or propagation rules are used. Each constraint is implemented by a propagator. Such a propagator is activated when the domain of one of the variables of the constraint changes. A propagator takes the domain as input and outputs a failed domain in case the constraint can no longer be satisfied, i.e. if there exists no

$$\text{fixed } D' \text{ stronger than } D \text{ with } C\big(D'(x_1), \ldots, D'(x_k)\big) = \text{true}. \tag{1}$$

When possible, the propagator will remove values from the domain that can never satisfy the constraint, giving as output a stronger, locally consistent domain. More formally, a value c should be removed from the domain of a variable $\tilde{x}$ if there is no

$$\text{fixed } D' \text{ stronger than } D \text{ with } D'(\tilde{x}) = c \text{ and } C\big(D'(x_1), \ldots, D'(x_k)\big) = \text{true}. \tag{2}$$

This is referred to as propagation; propagation ensures *domain consistency*. The repeated application of propagators can lead to increasingly stronger domains. Propagators are repeatedly applied until a *fixed point* is reached in which the domain does not change any more.

Consider the constraint $x_1 \neq x_2$, the corresponding propagator is given in Algorithm 2:

Algorithm 2 Conceptual propagator for $x_1 \neq x_2$

```
1: if |D(x1)| = 1 then
2:     D(x2) = D(x2) \ D(x1)
3: end if
4: if |D(x2)| = 1 then
5:     D(x1) = D(x1) \ D(x2)
6: end if
```

The propagator can only propagate when x_1 or x_2 is fixed (lines 1 and 4). If one of them is, its value is removed from the domain of the other variable (lines 2 and 5). In this propagator there is no need to explicitly check whether the constraint is violated, as a violation results in an empty and thus false domain in line 2.

Example 2 *(Example 1 continued).* The initial domain of this problem is not consistent: the constraint $x_1 \neq 2$ cannot be satisfied when $D(x_1) = \{2\}$ so value 2 is removed from the domain of x_1. Subsequently, the propagator for the constraint $x_1 \neq x_2$ is activated, which removes value 1 from $D(x_2)$. At this point, we obtain a fixed point with $D(x_1) = \{1\}$, $D(x_2) = \{2\}$, $D(x_3) = D(x_4) = \{1, 2\}$. Persons 1 and 2 have now each been allocated to an office, and two rooms are possible for persons 3 and 4. The search branches over x_3 and for each branch, constraint $x_3 \neq x_4$ is propagated; a fixed point is then reached in which every variable is fixed, and a solution is found.

In the above example for every variable its entire domain $D(x)$ is maintained. In constraint programming many types of consistency and algorithms for maintaining consistency have been studied. A popular type of consistency is *bound consistency*. In this case, for each variable only a lower- and an upper-bound on the values in its domain is maintained. A propagator will try to narrow the domain of a variable to that range of values for which it still believes a solution can be found, but does not maintain consistency for individual values. To formulate itemset mining problems as constraint programming models, we mostly use variables with binary domains, i.e. $D(x) = \{0, 1\}$ with $x \in \mathcal{V}$. For such variables there is no difference between bound and domain consistency.

Furthermore, we make extensive use of two types of constraints over boolean variables, namely the *summation constraint*, Eq. (3), and *reified summation constraint*, Eq. (6), which are introduced below.

2.1. Summation constraint

Given a set of variables $V \subseteq \mathcal{V}$ and weights w_x for each variable $x \in V$, the general form of the summation constraint is:

$$\sum_{x \in V} w_x x \geqslant \theta. \tag{3}$$

The first task of the propagator is to discover as early as possible whether the constraint is violated. To this aim, the propagator needs to determine whether the upper-bound of the sum is still above the required threshold; filling in the constraint of Eq. (1), this means we need to check whether:

$$\max_{\text{fixed } D' \text{ stronger than } D} \left(\sum_{x \in V} w_x D'(x) \right) \geqslant \theta. \tag{4}$$

A more intelligent method to evaluate this property works as follows. We denote the maximum value of a variable x by $x^{max} = \max_{d \in D(x)} d$, and the minimum value by $x^{min} = \min_{d \in D(x)} d$. Denoting the set of variables with a positive, respectively negative, weight by $V^+ = \{x \in V \mid w_x \geqslant 0\}$ and $V^- = \{x \in V \mid w_x < 0\}$, the bounds of the sum are now defined as:

$$\max\left(\sum_{x \in V} w_x x\right) = \sum_{x \in V^+} w_x x^{max} + \sum_{x \in V^-} w_x x^{min}, \qquad \min\left(\sum_{x \in V} w_x x\right) = \sum_{x \in V^+} w_x x^{min} + \sum_{x \in V^-} w_x x^{max}.$$

These bounds allow one to determine whether an inequality constraint $\sum_{x \in V} w_x x \geqslant \theta$ can still be satisfied.

The second task of the propagator is to maintain the bounds of the variables in the constraint, which in this case are the variables in V. In general, for every variable $\tilde{x} \in V$, we need to update $\tilde{x}^{min}$ to the lowest value c for which there exists a domain D' with

$$\text{fixed } D' \text{ stronger than } D, \quad D'(\tilde{x}) = c \quad \text{and} \quad \left(\sum_{x \in V} w_x D'(x)\right) \geqslant \theta. \tag{5}$$

Also this can be computed efficiently; essentially, for binary variables $x \in V$ we can update all domains as follows:

- $D(x) \leftarrow D(x) \setminus \{0\}$ if $w_x \in V^+$ and $\theta \leqslant \max(\sum_{x \in V} w_x x) < \theta + w_x$;
- $D(x) \leftarrow D(x) \setminus \{1\}$ if $w_x \in V^-$ and $\theta \leqslant \max(\sum_{x \in V} w_x x) < \theta - w_x$.

Example 3. Let us illustrate the propagation of the summation constraint. Given

$$D(x_1) = \{1\}, \qquad D(x_2) = D(x_3) = \{0, 1\},$$
$$2 * x_1 + 4 * x_2 + 8 * x_3 \geqslant 3;$$

we know that at least one of x_2 and x_3 must have value 1, but we cannot conclude that either one of these variables is certainly zero or one. The propagator does not change any domains. On the other hand, given

$$D(x_1) = \{1\}, \qquad D(x_2) = D(x_3) = \{0, 1\},$$
$$2 * x_1 + 4 * x_2 + 8 * x_3 \geqslant 7;$$

the propagator determines that the constraint can never be satisfied if x_3 is false, so $D(x_3) = \{1\}$.

2.2. Reified summation constraint

The second type of constraints we will use extensively is the reified summation constraint. Reified constraints are a common construct in constraint programming [52,54]. Essentially, a reified constraint binds the truth value of a constraint C' to a binary variable b:

$$b \leftrightarrow C'.$$

In principle, C' can be any boolean constraint. In this article, C will usually be a constraint on a sum. In this case we speak of a *reified summation constraint*:

$$b \leftrightarrow \sum_{x \in V} w_x x \geqslant \theta. \tag{6}$$

This constraint states that b is true if and only if the weighted sum of the variables in V is higher than θ. The most important constraint propagation that occurs for this constraint is the one that updates the domain of variable b. Essentially, the domain of this variable is updated as follows:

- $D(b) \leftarrow D(b) \setminus \{1\}$ if $\max(\sum_{x \in V} w_x x) < \theta$;
- $D(b) \leftarrow D(b) \setminus \{0\}$ if $\min(\sum_{x \in V} w_x x) \geqslant \theta$.

In addition, in some constraint programming systems, constraint propagators can also simplify constraints. In this case, if $D(b) = \{1\}$, the reified constraint can be simplified to the constraint $\sum_{x \in V} w_x x \geqslant \theta$; if $D(b) = \{0\}$, the simplified constraint becomes $\sum_{x \in V} w_x x < \theta$.

Many different constraint programming systems exist. They differ in the types of variables they support, the constraints they implement, the way backtracking is handled and the data structures that are used to store constraints and propagators. Furthermore, in some systems constraints are specified in logic (for instance, in the *constraint logic programming* system ECLiPSe [3]), while in others the declarative primitives are embedded in an imperative programming language. An example of the latter is Gecode [47], which we use in the experimental section of this article.

Tid	Itemset
T_1	{B}
T_2	{E}
T_3	{A,C}
T_4	{A,E}
T_5	{B,C}
T_6	{D,E}
T_7	{C,D,E}
T_8	{A,B,C}
T_9	{A,B,E}
T_{10}	{A,B,C,E}

Tid	A	B	C	D	E
1	0	1	0	0	0
2	0	0	0	0	1
3	1	0	1	0	0
4	1	0	0	0	1
5	0	1	1	0	0
6	0	0	0	1	1
7	0	0	1	1	1
8	1	1	1	0	0
9	1	1	0	0	1
10	1	1	1	0	1

Fig. 1. A small example of an itemset database, in multiset notation (left) and in binary matrix notation (right).

3. Frequent itemset mining

Now that we have introduced the key concepts underlying constraint programming (CP), we study various itemset mining problems within this framework. We start with *frequent* itemset mining in the present section, and then discuss *closed*, *discriminative* and *cost-based* itemset mining in the following sections. For every problem, we provide a formal definition, then introduce a constraint programming model that shows how the itemset mining problem can be formalized as a CP problem, and then compare the search strategy obtained by the constraint programming approach to that of existing itemset mining algorithms.

We start with the problem of frequent itemset mining and we formulate two CP models for this case. The difference between the initial model and the improved one is that the later uses the notion of *reified* constraints, which yields better propagation as shown by an analysis of the resulting search strategies.

3.1. Problem definition

The problem of frequent itemset mining was proposed in 1993 by Agrawal et al. [1]. Given is a database with a set of *transactions*.[2] Let $\mathcal{I} = \{1, \ldots, m\}$ be a set of items and $\mathcal{A} = \{1, \ldots, n\}$ be a set of transaction identifiers. An itemset database $\mathcal{D}$ is as a binary matrix of size $n \times m$ with $\mathcal{D}_{ti} \in \{0, 1\}$, or, equivalently, a multi-set of itemsets $I \subseteq \mathcal{I}$, such that

$$\mathcal{D}' = \big\{(t, I) \,\big|\, t \in \mathcal{A}, I \subseteq \mathcal{I},\ \forall i \in I\colon \mathcal{D}_{ti} = 1\big\}.$$

A small example of an itemset database is given in Fig. 1, where for convenience every item is represented as a letter.

There are many databases that can be converted into an itemset database. The traditional example is a supermarket database, in which each transaction corresponds to a customer and every item in the transaction to a product bought by the customer. Attribute–value tables can be converted into an itemset database as well. For categorical data, every attribute–value pair corresponds to an item and every row is converted into a transaction.

The coverage $\varphi_{\mathcal{D}}(I)$ of an itemset I consists of all transactions in which the itemset occurs:

$$\varphi_{\mathcal{D}}(I) = \{t \in \mathcal{A} \mid \forall i \in I\colon \mathcal{D}_{ti} = 1\}.$$

The *support* of an itemset I, which is denoted as $support_{\mathcal{D}}(I)$, is the size of the coverage:

$$support_{\mathcal{D}}(I) = \big|\varphi_{\mathcal{D}}(I)\big|.$$

In the example database we have $\varphi_{\mathcal{D}}(\{D, E\}) = \{T_6, T_7\}$ and $support_{\mathcal{D}}(\{D, E\}) = |\{T_6, T_7\}| = 2$.

Definition 2 *(Frequent itemset mining).* Given an itemset database $\mathcal{D}$ and a threshold θ, the frequent itemset mining problem consists of computing the set

$$\big\{I \,\big|\, I \subseteq \mathcal{I}, support_{\mathcal{D}}(I) \geqslant \theta\big\}.$$

The threshold θ is called the *minimum support* threshold. An itemset with $support_{\mathcal{D}}(I) \geqslant \theta$ is called a *frequent itemset*.

Note that we are interested in finding *all* itemsets satisfying the frequency constraint.

The subset relation between itemsets defines a partial order. This is illustrated in Fig. 2 for the example database of Fig. 1; the frequent itemsets are visualized in a *Hasse diagram*: a line is drawn between two itemsets I_1 and I_2 iff $I_1 \subset I_2$ and $|I_2| = |I_1| + 1$.

By changing the support threshold, an analyst can influence the number of patterns that is returned by the data mining system: the lower the support threshold, the larger the number of frequent patterns.

[2] Itemset mining was first applied in a supermarket setting; the terminology still reflects this.

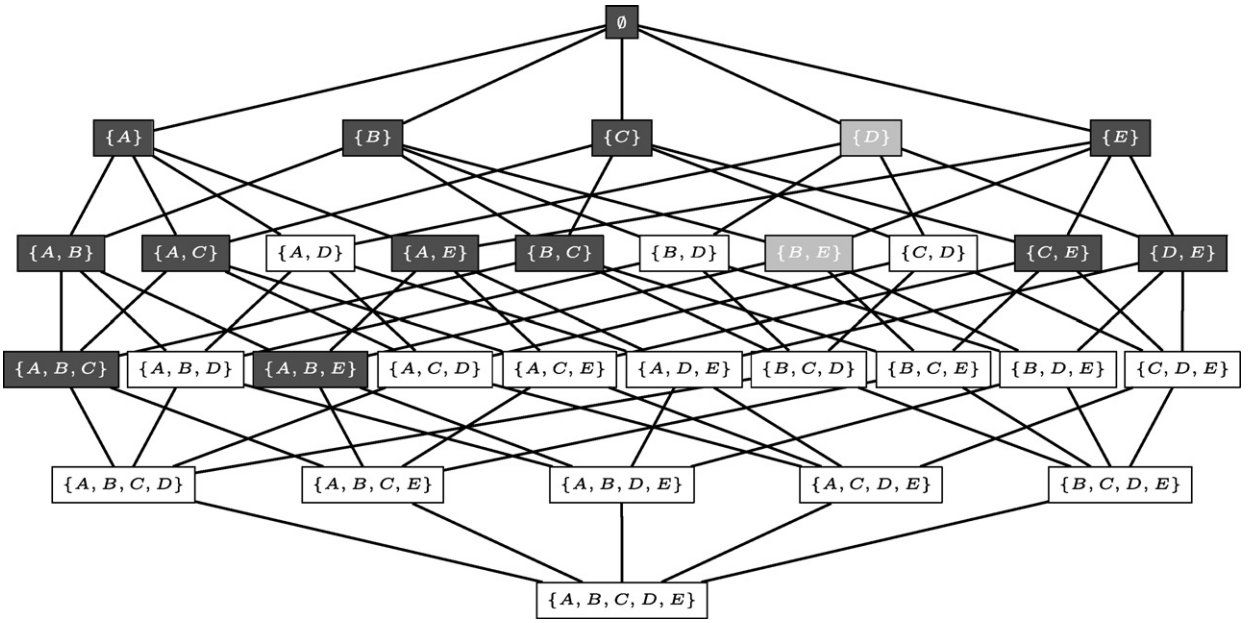


Fig. 2. A visualization of the search space for the database of Fig. 1; frequent itemsets for $\theta = 2$ are highlighted. Frequent closed itemsets are highlighted black; non-closed frequent itemsets are grey.

3.2. Initial constraint programming model

Our model of the frequent itemset mining problem in constraint programming is based on the observation that we can formalize the frequent itemset mining problem also as finding the set:

$$\big\{(I,T) \;\big|\; I \subseteq \mathcal{I},\ T \subseteq \mathcal{A},\ T = \varphi_{\mathcal{D}}(I),\ |T| \geqslant \theta\big\}.$$

Here we make the set of transactions $T = \varphi_{\mathcal{D}}(I)$ explicit. This yields the same solutions as the original problem because the set of transactions T is completely determined by the itemset I. We will refer to $T = \varphi_{\mathcal{D}}(I)$ as the *coverage constraint* while $|T| \geqslant \theta$ expresses a *support constraint*.

To model this formalization in CP, we need to represent the set of items I and the set of transactions T. In our model we use a boolean variable I_i for every individual item i; furthermore we use a boolean variable T_t for every transaction t. An itemset I is represented by setting $I_i = 1$ for all $i \in I$ and $I_i = 0$ for all $i \notin I$. The variables T_t represent the transactions that are covered by the itemset, i.e. $T = \varphi(I)$; $T_t = 1$ iff $t \in \varphi(I)$. One assignment of values to all I_i and T_t corresponds to one itemset and its corresponding transaction set.

We now show that the coverage constraint can be formulated as follows.

Property 1 *(Coverage constraint). Given a database $\mathcal{D}$, an itemset I and a transaction set T, then*

$$T = \varphi_{\mathcal{D}}(I) \Longleftrightarrow \left(\forall t \in \mathcal{A}\colon\ T_t = 1 \quad \leftrightarrow \quad \sum_{i \in \mathcal{I}} I_i(1 - \mathcal{D}_{ti}) = 0\right), \tag{7}$$

or equivalently,

$$T = \varphi_{\mathcal{D}}(I) \Longleftrightarrow \left(\forall t \in \mathcal{A}\colon\ T_t = 1 \quad \leftrightarrow \quad \bigwedge_{i \in \mathcal{I}} \mathcal{D}_{ti} = 1 \vee I_i = 0\right), \tag{8}$$

where $I_i, T_t \in \{0, 1\}$ and $I_i = 1$ iff $i \in I$ and $T_t = 1$ iff $t \in T$.

Proof. Essentially, the constraint states that for one transaction t, all items i should either be included in the transaction ($\mathcal{D}_{ti} = 1$) or not be included in the itemset ($I_i = 0$):

$$\begin{aligned} T = \varphi_{\mathcal{D}}(I) &= \{t \in \mathcal{A} \mid \forall i \in I\colon\ \mathcal{D}_{ti} = 1\} \\ &\Longleftrightarrow \quad \forall t \in \mathcal{A}\colon \quad T_t = 1 \leftrightarrow \forall i \in I\colon\ \mathcal{D}_{ti} = 1 \\ &\Longleftrightarrow \quad \forall t \in \mathcal{A}\colon \quad T_t = 1 \leftrightarrow \forall i \in I\colon\ 1 - \mathcal{D}_{ti} = 0 \end{aligned}$$

$$\iff \quad \forall t \in \mathcal{A}: \quad T_t = 1 \leftrightarrow \sum_{i \in \mathcal{I}} I_i(1 - \mathcal{D}_{ti}) = 0.$$

The representation as a clause in Eq. (8) follows from this. □

It is quite common in constraint programming to encounter different ways to model the same problem or even the same conceptual constraint, as above. How propagation is implemented for these constraints can change from solver to solver. For example, *watched literals* could be used for the clause constraint, leading to different runtime and memory characteristics compared to a setting where no watched literals are used. We defer the study of such characteristics to Section 7.

Under the coverage constraint, a transaction variable will only be true if the corresponding transaction covers the itemset. Counting the frequency of the itemset can now be achieved by counting the number of transactions for which $T_t = 1$.

Property 2 *(Frequency constraint). Given a database $\mathcal{D}$, a transaction set T and a threshold θ, then*

$$|T| \geqslant \theta \quad \iff \quad \sum_{t \in \mathcal{A}} T_t \geqslant \theta, \tag{9}$$

where $T_t \in \{0, 1\}$ and $T_t = 1$ iff $t \in T$.

We can now model the frequent itemset mining problem as a combination of the coverage constraint (7) and the frequency constraint (9). To illustrate this, we provide an example of our model in the Essence' language in Algorithm 3:

Algorithm 3 Fim_cp's frequent itemset mining model, in Essence'

```
1: given NrT, NrI : int
2: given TDB : matrix indexed by [int(1..NrT),int(1..NrI)] of int(0..1)
3: given Freq : int

4: find Items : matrix indexed by [int(1..NrI)] of bool
5: find Trans : matrix indexed by [int(1..NrT)] of bool

6: such that

7: $ Coverage Constraint (Eq. (7))
8: forall t: int(1..NrT).
9:    Trans[t] <=> ((sum i: int(1..NrI). (1-TDB[t,i])*Items[i]) <= 0),

10: $ Frequency Constraint (Eq. (9))
11: (sum t: int(1..NrT). Trans[t]) >= Freq.
```

Essence' is a solver-independent modeling language; it was developed to support intuitive modeling, abstracting away from the underlying solver technology [22].

We will now study how a constraint programming solver will search for the solutions given the above model. A first observation is that the set of transactions is completely determined by the itemset, so we need only to search over the item variables.

When an item variable is set ($D(I_i) = \{1\}$) by the search, only the constraints that contain this item will be activated. In other words, the frequency constraint will not be activated, but every coverage constraint that contains this item will be. A coverage constraint is a *reified summation constraint*, for which the propagator was explained in Section 2. In summary, when an item variable is set, the following propagation is possible for the coverage constraint:

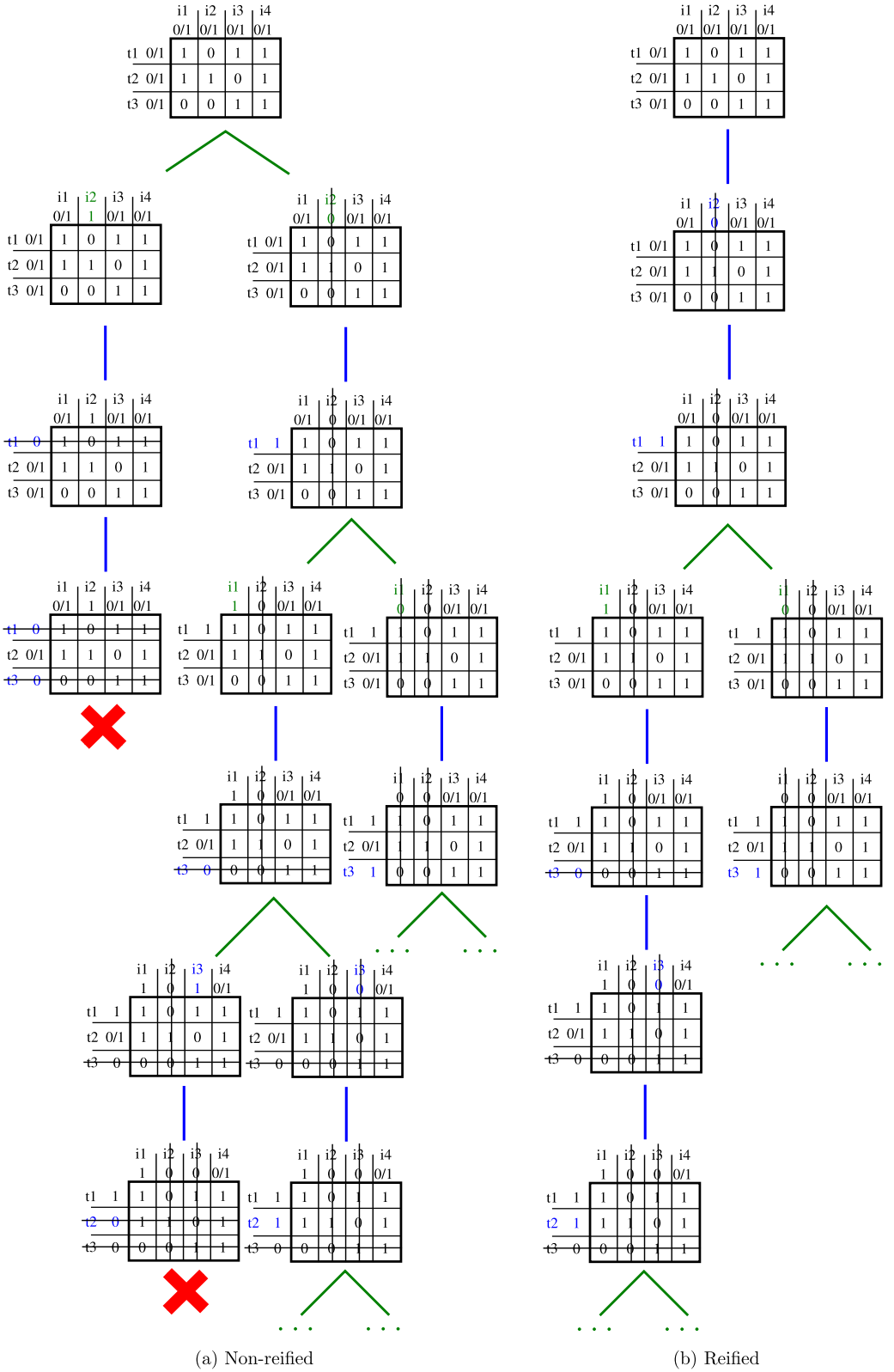
- if for some t: $\sum_{i \in \mathcal{I}}(1 - \mathcal{D}_{ti}) * I_i^{min} > 0$ then remove 1 from $D(T_t)$;
- if for some t: $\sum_{i \in \mathcal{I}}(1 - \mathcal{D}_{ti}) * I_i^{max} = 0$ then remove 0 from $D(T_t)$.

Once the domain of a variable T_t is changed, the support constraint will be activated. The support constraint is a *summation constraint*, which will check whether:

$$\sum_{t \in \mathcal{A}} T_t^{max} \geqslant \theta.$$

If this constraint fails, we do not need to branch further and we can backtrack.

Example 4. Fig. 3(a) shows part of a search tree for a small example with a minimum frequency threshold of 2. Essentially, the search first tries to add an item to an itemset and after backtracking it will only consider itemsets not including it. After a search step (indicated in green), the propagators are activated. The coverage propagators can set transactions to 0 or 1, while the frequency constraint can cause failure when the desired frequency can no longer be obtained (indicated by a red cross in the two left-most branches).



(a) Non-reified (b) Reified

Fig. 3. Search-propagation interaction of the non-reified frequent itemset model (left) and the reified frequent itemset model (right). A propagation step is colored in blue, a search step in green. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

Observe that in the example we branched on item 2 first. This follows the generic 'most constrained' variable order heuristic, which branches over the variable contained in most constraints first (remember that the coverage constraints are posted on items that have a 0 in the matrix). If item 1 would be branched over first, the search tree would be larger, as both branches would have to determine separately that $I_2 = 1$ does not result in a frequent itemset. An experimental investigation of different branching heuristics is done in Section 7.

3.3. Improved model

Inspired by observations in traditional itemset mining algorithms, we propose an alternative model that substantially reduces the size of the search tree by introducing fine-grained constraints. The main observation is that we can formulate the frequency constraint on each item individually:

Property 3 *(Reified frequency constraint). Given a database $\mathcal{D}$, an itemset $I \neq \emptyset$ and a transaction set T, such that $T = \varphi_{\mathcal{D}}(I)$, then*

$$|T| \geqslant \theta \quad \Longleftrightarrow \quad \forall i \in \mathcal{I} : I_i = 1 \rightarrow \sum_{t \in \mathcal{A}} T_t \mathcal{D}_{ti} \geqslant \theta, \tag{10}$$

where $I_i, T_t \in \{0, 1\}$ and $I_i = 1$ iff $i \in I$ and $T_t = 1$ iff $t \in T$.

Proof. We observe that we can rewrite $\varphi_{\mathcal{D}}(I)$ as follows:

$$\varphi_{\mathcal{D}}(I) = \{t \in \mathcal{A} \mid \forall i \in I\colon \mathcal{D}_{ti} = 1\} = \bigcap_{i \in I} \varphi_{\mathcal{D}}\big(\{i\}\big).$$

Using this observation, it follows that:

$$\begin{aligned}
|T| \geqslant \theta &\Longleftrightarrow \left| \bigcap_{j \in I} \varphi_{\mathcal{D}}\big(\{j\}\big) \right| \geqslant \theta \\
&\Longleftrightarrow \forall i \in I\colon \left| \varphi_{\mathcal{D}}\big(\{i\}\big) \cap \bigcap_{j \in I} \varphi_{\mathcal{D}}\big(\{j\}\big) \right| \geqslant \theta \\
&\Longleftrightarrow \forall i \in I\colon \left| \varphi_{\mathcal{D}}\big(\{i\}\big) \cap T \right| \geqslant \theta \\
&\Longleftrightarrow \forall i \in \mathcal{I}\colon I_i = 1 \rightarrow \sum_{t \in \mathcal{A}} T_t \mathcal{D}_{ti} \geqslant \theta. \qquad \square
\end{aligned}$$

The improved model consists of the coverage constraint (Eq. (7)) and the newly introduced reified frequency constraint (Eq. (10)). This model is equivalent to the original model, and also finds all frequent itemsets.

The reified frequency constraint is posted on every item separately, resulting in more fine-grained search-propagation interactions. Essentially, the reified frequency constraint performs a kind of look-ahead; for each item, a propagator will check whether that item is still frequent given the current itemset. If it is not, it will be removed from further consideration, as its inclusion would make the itemset infrequent. In summary, the main additional propagation allowed by the reified constraint is the following:

- if for some i: $\sum_{t \in \mathcal{A}} \mathcal{D}_{ti} * T_t^{max} < \theta$ then remove 1 from $D(I_i)$, i.e. $I_i = 0$.

Example 5. Fig. 3 shows the search trees of both the original non-reified model as well as the improved model using the reified frequency constraint.

In the original model (Fig. 3(a)), the search branches over $I_2 = 1$, after which the propagation detects that this makes the itemset infrequent and fails (left-most branch). In the reified model (Fig. 3(b)) the reified frequency propagator for I_2 detects that this item is infrequent. When evaluating the sum $(0 * T_1^{max} + 1 * T_2^{max} + 0 * T_3^{max})$, it is easy to see that the maximum is $1 < 2$, leading to $I_2 = 0$ (second level). The same situation occurs for I_3 near the bottom of the figure. This time, the propagator takes into account that at this point $T_3 = 0$ and hence $T_3^{max} = 0$.

The reified propagations avoid creating branches that can only fail. In fact, using the reified model, the search becomes failure-free: every branch will lead to a solution, namely a frequent itemset. This comes at the cost of a larger number of propagations. In Section 7 we experimentally investigate the difference in efficiency between the two formulations.

3.4. Comparison

Let us now study how the proposed CP-based approach compares to traditional itemset mining algorithms. In order to understand this relationship, let us first provide a short introduction to these traditional algorithms.

The most important property exploited in traditional algorithms is *anti-monotonicity*.

Definition 3 *(Anti-monotonic constraints).* Assume given two itemsets I_1 and I_2, and a predicate $p(I, \mathcal{D})$ expressing a constraint that itemset I should satisfy on database $\mathcal{D}$. Then the constraint is anti-monotonic iff $\forall I_1 \subseteq I_2 : p(I_2, \mathcal{D}) \Longrightarrow p(I_1, \mathcal{D})$.

Indeed, if an itemset I_2 is frequent, any itemset $I_1 \subseteq I_2$ is also frequent, as it must be included in at least the same transactions as I_2. This property allows one to develop search algorithms that do not need to consider all possible itemsets. Essentially, no itemset $I_2 \supset I_1$ needs to be considered any more once it has been found that I_1 is infrequent.

Starting a search from the empty itemset, there are many ways in which one could traverse the search space, the most important ones being breadth-first search and depth-first search. Initial algorithms for itemset mining were mostly breadth-first search algorithms, of which the Apriori algorithm is the main representative [2]. However, more recent algorithms mostly use depth-first search. Given that most CP systems also perform depth-first search, the similarities between CP and depth-first itemset mining algorithms are much larger than between CP and breadth-first mining algorithms. An outline of a general depth-first frequent itemset mining algorithm is given in Algorithm 4. The main observations are the following:

- if an item is infrequent in a database, we can remove the corresponding column from the database, as no itemset will contain this item and hence the column is redundant;
- once an item is added to an itemset, all transactions not containing this item become irrelevant for the search tree below this itemset; hence we can remove the corresponding row from the database.

The resulting database, which contains a smaller number of transactions having a smaller number of items, is often called a *projected database*. Hence, every time that we add an item to an itemset, we determine which items and transactions become irrelevant, and continue the search for the resulting database, which only contains frequent items and transactions covered by the current itemset. Important benefits are that the search procedure will never try to add items once they have been found to be infrequent; transactions no longer relevant can similarly be ignored.

Please note the following detail: in the projected database, we only include items which are strictly higher in order than the highest item currently in the itemset. The reason for this is that we wish to avoid that the same itemset is found multiple times; for instance, we wish to find itemset $\{1, 2\}$ only as a child of itemset $\{1\}$ and not also as a child of itemset $\{2\}$.

Algorithm 4 Depth-First-Search (Itemset I, Database $\mathcal{D}$)

```
1: F := {I}
2: determine a total order R on the items in D
3: for all items i occurring in D do
4:    create from D projected database D_i, containing:
5:    - only transactions in D that contain i
6:    - only items in D_i that are frequent and higher than i in chosen order R
7:    F := F ∪ Depth-First-Search(I ∪ {i}, D_i);
8: end for
9: return F
```

An important choice in this general algorithm is how the projected databases are stored. A very large number of strategies have been explored, among which tid-lists and FP-trees [30]. Tid-lists are most relevant here, as they compare best to strategies chosen in CP systems. Given an item i, its tid-list in a database $\mathcal{D}$ is $\varphi_{\mathcal{D}}(\{i\})$. We can store this list as a list of integers [56], a binary vector, or using variations of run-length encoding [49]. The projected database of a given itemset I is thus the set

$$\big\{\big(i, \varphi_{\mathcal{D}}\big(I \cup \{i\}\big)\big) \;\big|\; \big|\varphi_{\mathcal{D}}\big(I \cup \{i\}\big)\big| \geqslant \theta\big\}.$$

The interesting property of tid-lists is that they can easily be updated incrementally: if we wish to obtain a tid-list for item j in the projected database of itemset $\{i\}$, this can be obtained by computing $\varphi_{\mathcal{D}}(\{i\}) \cap \varphi_{\mathcal{D}}(\{j\})$, where $\mathcal{D}$ is the original database; for instance, in the case bit vectors are used this is a binary AND operation, which most CPUs evaluate efficiently. The most well-known algorithm using this approach is the Eclat algorithm [56].

An example of a depth-first search tree is given in Fig. 4, using the same database as in Fig. 3; we represent the projected database using tid-lists. The order of the items is assumed to be the usual order between integers. In the initial projected database, item 2 does not occur as it is not frequent. Each child of the root corresponds to an itemset with one item.

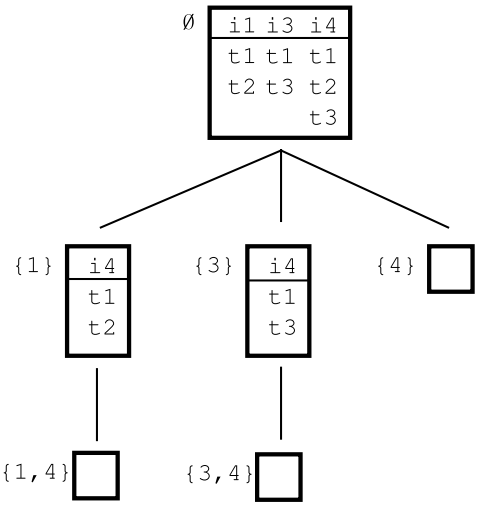


Fig. 4. The search tree for depth-first frequent itemset miners, for the same example as in Fig. 3, where the items are ordered by the natural order on integers. Each itemset has a corresponding projected database containing only frequent items higher than the items chosen so far. For instance, the projected database for itemset {4} is empty as items 1 and 3 are lower than 4; the database of {1} does not contain item i_3 as $\{1, 3\}$ is not frequent.

3.4.1. Comparison with search using the CP model

We now compare the above descriptions of itemset mining algorithms and constraint programming systems. Necessarily we need to restrict this discussion to a comparison of high-level principles; a detailed comparison of both approaches is not possible without studying the data structures of specific constraint programming systems in detail, which we consider to be out of the scope of this article; see [37] for a first attempt in that direction.

We first consider the differences in the search trees when using our CP model as compared to traditional mining algorithms. These differences can be understood by comparing the trees in Figs. 3 and 4. In depth-first itemset mining, each node in the search tree corresponds to an itemset. Search proceeds by adding items to it; nodes in the search tree can have an arbitrary number of children. In CP, each node in the search tree corresponds to a domain, which in our model represents a choice for possible values of items and transactions. Search proceeds by restricting the domain of a variable. The resulting search tree is always binary, as every item is represented by a boolean variable that can either be true or false (include or exclude the item).

We can identify the following relationship between nodes in the search tree of a CP system and nodes in the search tree of itemset miners. Denoting by $D(I_i)$ the domain of item variable I_i in the state of the CP system, we can map each state to an itemset as follows:

$$\left\{i \in \mathcal{I} \;\middle|\; D(I_i) = \{1\}\right\}.$$

Essentially, in CP some branches in the search tree correspond to an assignment $D(I_i) = \{0\}$ for an item i (i.e. the item i is removed from consideration). All nodes across a path of such branches are collapsed in one node of the search tree of the itemset miner, turning the binary tree in an n-ary tree.

Even though it might seem that this different perception of the search tree leads to a higher memory use in CP systems, this is not necessarily the case. If the search tree is traversed in the order indicated in Fig. 3(b), once we have assigned value $D(I_1) = \{0\}$ and generated the corresponding child node, we no longer need to store the original domain D with $D(I_1) = \{0, 1\}$. The reason is that there are no further children to generate for this original node in the search tree; if the search returns to this node, we can immediately backtrack further to its parent (if any). Hence, additional memory only needs to be consumed by branches corresponding to $D(I_i) = \{1\}$ assignments. This implies that in practice the efficiency depends on the implementation of the CP system; it does not depend on the theoretically different shape of the search tree.

In more detail these are the possible domains for the variables representing items during the search of the CP system:

- $D(I_i) = \{0, 1\}$: this represents an item that can still be added to the itemset, but that currently is not included; in traditional itemset mining algorithms, these are the items included in the projected database;
- $D(I_i) = \{0\}$: this represents an item that will not be added to the itemset. In the case of traditional itemset mining algorithms, these are items that are neither part of the projected database nor part of the current itemset;
- $D(I_i) = \{1\}$: this represents an item that will be part of all itemsets deeper down the search tree; in the case of traditional algorithms, this item is part of the itemset represented in the search tree node.

Similarly, we have the transaction variables:

- $D(T_t) = \{0, 1\}$: this represents a transaction that is covered by the current itemset (since 1 is still part of its domain), but may still be removed from the coverage later on; in traditional algorithms, this transaction is part of the projected database;
- $D(T_t) = \{0\}$: this represents a transaction that is not covered by the current itemset; in traditional algorithms, this transaction is not part of the projected database;
- $D(T_t) = \{1\}$: this represents a transaction that is covered by the current itemset and that will be covered by all itemsets deeper down the search tree, as the transaction contains all items that can still be added to the itemset; in traditional itemset mining algorithms, this transaction is part of the projected database.

A second difference is hence which information is available about transactions during the search. In our CP formalization, we distinguish transactions with domain $D(T_t) = \{0, 1\}$ and $D(T_t) = \{1\}$. Frequent itemset mining algorithms do not make this distinction. This difference allows one to determine when transactions are *unavoidable*: A transaction becomes unavoidable ($D(T_t) = \{1\}$) if all remaining items ($1 \in D(I_i)$) are included in it; the propagation to detect this occurs in branches where items are removed from consideration. Such branches are not present in the itemset mining algorithms; avoiding this propagation could be important in the development of new constraint programming systems.

Thirdly, to evaluate the constraints, CP systems store constraints or propagators during the search. Essentially, to every node in the search tree a *state* is associated that reflects active constraints, propagators and variables. Such a state corresponds to the concept of a projected database in itemset mining algorithms. The data structures for storing and maintaining propagators in CP systems and in itemset mining algorithms are however often very different. For example, in itemset mining efficient data representations such as tid-lists and fp-trees have been developed; CP systems use data structures for storing both propagators and constraints, which may be redundant in this problem setting. For instance, while in depth-first itemset mining, a popular approach is to store a tid-list in an integer array, CP systems may use both an array to store the indexes of variables in a constraint, and use an array to store a list of constraints watching a variable. Resolving these differences however requires a closer study of particular constraint programming systems, which is outside the scope of this paper.

Overall, this comparison shows that there are many high-level similarities between itemset mining and constraint programming systems, but that in many cases one can also expect lower-level differences. Our experiments will show that these low-level differences can have a significant practical impact, and hence that an interesting direction for future research is to bridge the gap between these systems.

4. Closed itemset mining

Even though the frequency constraint can be used to limit the number of patterns, the constraint is often not restrictive enough to find useful patterns. A high support threshold usually has as effect that only well-known itemsets are found; for a low threshold the number of patterns is often too large. To alleviate this problem, many additional types of constraints have been introduced. In this and the following sections, we will study how three further representative types of constraints can be formalized as constraint programming problems. Closed itemset mining aims at avoiding redundant itemsets, discriminative itemset mining wants to find itemsets that discriminate two classes of transactions, and cost-based constraints are representative for a fairly general class of constraints in the monotonicity framework.

4.1. Problem definition

Condensed representations aim at avoiding redundant itemsets, which are itemsets whose necessary presence in the full solution set may be derived from other itemsets found by the algorithm. The *closedness* constraint is a typical constraint that is used to find such a condensed representation [40]. We now introduce the closedness constraint more formally.

One way to interpret itemsets, is by seeing them as rectangles of ones in a binary matrix. For instance, in our example database of Fig. 1 on page 1956, for itemset $\{D\}$ we have corresponding transactions $\{T_6, T_7\}$. The itemset $\{D\}$ and the transaction set $\{T_6, T_7\}$ select a submatrix which can be seen as a rectangle in the matrix. Observe that due to the way that we calculate the set of transactions from the set of items, we cannot add a transaction to the set of transactions without including a zero element in the rectangle. However, this is not the case for the columns. In this case, we have $\varphi(\{D\}) = \varphi(\{D, E\}) = \{T_6, T_7\}$; we can add item E and still obtain a rectangle containing only ones. *Closed itemset mining* can be seen as the problem of finding maximal rectangles of ones in the matrix.

An essential property of 'maximal' rectangles is that if we consider its transactions, we can derive the corresponding set of items: the largest itemset shared by all transactions must define all columns included in the rectangle. This leads us to the following definition of closed itemset mining.

Definition 4 *(Frequent closed itemset mining).* Given a database $\mathcal{D}$, let $\psi_{\mathcal{D}}(T)$ be defined as

$$\psi_{\mathcal{D}}(T) = \{i \in \mathcal{I} \mid \forall t \in T\colon \mathcal{D}_{ti} = 1\}.$$

Given a threshold θ, the frequent closed itemsets are the itemsets in

$$\{I \mid I \subseteq \mathcal{I},\ support_{\mathcal{D}}(I) \geqslant \theta,\ \psi_{\mathcal{D}}(\varphi_{\mathcal{D}}(I)) = I\}.$$

Given an itemset I, the itemset $\psi_{\mathcal{D}}(\varphi_{\mathcal{D}}(I))$ is called the *closure* of I. Closed itemsets are those which equal their closure.

If an itemset is not equal to its closure, this means that we can add an item to the itemset without changing its support. Closed itemsets for our example database are highlighted in black in Fig. 2.

The idea behind closed itemsets has also been studied in other communities; closed itemset mining is in particular closely related to the problem of finding *formal concepts* in *formal contexts* [24]. Essentially, formal concepts can be thought of as closed itemsets that are found without applying a support threshold. In formal concept analysis, the operators φ and ψ are called Galois operators. These operators define a *Galois connection* between the partial orders for itemsets and transaction sets, respectively.

4.2. Constraint programming model

Compared to frequent itemset mining, the additional constraint that we need to express is the closedness constraint. We can deal with this constraint in a similar way as with the coverage constraint. Assuming that T represents the set of transactions covered by an itemset I, the constraint that we need to check is the following:

$$\psi_{\mathcal{D}}(T) = I, \tag{11}$$

as in this case $\psi_{\mathcal{D}}(\varphi_{\mathcal{D}}(I)) = I$. This leads to the following constraint in the CP model, which should be posted together with the constraints in Eqs. (7) and (10).

Property 4 *(Closure constraint). Given a database $\mathcal{D}$, an itemset I and a transaction set T, then*

$$I = \psi_{\mathcal{D}}(T) \iff \left(\forall i \in \mathcal{I}\colon\ I_i = 1 \leftrightarrow \sum_{t \in \mathcal{A}} T_t(1 - \mathcal{D}_{ti}) = 0\right), \tag{12}$$

where $I_i, T_t \in \{0, 1\}$ and $I_i = 1$ iff $i \in I$ and $T_t = 1$ iff $t \in T$.

The proof is similar to the proof for the coverage constraint.

4.3. Comparison

Several classes of algorithms have been proposed for closed itemset mining, each being either an extension of a breadth-first algorithm such as Apriori, or a depth-first algorithm, operating on tid-lists or fp-trees. We limit ourselves here to depth-first mining algorithms once more.

Initial algorithms for mining closed itemsets were based on a *repository* in which closed itemsets were stored. The search is performed by a depth-first frequent itemset miner which is modified as follows:

- when it is found that all transactions in a projected database contain the same item, this item is immediately added to the itemset as without it the itemset cannot be closed;
- for each itemset I_1 found in this way, it is checked in the repository whether an itemset $I_2 \supseteq I_1$ has been found earlier which has the same support; if not, the itemset is stored in the repository and the search continues; otherwise the closed supersets, starting with I_2, have already been found earlier as children of I_2 so this branch of the search tree is pruned.

The first modification only checks items that are in the projected database, which are by construction items $i > \max(I)$ that are higher in the lexicographic order. The repository is needed to check whether there is no superset with an additional item $i < \max(I)$; this is what the second modification does. With an appropriate search order only closed itemsets are stored [43].

This procedure works well if the number of closed sets is small and the database is large. When the number of closed itemsets is large storing itemsets and searching in them can be costly. The LCM algorithm addresses this problem [51]. In this algorithm also for the items $i < \max(I)$ it is checked in the data whether they should be part of the closure, even though the depth-first search procedure does not recurse on such items.

4.3.1. Constraint propagation

The additional constraint (12) for closed itemset mining is similar to the coverage constraint and hence its propagation is also similar. When all remaining transactions (i.e. those for which $1 \in D(T_t)$) contain a certain item, the propagator will:

- change the domain of the item i to $D(I_i) = \{1\}$ if $1 \in D(I_i)$;
- fail if $1 \notin D(I_i)$.

Tid	Itemset	Class
T_1	{B}	−
T_2	{E}	−
T_3	{A, C}	−
T_4	{A, E}	−
T_5	{B, C}	+
T_6	{D, E}	−
T_7	{C, D, E}	−
T_8	{A, B, C}	+
T_9	{A, B, E}	−
T_{10}	{A, B, C, E}	+

Fig. 5. A small example of a class-labeled itemset database, in multiset notation.

Hence, in this case we do not have failure-free search; if the closure constraint requires the inclusion of an item in the closure that cannot be included, the search will backtrack.

Overall this behavior is very similar to that of the LCM algorithm: essentially we are performing a backtracking search without storing solutions, in which items in the closure are immediately added and some branches are pruned as they fail to satisfy an order constraint. The main difference between LCM and the CP system is as in the previous section: other data structures are used and the search tree is differently organized.

5. Discriminative itemset mining

Itemset mining was initially motivated by the need to find rules, namely association rules. However, in the problem settings discussed till now, no rules were found; instead we only found conditions and no consequents. In this section we study the discovery of rules in a special type of transactional data, namely, data in which every transaction is labeled with a (binary) class label. The task is here to find itemsets that allow one to discriminate the transactions belonging to one class from those belonging to the other class. As it turns out, integrating this constraint efficiently in constraint programming requires the addition of a new primitive to the constraint programming system that we used till now. On the one hand this shows the limits of the declarative approach presented till now; on the other hand, our results demonstrate the feasibility of adding new data mining primitives as *global constraints*. Furthermore, as we will see, the application of the CP principles in the development of a new constraint propagator turns out to be crucial in improving the performance of existing mining systems.

5.1. Problem definition

To illustrate the problem of discriminative itemset mining, consider the database given in Fig. 5. We are interested in finding itemsets that discriminate transactions in different classes from one another. In the example database, for instance, the itemset $\{B, C\}$ almost only occurs in the positive examples, and hence can be thought to discriminate positive transactions from negative ones, leading to the rule $\{B, C\} \rightarrow +$.

Whereas we will refer to this mining problem here as the problem of *discriminative itemset mining* [35,13,21], it is also known under many other names, such as correlated itemset mining [48,39], interesting itemset mining [5], contrast set mining [4], emerging itemset mining [20] and subgroup discovery [55,32,28]. The problem is also highly related to that of rule learning in machine learning [23]. The key difference is that rule learning in machine learning usually uses heuristic techniques, while in itemset mining typically exhaustive techniques are used to find the global optimum.

Even though in the general case a target attribute may have multiple values, we will restrict ourselves to the case where the target attribute has two values: positive and negative. We refer to the part of the database for which the target attribute is positive as $\mathcal{D}^+$, and the part for which the target attribute is negative as $\mathcal{D}^-$. The set of transactions identifiers appearing in the corresponding parts is indicated by $\mathcal{A}^+$ and $\mathcal{A}^-$. We define the *stamp point* of an itemset I as $\sigma(I) = (|support_{\mathcal{D}^+}(I)|, |support_{\mathcal{D}^-}(I)|)$. Hence the stamp point of an itemset is a vector (p, n) where p is the frequency of this itemset in $\mathcal{D}^+$ and n is the frequency of this itemset in $\mathcal{D}^-$. Given these numbers, we can compute a *discrimination score* $f(p, n)$. For itemsets, the stamp point $\sigma(I) = (p, n)$ is used to calculate the score $f(\sigma(I))$, written $f(I)$ in short. Examples of discrimination measures f include χ^2, information gain, Gini index, Fisher score and others. For example χ^2 is a well-known measure of correlation in statistics:

$$\chi^2(p,n) = \frac{(p - \frac{(p+n)}{|\mathcal{D}|} \cdot |\mathcal{D}^+|)^2}{\frac{(p+n)}{|\mathcal{D}|} \cdot |\mathcal{D}^+|} + \frac{(n - \frac{(p+n)}{|\mathcal{D}|} \cdot |\mathcal{D}^-|)^2}{\frac{(p+n)}{|\mathcal{D}|} \cdot |\mathcal{D}^-|} + \frac{(|\mathcal{D}^+| - p - \frac{|\mathcal{D}|-(p+n)}{|\mathcal{D}|} \cdot |\mathcal{D}^+|)^2}{\frac{|\mathcal{D}|-(p+n)}{|\mathcal{D}|} \cdot |\mathcal{D}^+|} + \frac{(|\mathcal{D}^-| - n - \frac{|\mathcal{D}|-(p+n)}{|\mathcal{D}|} \cdot |\mathcal{D}^-|)^2}{\frac{|\mathcal{D}|-(p+n)}{|\mathcal{D}|} \cdot |\mathcal{D}^-|} \tag{13}$$

where it is assumed that $0/0 = 0$. An illustration of this measure is given in Fig. 6. The domain of stamp points $[0, |\mathcal{D}^+|] \times [0, |\mathcal{D}^-|]$ is often called *PN-space*.

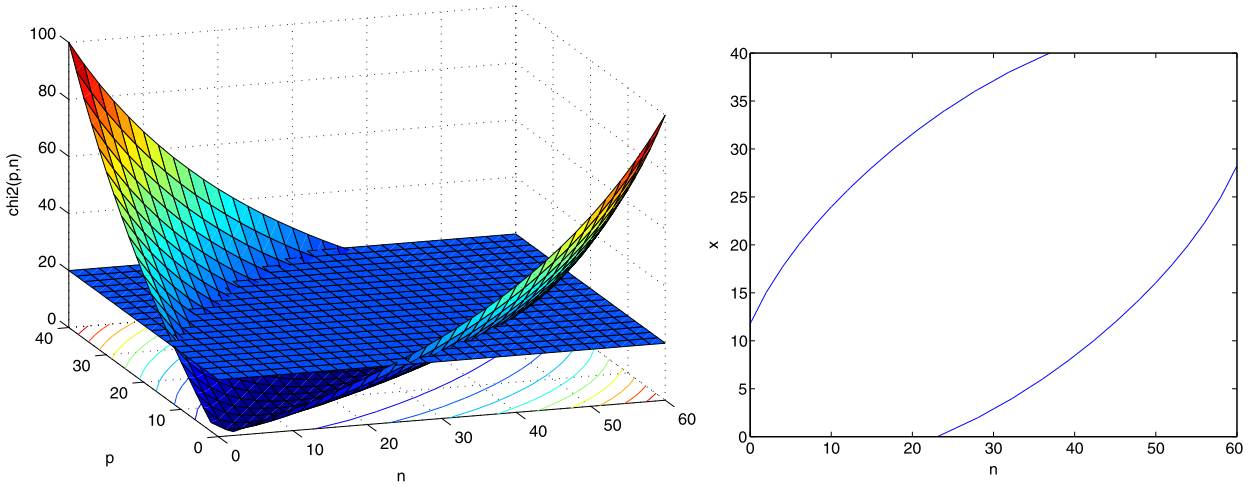


Fig. 6. Left: Plot of the χ^2 measure and a threshold at $\chi^2 = 20$. Right: isometric of the threshold in PN space.

Essentially, we are interested in finding itemsets which are as close as possible to the maxima in one of the opposite corners; the χ^2 measure scores higher the closer we are to these maxima.

A discrimination measure can be used in a constraint in several ways. We will limit ourselves to the following two cases.

Definition 5 *(Discriminative itemset mining).* Given a database $\mathcal{D}$, a discrimination measure f and a parameter θ, the discriminative itemset mining problem is that of finding all itemsets in

$$\{I \mid I \subseteq \mathcal{I},\ f(I) \geqslant \theta\}.$$

Definition 6 *(Top-k discriminative itemset mining).* Given a database $\mathcal{D}$, a discrimination measure f and a value k, the top-k discriminative itemset mining problem is the problem of finding the first k elements of the list $[I_1, I_2, \ldots, I_n]$ consisting of all itemsets $I \subseteq \mathcal{I}$ downward sorted by their $f(I)$ values.

In other words, the set of k patterns that score highest according to the discrimination measure. For $k = 1$ this corresponds to finding $\arg\max_{I \subseteq \mathcal{I}} f(I)$.

5.2. Constraint programming model

The discrimination constraint can be expressed in a straightforward way. In addition to the variables T_t and I_i we introduce two new variables p and n, calculated as follows:

$$p = \sum_{t \in \mathcal{A}^+} T_t, \qquad n = \sum_{t \in \mathcal{A}^-} T_t. \tag{14}$$

Remember that $\mathcal{A}^+$ and $\mathcal{A}^-$ represent the set of transaction identifiers in the positive database $\mathcal{D}^+$ and negative database $\mathcal{D}^-$ respectively. The discrimination constraint is now expressed as follows.

Property 5 *(Discrimination constraint). Given a database $\mathcal{D}$, a transaction set T, a discrimination measure f and a threshold θ, an itemset is discriminative iff*

$$f(p, n) \geqslant \theta,$$

where p and n are defined as described in Eq. (14)*.*

Such a constraint could be readily expressed in CP systems; essentially, a discrimination measure such as χ^2 is composed of a large number of mathematical operations on the variables p, n, $|\mathcal{A}^-|$ and $|\mathcal{A}^+|$. By carefully decomposing the measure into simple operations using intermediate variables, CP systems may be able to maintain bound consistency. This approach would however be cumbersome (for instance, in the case of the χ^2 function we would need to rewrite the formula to take care of the division by zero) and it is not guaranteed that rewriting its formula leads to an efficient computation strategy for all discrimination measures.

Hence, we propose a more robust approach here, which requires the addition of a new constraint in a CP system to enable the maintenance of tighter bounds for discrimination measures with 'nice' properties. Adding specialized *global*

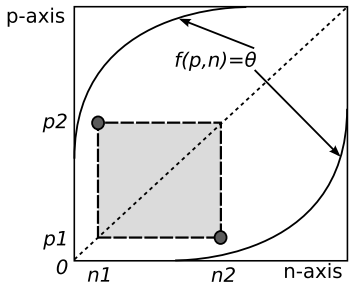


Fig. 7. Illustration of a rectangle of stamp points in PN space; within the rectangle $[p_1, p_2] \times [n_1, n_2]$, a ZDC measure reaches its highest value in one of the two highlighted stamp points.

constraints is common practice in CP [45] and hence well supported in systems such as Gecode. The main observation that we use in this case is that many discrimination measures, such as χ^2, are *zero on the diagonal and convex* (*ZDC*).

Definition 7. A scoring function f is *zero diagonal convex (ZDC)* if it has the following two properties:

- the function reaches its minimum in all stamp points on the diagonal in PN-space, i.e.,

$$\forall 0 \leqslant \alpha \leqslant 1\colon\ f\big(\alpha\big|\mathcal{A}^+\big|, \alpha\big|\mathcal{A}^-\big|\big) = 0;$$

- the function is *convex*, i.e., for every pair of stamp points $\sigma \neq \sigma'$ it holds that

$$\forall 0 \leqslant \alpha \leqslant 1\colon\ f\big(\alpha\sigma + (1-\alpha)\sigma'\big) \leqslant \alpha f(\sigma) + (1-\alpha) f\big(\sigma'\big).$$

Theorem 1. *Fisher score, information gain, Gini index, and χ^2 are ZDC measures.*

Definitions, as well as independent, alternative proofs of this theorem, can be found in [13,34,35]. The plot of χ^2 in Fig. 6 illustrates these two properties: the function is zero on the diagonal and convex.

For a ZDC measure the following can be proved.

Theorem 2 *(Maximum for ZDC measures). Let f be a ZDC measure and $0 \leqslant p_1 \leqslant p_2$ and $0 \leqslant n_1 \leqslant n_2$. Then*

$$\max_{(\sigma_1,\sigma_2)\in[p_1,p_2]\times[n_1,n_2]} f(\sigma_1,\sigma_2) = \max\big\{f(p_1,n_2), f(p_2,n_1)\big\}.$$

Proof. The proof is similar to that of [35]. First, we observe that the function is *convex*. Hence, we know that the maximum in a space $[p_1, p_2] \times [n_1, n_2]$ is reached in one of the points (p_1, n_1), (p_1, n_2), (p_2, n_1) and (p_2, n_2). Next, we need to show that we can ignore the corners (p_1, n_1) and (p_2, n_2). Observing that the minimum is reached on the diagonal, we can distinguish several cases.

If $n_1/|\mathcal{A}^-| < p_1/|\mathcal{A}^+|$, the point (p_1, n_1) is 'below' the diagonal. We know for the point $(\frac{|\mathcal{A}^+|}{|\mathcal{A}^-|}n_1, n_1)$ on the diagonal that $f(\frac{|\mathcal{A}^+|}{|\mathcal{A}^-|}n_1, n_1) = 0$. Due to the convexity we know then that $f(\frac{|\mathcal{A}^+|}{|\mathcal{A}^-|}n_1, n_1) = 0 \leqslant f(p_1, n_1) \leqslant f(p_2, n_1)$.

Similarly, we can show that if (p_1, n_1) is above the diagonal that $f(p_1, n_1) \leqslant f(p_1, n_2)$; that $f(p_2, n_2) \leqslant f(p_2, n_1)$ if (p_2, n_2) is below the diagonal; and that $f(p_2, n_2) \leqslant f(p_1, n_2)$ if (p_2, n_2) is above the diagonal. □

The bound states that to find the highest possible score in a rectangle of points, it suffices to check two corners of the rectangle. This is illustrated in Fig. 7, where a rectangle $[p_1, p_2] \times [n_1, n_2]$ is highlighted; the maximum on a ZDC measure is reached in one of the two corners (p_2, n_1) and (p_1, n_2). This property can be used to implement a propagator for a discrimination constraint.

Similar to the model for standard frequent itemset mining, we can improve the model by posting the discrimination constraint on each item individually, leading to the *reified discrimination constraint*:

$$\forall i \in \mathcal{I} : I_i = 1 \rightarrow f\left(\sum_{t\in\mathcal{A}^+} T_t\mathcal{D}_{ti}, \sum_{t\in\mathcal{A}^-} T_t\mathcal{D}_{ti}\right) \geqslant \theta. \tag{15}$$

Our CP model of discriminative itemset mining is a combination of this constraint and the coverage constraint in Eq. (7).

The propagator for the above constraint is obtained by applying Theorem 2 (see Algorithm 5).

To understand this propagator, consider the example in Fig. 7, where we have marked the curve $f(p, n) = \theta$ for a particular value of θ. Due to the convexity of the function f, stamp points for which $f(p, n) \geqslant \theta$ can be found in the

Algorithm 5 Conceptual propagator for $I_i = 1 \rightarrow f(\sum_{t\in\mathcal{A}^+} T_t\mathcal{D}_{ti}, \sum_{t\in\mathcal{A}^-} T_t\mathcal{D}_{ti}) \geqslant \theta$

```
1: if D(I_i) = {1} then
2:     post constraint f(Σ_{t∈A+} T_t D_ti, Σ_{t∈A-} T_t D_ti) ⩾ θ
3: else
4:     upper = max{f(Σ_{t∈A+} T_t^max D_ti, Σ_{t∈A-} T_t^min D_ti),
5:                 f(Σ_{t∈A+} T_t^min D_ti, Σ_{t∈A-} T_t^max D_ti)}
6:     if upper < θ then
7:         D(I_i) = D(I_i) \ {1}
8:     end if
9: end if
```

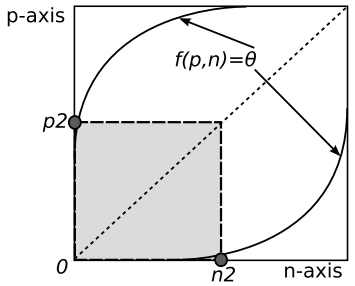


Fig. 8. Stamp points $(p_2, 0)$ and $(0, n_2)$ are upper bounds for the itemset I with $(p_2, n_2) = \sigma(I)$.

lower-right and upper-left corner. None of the stamp points in $(p, n) \in [p_1, p_2] \times [n_1, n_2]$, where $p_1 = \sum_{t\in\mathcal{A}^+} T_t^{min}\mathcal{D}_{ti}$, $p_2 = \sum_{t\in\mathcal{A}^+} T_t^{max}\mathcal{D}_{ti}$, $n_1 = \sum_{t\in\mathcal{A}^-} T_t^{min}\mathcal{D}_{ti}$ and $n_2 = \sum_{t\in\mathcal{A}^-} T_t^{max}\mathcal{D}_{ti}$, satisfies $f(p, n) \geqslant \theta$ in the figure; this can easily be checked by the propagator by determining that $f(p_1, n_2) < \theta$ and $f(p_2, n_1) < \theta$.

5.3. Comparison

Traditional discriminative itemset mining algorithms essentially proceed by updating frequent itemset mining algorithms such that a different anti-monotonic constraint is used during the search. This constraint is based on the derivation of an upper-bound on the discrimination measure [35].

Definition 8 *(Upper-bound).* Given a function $f(p, n)$, function $g(p, n)$ is an upper-bound for f iff $\forall p, n$: $f(p, n) \leqslant g(p, n)$.

In the case of itemsets it is said that the upper-bound is anti-monotonic, if the constraint $g(I) \geqslant \theta$ is anti-monotonic. The following upper-bound was presented by Morishita and Sese for ZDC measures [35].

Theorem 3 *(Upper-bound for ZDC measures). Let $f(p, n)$ be a ZDC measure, then $g(p, n) = \max(f(p, 0), f(0, n))$ is an upper-bound for $f(p, n)$ and $g(I) \geqslant \theta$ is an anti-monotonic constraint.*

Proof. The fact that this function is an upper-bound follows from Theorem 2, where we take $p_1 = n_1 = 0$, $p_2 = \sigma_1(I)$ and $n_2 = \sigma_2(I)$. The anti-monotonicity follows from the fact that $f(p, 0)$ and $f(0, n)$ are monotonically increasing functions in p and n, respectively. p and n represent the support of the itemset in the positive and negative databases $\mathcal{D}^+$ and $\mathcal{D}^-$ respectively, which are anti-monotonic as well. □

The bound is illustrated in Fig. 8. Given the threshold θ in this figure, the itemset I with stamp point $(p, n) = \sigma(I)$ will not be pruned, as at least one of $f(p, 0)$ or $f(0, n)$ has a discrimination score that exceeds the threshold θ.

This bound is used in an updated frequent itemset miner, of which the main differences are:

- we need to be able to compute the support in the two classes of data separately. This can be achieved both using *tid-list* and *fp-trees*;
- to prune items from the projected database, instead of a support constraint, a constraint on the upper-bound of the discrimination score is used: a subtree of the search tree is pruned iff $g(I) < \theta$, where θ is the threshold on the score (or the score of the kth best itemset found so far in top-k mining).

In case we do not wish to find all discriminative patterns with a score above θ, but instead the top-k patterns with the highest discriminative score, a branch-and-bound search strategy can be employed. In $top-1$ branch-and-bound search, the bound on the discrimination score $f(p, n)$ is increased as patterns with a higher score are found. For $top-k$ branch-and-bound search, the bound is set to that of the kth pattern.

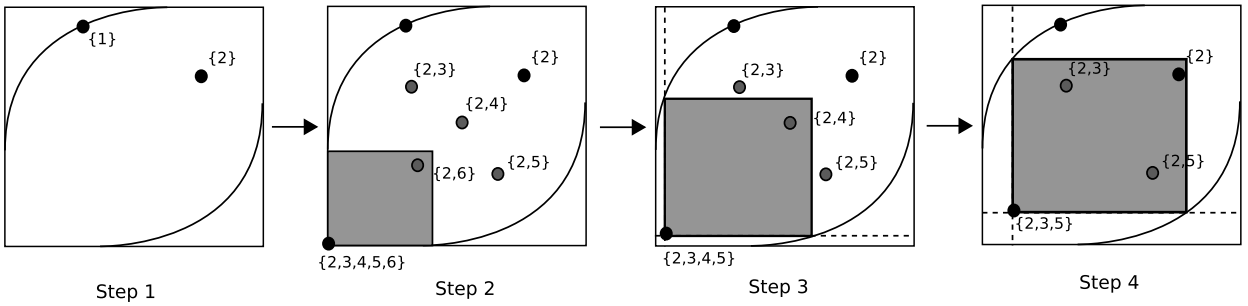


Fig. 9. Illustration of the possible propagation for discriminative itemset mining; this propagation loop was not yet studied in specialized itemset mining algorithms. In step 1 we have itemset {2}; we find out that itemset {2, 6} cannot reach the desired score and hence item 6 is excluded from consideration. As a result, some transactions may become unavoidable. Consequently, itemset {2, 4} may now be known not to reach the threshold and item {4} is excluded from consideration.

5.3.1. Constraint propagation

Intuitively, when we compare Figs. 7 and 8, we can see that the search would continue for the itemset in Fig. 8 because the maximum reachable score is measured in the points $(p_2, 0)$ and $(0, n_2)$, for which the score is above the threshold θ. On the other hand the search would stop in Fig. 7 because the maximum the itemset can reach is measured in (p_2, n_1) and (p_1, n_2), for which the score is below the threshold. The difference is that in Fig. 7 p_1 and n_1 are taken into account, which is the number of *unavoidable* transactions. As outlined on page 1963, unavoidable transactions are transactions for which $\min D(T_t) = 1$. So instead of having to use the upper-bound of Theorem 3, which does not take unavoidable transactions into account, we can use Theorem 2, which offers a much tighter bound, especially in the case of many unavoidable transactions.

Using the reified discrimination constraint leads to fine-grained interaction between search and propagation similar to the reified frequency constraint in Section 3.3; Excluding an item from the itemset by reducing its domain to $D(I_i) = \{0\}$, can lead to the following propagation loop:

1. Some transactions become unavoidable and are changed to $D(T_t) = \{1\}$.
2. $D(T_t)$ having changed, the reified discrimination constraints are checked; possibly a constraint detects that some item can no longer be included in the itemset and the item's domain is reduced to $D(I_i) = \{0\}$.
3. Return to step 1.

This propagation loop is illustrated in Fig. 9. It is absent in traditional discriminative itemset miners, which use the more simple bound $g(I)$. We will experimentally verify whether it is beneficial to perform the additional proposed propagation in Section 7.

6. Itemset mining with costs

In cases where the mining process still yields a very large set of patterns, additional constraints can reduce the number of patterns. Several papers have studied alternative constraints to the support constraint, which has lead to the concepts of monotonic, anti-monotonic and convertible anti-monotonic constraints. The prime example on which these concepts have been illustrated both in theory and in practice are constraints in which a cost, or weight, is associated with every item. In this section, we review these constraints and then show how they can be handled in the constraint programming approach.

6.1. Problem definition

Essentially, every item i now has an associated weight $c(i)$, often called the *cost*[3] of the item. Let us now define the total cost of an itemset as

$$c(I) = \sum_{i \in I} c(i).$$

Then we may be interested in finding itemsets for which we have a high total cost [42,11,9].

Definition 9 *(Frequent itemset mining with minimum total cost).* Given a database $\mathcal{D}$ and two parameters θ and γ, the frequent itemset mining problem under a minimum cost constraint is the problem of finding the itemsets in

$$\left\{ I \;\middle|\; I \subseteq \mathcal{I}, \mathit{support}_{\mathcal{D}}(I) \geqslant \theta,\ c(I) \geqslant \gamma \right\}.$$

[3] This terminology is again from the supermarket setting, where the cost of an item could be its price or profit.

Similarly, we can mine under maximum cost constraints and average cost constraints.

Definition 10 *(Frequent itemset mining with maximum total cost).* Given a database $\mathcal{D}$ and two parameters θ and γ, the frequent itemset mining problem under a maximum cost constraint is the problem of finding the itemsets in

$$\left\{ I \;\middle|\; I \subseteq \mathcal{I}, support_{\mathcal{D}}(I) \geqslant \theta,\ c(I) \leqslant \gamma \right\}.$$

Definition 11 *(Frequent itemset mining with minimum average cost).* Given a database $\mathcal{D}$ and two parameters θ and γ, the frequent itemset mining problem under a minimum average cost constraint is the problem of finding the itemsets in

$$\left\{ I \;\middle|\; I \subseteq \mathcal{I}, support_{\mathcal{D}}(I) \geqslant \theta,\ c(I)/|I| \geqslant \gamma \right\}.$$

Please note that a special case of cost-based itemset mining is achieved when $c(i) = 1$ for all i. These constraints are usually referred to as *size* constraints. A minimum size constraint is similar to a minimum support constraint: one is defined on the items, the other on the transactions.

6.2. Constraint programming model

In analogy to the support constraint, cost constraints can be expressed in two ways, non-reified and reified, and can be added to the usual support and coverage constraints in the CP model.

Property 6 *(Non-reified minimum and maximum total cost constraint). Given a database $\mathcal{D}$, an itemset I and a threshold γ, then*

$$c(I) \lesseqgtr \gamma \quad \Longleftrightarrow \quad \left(\sum_{i \in \mathcal{I}} I_i c(i) \lesseqgtr \gamma \right), \tag{16}$$

where $\lesseqgtr \in \{<, \leqslant, \geqslant, >\}$, $I_i \in \{0, 1\}$ and $I_i = 1$ iff $i \in I$.

Property 7 *(Reified minimum and maximum total cost constraint). Given a database $\mathcal{D}$, an itemset I and a threshold γ, if $support_{\mathcal{D}}(I) \geqslant 1$ then*

$$c(I) \lesseqgtr \gamma \quad \Longleftrightarrow \quad \left(T_t = 1 \rightarrow \sum_{i \in \mathcal{I}} I_i D_{ti} c(i) \lesseqgtr \gamma \right), \tag{17}$$

where $\lesseqgtr \in \{<, \leqslant, \geqslant, >\}$, $I_i \in \{0, 1\}$, $I_i = 1$ iff $i \in I$ and $T_t = 1$ iff $t \in T$.

Proof. This follows from the assumption that also the coverage constraint should hold; hence if $T_t = 1$ we know that for all i with $I_i = 1$ we must have $D_{ti} = 1$. Because $support_{\mathcal{D}}(I) \geqslant 1$, we know that there is at least one transaction for which $T_t = 1$. □

Average cost constraints can be expressed by allowing for negative coefficients.

Property 8 *(Non-reified minimum and maximum average cost constraint). Given a database $\mathcal{D}$, an itemset I and a transaction set T, then*

$$c(I)/|I| \lesseqgtr \gamma \quad \Longleftrightarrow \quad \left(\sum_{i \in \mathcal{I}} I_i \big(c(i) - \gamma\big) \lesseqgtr 0 \right), \tag{18}$$

where $\lesseqgtr \in \{<, \leqslant, =, \neq, \geqslant, >\}$, $I_i \in \{0, 1\}$ and $I_i = 1$ iff $i \in I$.

Proof. This follows from the following observation:

$$c(I)/|I| \lesseqgtr \gamma \quad \Leftrightarrow \quad c(I) \lesseqgtr \gamma |I| \quad \Leftrightarrow \quad \big(c(I) - \gamma |I|\big) \lesseqgtr 0. \qquad \square$$

The reified average cost constraints are obtained in a similar way as the reified total cost constraints.

6.3. Comparison

All specialized algorithms for mining under cost constraints exploit that these constraints have properties similar to anti-monotonicity.

Definition 12 *(Monotonic constraints).* Assume given two itemsets I_1 and I_2, and a predicate $p(I, \mathcal{D})$ expressing a constraint. Then the constraint is monotonic iff $\forall I_1 \subseteq I_2 : p(I_1, \mathcal{D}) \Longrightarrow p(I_2, \mathcal{D})$.

Examples are maximum support and minimum cost constraints. Different approaches have been proposed for dealing with monotonic constraints in the literature. We will discuss these approaches separately, at the same time pointing out the relation to our models in CP.

6.3.1. Minimum total cost constraint: simple approach

The *simplest* depth-first algorithms developed in the data mining community for dealing with monotonic constraints are based on the observation that supersets of itemsets satisfying the constraint also satisfy the constraint. Hence, during the depth-first search procedure, we do not need to check the monotonic constraint for children of itemsets satisfying the monotonic constraint [41]. To emulate this behavior in CP, we would only check the satisfiability of the monotone constraint, and refrain from possibly propagating over variables. This would result in branches of the search tree being cut when they can no longer satisfy the constraint; the constraint would be disabled once it can no longer be violated.

6.3.2. Minimum total cost constraint: DualMiner/non-reified

More advanced is the specialized *DualMiner* algorithm [11]. DualMiner associates a triplet $(I_{in}, I_{check}, I_{out})$ with every node in the depth-first search tree of an itemset miner. Element I_{in} represents the itemset to which the node in the search tree corresponds; I_{check} and I_{out} provide additional information about the search node. Items in I_{check} are currently not included in the itemset, but may be added in itemsets deeper down the search tree; these items are part of the projected database. For items in I_{out} it is clear that they can no longer be added to any itemset deeper down the search tree. Adding an item of I_{check} to I_{in} leads to a branch in the search tree. An iterative procedure is applied to determine a final triplet $(I_{in}, I_{check}, I_{out})$ for the new search node, and to determine whether the recursion should continue:

- it is checked whether the set I_{in} satisfies all anti-monotonic constraints. If not, stop;
- it is checked which individual items in I_{check} can be added to I_{in} and satisfy the anti-monotonic constraints. Only those that do satisfy the constraints are kept in I_{check}, others are moved to I_{out};
- it is checked whether the set $I_{in} \cup I_{check}$ satisfies the monotonic constraints. If not, stop. Every item $i \in I_{check}$ for which itemset $(I_{in} \cup I_{check}) \backslash \{i\}$ does not satisfy the monotonic constraints, is added to I_{in}. (For instance, if the total cost is too low without a certain item, we have to include this item in the itemset.) Finally, the procedure is iterated again to determine whether I_{in} still satisfies the anti-monotonic constraints.

If the loop reaches a fixed point and items are still left in I_{check} the search continues, unless it also appears that $I_{in} \cup I_{check}$ satisfies the anti-monotonic constraints and I_{in} satisfies the monotonic constraints; in this case the sets $I_{check} \subseteq I \subseteq I_{in} \cup I_{check}$ could immediately be listed.

A similar search procedure is obtained when the cost constraints are formulated in a non-reified way in the CP system. As pointed out earlier, a non-reified minimum or maximum cost constraint takes the following form:

$$\sum_i I_i c(I) \lesseqgtr \gamma .$$

Propagation for this constraint is of the following kind:

- if according to current domains the constraint can only be satisfied when the CP system includes (minimum cost constraint) or excludes (maximum cost constraint) an item, then the system does so; this corresponds to moving an item to the I_{in} or I_{out} set in DualMiner;
- if according to current domains the constraint can no longer be satisfied, backtrack;
- if according to current domains the constraint will always be satisfied the constraint is removed from the constraint store.

Hence, the overall search strategy for the non-reified constraint is similar to that of DualMiner. There are also some differences. DualMiner does not aim at finding the transaction set for every itemset. If it finds that I_{in} satisfies the monotonic constraint and $I_{in} \cup I_{check}$ satisfies the anti-monotonic constraint, it does not continue searching, and outputs the corresponding range of itemsets explicitly or implicitly. The CP system will continue enumerating all itemsets in the range in order to find the corresponding transaction sets explicitly.

6.3.3. Minimum total cost constraint: FP-Bonsai/reified

In the FP-Bonsai algorithm [8], the idea of iteration till a fixed point is reached was extended to monotonic constraints. The main observation on which this algorithm is based is that a transaction which does not satisfy the minimum cost constraints, will never contain an itemset that satisfies the minimum cost constraint. Hence, we can remove such transactions from consideration. This may reduce the support of items in the projected database and result in the removal of more items from the database. The reduction in size of some transactions may trigger a new step of propagation.

If we consider the reified minimum cost constraint,

$$T_t = 1 \rightarrow \sum_i I_i c(I) \mathcal{D}_{ti} \geqslant \gamma,$$

we can observe a similar propagation. Propagation essentially removes a transaction from consideration when the constraint can no longer be satisfied on the transaction. The removal of this transaction may affect the support of some items, requiring the propagators for the support constraints to be re-evaluated.

Note that the reified constraint is less useful for a maximum total cost constraint, $c(I) \leqslant \gamma$. Essentially, for every transaction only items already included in the itemset can be considered. If the sum of these items is to large, the transaction is removed from consideration. This would happen for all transactions separately, leading to a failed branch. Overall, the propagation is expensive to evaluate (as it needs to be done for each transaction) and not as effective as the non-reified propagator which can also prune items from consideration instead of only failing.

Thus the reified and non-reified form are complementary to each other. We can obtain both types of propagation by posting constraints of both types in a CP system.

6.3.4. Minimum and maximum average cost constraints.

Average cost constraints are neither monotonic nor anti-monotonic. Still, they have a property that is related to that of monotonic and anti-monotonic constraints.

Definition 13 *(Convertible anti-monotonic constraints).* Assume given two itemsets I_1 and I_2, a predicate $p(I, \mathcal{D})$ expressing a constraint, and an order $<$ between items. Then the constraint is convertible anti-monotonic for this order iff $\forall I_1 \subseteq I_2, \min(I_2 \setminus I_1) \geqslant \max(I_1) : p(I_2, \mathcal{D}) \Longrightarrow p(I_1, \mathcal{D})$.

For example, if the items are ordered according to increasing cost $c(I)$, when adding items that are more expensive than the current items, the average cost can only increase. For a decreasing order in item cost, the minimum average cost constraint is convertible anti-monotonic. Different orderings will not result in anti-monotonic behavior, i.e. if after adding expensive items an item with a low cost would be added, the average cost would go down. Note that a conjunction of maximum and minimum cost constraints is hence not convertible anti-monotonic, as we would need opposing orders for each of the two constraints.

Essentially our formalization in CP of average cost constraints is very similar to that of total cost constraints, the main difference being that negative costs are allowed. Consequently, depending on the constraint (minimum or maximum) and the weight (positive or negative) either the maximum value in the domain or the minimum value in the domain is used in the propagator. In the non-reified form, we obtain propagation towards the items; in the reified form towards the transactions.

This search strategy is fundamentally different from the search strategy used in specialized mining systems, in which the property is used that one average cost constraint is convertible anti-monotonic. Whereas in specialized systems the combination of multiple convertible constraints poses problems, in the CP-based approach this combination is straightforward.

6.3.5. Conclusions

Interestingly, when comparing models in CP and algorithms proposed in the data mining community, we can see that there are many similarities between these approaches. The different approaches can be distinguished based on whether they represent a reified or an non-reified constraint. The main advantage of the constraint programming approach is that these approaches can also easily be combined. This advantage is also evident when combining convertible constraints; specialized algorithms can only optimize on one such constraint at the same time.

7. Experiments

In previous sections we concentrated on the conceptual differences between traditional algorithms for itemset mining and constraint programming; we showed that itemset mining can be modeled in constraint programming. In the present section, we first consider several choices that have to be made when implementing itemset mining in a constraint programming framework and evaluate their influence on the performance of the mining process. More specifically, we answer the following two questions about such choices:

Table 1
Properties of the datasets used.

Dataset	# transactions	# items	Density	10% freq	# solutions (10% freq)
1. Soybean	630	59	0.25	63	12 754
2. Splice-1	3190	290	0.21	319	1963
3. Anneal	812	41	0.43	81	1 891 712
4. Mushroom	8124	119	0.19	812	574 514

QA What is the difference in performance between using reified versus non-reified constraints of itemset mining?
QB What is the effect of the different variable orderings on the performance of itemset mining?

Further (more technical and system dependent) choices made in our implementation are explained in Appendix A for completeness and reproducibility. All used implementations are also available on our website: http://dtai.cs.kuleuven.be/CP4IM/.

We use the best approach resulting from the above experiments to experimentally compare our constraint programming framework CP4IM to state-of-the-art itemset mining systems. More specifically, our comparative experiments focus on answering the following questions:

Q1 What is the difference in performance of CP4IM for *frequent* itemset mining and traditional algorithms?
Q2 What is the difference in performance of CP4IM for *closed* itemset mining and traditional algorithms?
Q3 Is the additional propagation in CP4IM for *discriminative* itemset mining beneficial? If so, how much?
Q4 Is the alternative approach for dealing with *convertible* constraints in CP4IM beneficial? If so, how much?

We ran experiments on PCs with Intel Core 2 Duo E6600 processors and 4 GB of RAM, running Ubuntu Linux. The experiments are conducted using the Gecode constraint programming system [25]. Gecode[4] is an open source constraint programming system which is representative for the current state-of-the-art of efficient constraint programming.

The starting point for our experiments was Gecode version 2.2.0. In the course of our experiments we tried several formulations and implemented alternative propagators, explained in detail in Appendix A, some of which are now included in Gecode by default.

7.1. Data sets

In our experiments we used data from the UCI Machine Learning repository.[5] To deal with missing values we preprocessed each dataset in the same way as [16]: we first eliminated all attributes having more than 10% of missing values and then removed all examples (transactions) for which the remaining attributes still had missing values. Numerical attributes were binarized by using unsupervised discretization with 4 equal-frequency bins; each item for an attribute corresponds to a threshold between two bins. These preprocessed datasets can be downloaded from our website.[6] The datasets and their basic properties can be found in Table 1. The density is the relative amount of ones in the matrix. In many itemset problems, a higher density indicates that the dataset is more difficult to mine.

7.2. Alternative itemset miners

We used the following state-of-the-art specialized algorithms, for which implementations are freely available, as the basis for our comparative evaluation:

LCM LCM [51] is a specialized frequent closed itemset mining algorithm based on tid-lists;
Eclat Eclat [56] is a specialized depth-first frequent itemset mining based on tid-lists;
Patternist Patternist [9] is a specialized breadth-first algorithm for mining under monotonic and anti-monotonic constraints;
DDPMine DDPMine [13] is a specialized depth-first closed discriminative itemset mining algorithm based on fp-trees and a repository of closed itemsets; it uses a less tight bound than the bound summarized in Section 5 [38].

Note that in our experiments we are not using all algorithms discussed in previous sections. The reason is that we preferred to use algorithms for which comparable implementations by the original authors were freely available (i.e. executable on the same Linux machine).

Table 2 provides an overview of the different tasks that these data mining systems support. The LCM and Eclat algorithms have been upgraded by their original authors to support respectively frequent itemset mining and closed itemset mining too. Patternist is a constraint-based mining algorithm which has been carefully designed to make maximal use of monotone

[4] http://www.gecode.org/.
[5] http://archive.ics.uci.edu/ml/.
[6] http://dtai.cs.kuleuven.be/CP4IM/.

Table 2
Comparing the mining tasks that different miners support.

	CP4IM	LCM	Eclat	Patternist	DDPMine
Frequent itemsets	X	X*	X	X	
Closed itemsets	X	X	X*		
Correlated itemsets	X				X
Monotone constraints	X			X	
Convertible constraints	X			X	
Combinations of the above	X			X**	

X* – Not originally designed for this task.
X** – Combinations of the constraints it supports, except multiple convertible constraints.

Table 3
Comparing the reified versus non-reified formulation of the frequency constraint on 2 datasets for different frequencies. The reified formulation never has failed branches.

Dataset	Freq.	Non-reified frequency				Reified frequency			Faster
		Mem.	Props.	Failures	Time (s)	Mem.	Props.	Time (s)	
Anneal	5%	2694	235 462	170	46.3	2950	236 382	44.7	1.04
Anneal	10%	2438	221 054	248	19.3	2501	224 555	18.9	1.02
Anneal	15%	2309	200 442	298	8.4	2373	203 759	8.3	1.01
Mushroom	5%	17 862	7 737 116	10 383	269.5	20 486	5 980 478	239.8	1.12
Mushroom	10%	17 862	4 184 940	3376	74.2	20 229	2 853 248	43.4	1.71
Mushroom	15%	17 862	2 680 479	1909	40.8	19 973	1 589 289	10.5	3.89

and convertible constraints during the search. Our CP4IM system is the only system that supports all of these constraints as well as combinations of these constraints. Furthermore, thanks to the use of a declarative constraint programming system it can easily be extended with further types of constraints. This is what we regard as the major of advantage of the constraint programming methodology. It is also interesting to contrast this approach with that taken by the alternative, more procedural systems, which were typically designed to cope with a single constraint family and were later upgraded to deal with other ones too. This upgrade usually involves changing the algorithm dramatically and hard-coding the new constraint in it. In contrast, in CP one might need to add a new propagator (as we have done for the discrimination constraint), but the corresponding constraint can freely be used and combined with any other current and future constraint in the system. This is essentially the difference between a declarative and a procedural approach.

On the other hand, generality and flexibility also may have a price in terms of performance. Therefore, we do not expect CP4IM to perform well on each task, but we would hope its performance is competitive when averaging over a number of tasks.

7.3. QA: non-reified vs reified

In Section 3.3 we argued that using reified frequency constraints for the standard frequent itemset mining problem can lead to more propagation. Table 3 shows a comparison between running the CP model with non-reified and reified frequency constraints. Two datasets were used, each with three different frequency thresholds. For the reasonably small *anneal* dataset, it can be noted that the non-reified version needs a bit less memory and propagation, but at the cost of some failed branches in the search tree. This leads to a small increase in run time. For the bigger *mushroom* dataset however, the difference is larger. For higher minimum frequency thresholds the reified pruning becomes stronger while for the non-reified formulation the cost of a failed branch increases, leading to a larger difference in runtime. In general we have observed that using the reified frequency constraints often leads to better performance, especially for larger datasets.

7.4. QB: variable ordering

In constraint programming it is known that the order in which the variables are searched over can have a large impact on the size of the search tree, and hence the efficiency of the search. This has received a lot less attention in the itemset mining community, except in algorithms like fp-growth where a good ordering is needed to keep the fp-tree size down.

We consider three possible variable ordering strategies, for the standard frequent itemset mining problem:

arbitrary: the input order of variables is used;
minimum degree: the variable occurring in the smallest number of propagators;
maximum degree: the variable occurring in the largest number of propagators.

The comparison of the three variable orderings can be found in Table 4. The experiments show that choosing the variable with maximum degree leads to a large reduction in the number of propagations and runtime. The maximum degree heuris-

Table 4
Comparison of peak memory, number of propagations and time (in seconds) using different variable ordering heuristics on the 4 datasets listed in Table 1.

	Arbitrary			Minimum degree			Maximum degree		
	Mem.	Props.	Time	Mem.	Props.	Time	Mem.	Props.	Time
1	1860	799 791	0.5	1540	3 861 055	3.1	1860	137 666	0.2
2	124 431	3 188 139	49	126 927	3 772 591	60	122 511	2 602 069	41
3	2116	121 495 848	73	2116	472 448 674	374	1796	577 719	18
4	31 236	344 377 153	365	30 148	997 905 725	1504	26 244	6 232 932	48

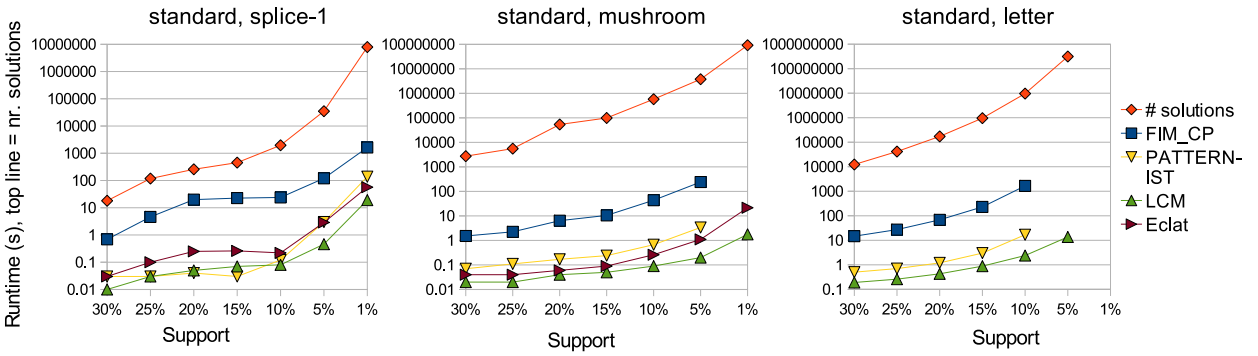


Fig. 10. Standard itemset mining on different datasets. (For interpretation of the references to color in this figure, the reader is referred to the web version of this article.)

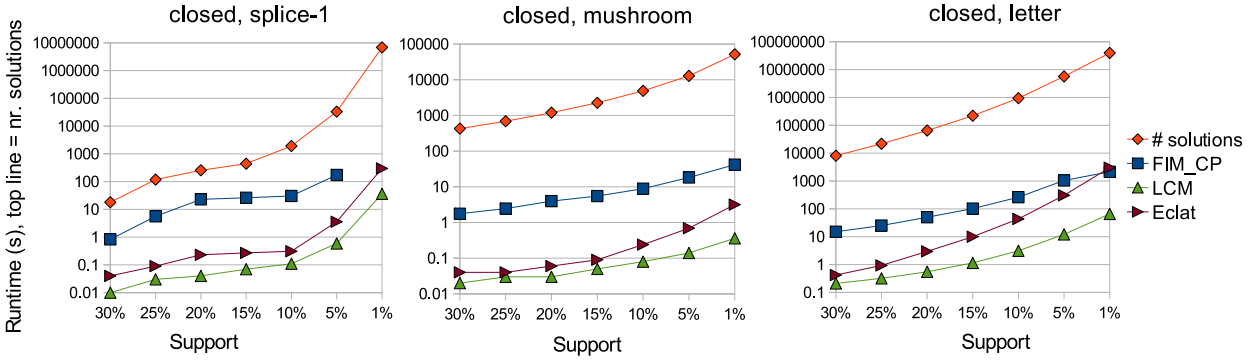


Fig. 11. Closed itemset mining on different datasets.

tic corresponds to choosing the item with the lowest frequency, as this item occurs in the coverage constraint (Eq. (7)) of most transactions. In other words, the most efficient variable ordering strategy is a fail-first strategy that explores the most unlikely branches of the search tree first. Such a conclusion is not surprising in the constraint programming community.

7.5. Q1: frequent itemset mining

A comparison of specialized frequent itemset miners and CP4IM is provided in Fig. 10 for a representative number of datasets. In this figure we show run times for different support thresholds as it was previously found that the differences between systems can highly depend on this constraint [27].

The run time of all systems is correlated to the number of patterns found (the red line). Our CP4IM model implemented in Gecode, indicated by FIM_CP, is significantly slower than the other depth-first miners, but shows similar behavior. This indicates that indeed its search strategy is similar, but the use of alternative data structures and other overhead in the constraint programming system introduces a lot of overhead for standard frequent itemset mining. It is remarkable that CP4IM compares well to the breadth-first Patternist system, which does not use the concept of projected databases as pervasively as other systems; the compact representations developed in the specialized itemset miners for projected databases indeed explain the performance difference.

Table 5
Statistics of UCI datasets, and runtimes, in seconds, of two CP models and other systems.

Name	Dense	Trans	Item	CP4IM(4)	CP4IM(2)	ddpmine [14]	lcm [51]
Anneal	0.45	812	93	0.22	24.09	22.46	7.92
Australian-cr	0.41	653	125	0.30	0.63	3.40	1.22
Breast-wisc	0.50	683	120	0.28	13.66	96.75	27.49
Diabetes	0.50	768	112	2.45	128.04	–	697.12
German-cr	0.34	1000	112	2.39	66.79	–	30.84
Heart-clevel	0.47	296	95	0.19	2.15	9.49	2.87
Hypothyroid	0.49	3247	88	0.71	10.91	–	>
Ionosphere	0.50	351	445	1.44	>	–	>
kr-vs-kp	0.49	3196	73	0.92	46.20	125.60	25.62
Letter	0.50	20000	224	52.66	>	–	>
Mushroom	0.18	8124	119	14.11	13.48	0.09	0.03
Pendigits	0.50	7494	216	3.68	>	–	>
Primary-tum	0.48	336	31	0.03	0.13	0.26	0.08
Segment	0.50	2310	235	1.45	>	–	>
Soybean	0.32	630	50	0.05	0.07	0.05	0.02
Splice-1	0.21	3190	287	30.41	31.11	1.86	0.02
Vehicle	0.50	846	252	0.85	>	–	>
Yeast	0.49	1484	89	5.67	781.63	–	185.28

7.6. Q2: closed itemset mining

In Fig. 11 the runtime of all mining algorithms is shown for the problem of closed itemset mining. Again, run time is correlated with the number of patterns found. The difference between CP4IM and the other miners is smaller in this experiment. We argued in the previous section that the CP system behaves similar to the LCM system. Indeed, our experiments on both the mushroom and letter data set show that this is the case; in one case even outperforming the Eclat system, which as not originally developed for closed itemset mining.

It should be noted that on sparse data, such as mushroom, the difference in performance between Gecode and specialized systems is larger than on dense data, such as the letter data. This can be explained by the inefficient representation of sparse data in Gecode; on dense data, as compared to Eclat, this inefficient representation is compensated by more effective propagation.

7.7. Q3: discriminative closed itemset mining

In this experiment we compare several approaches for finding the most discriminative itemset, given labeled data. Results are shown in Table 5. As in this setting we do not have to determine a threshold parameter, we perform experiments on a larger number of datasets. The missing values of the datasets were preprocessed in the same way as in previous experiments. However, the numerical attributes were binarized using unsupervised discretization with 7 binary split points (8 equal-frequency bins). This enforces a language bias on the patterns that is closer to that of rule learning and subgroup discovery systems [28]. In case of a non-binary class label, the largest class was labeled positive and the others negative. The properties of the datasets are summarized in Table 5; note the higher density of the datasets than in the previous experiments, resulting from the discretization procedure.

We report two types of experiments with CP4IM: using the propagator introduced in Section 5.2 (CP4IM(4)) and using a propagator that mimics the propagation occurring in the specialized discriminative itemset miner introduced in [35] (CP4IM(2)). Furthermore, we also apply the LCM algorithm; in [38] it was shown that for well-chosen support thresholds, the resulting set of frequent itemsets is guaranteed to contain all itemsets exceeding a correlation threshold. We use the correlation threshold of the best pattern (found using our algorithm) to compute a support threshold according to this method and run LCM with this support threshold. Note that by providing LCM knowledge about the best pattern to be found, the comparison is unfair to the advantage of LCM.

For experiments marked by ">" in our table no solution was found within 900 seconds. In experiments marked by "–" the repository of closed itemsets runs out of memory. The experiment shows that CP4IM(4) consistently outperforms existing data mining systems, where in most cases this increased performance can be attributed to the improved propagation that was revealed in CP4IM(4).

It can be noted that on one dataset, the mushroom dataset, the new propagator takes slightly more time. Our hypothesis is that this is related to the low density of the data, for which 4-bound pruning can be less effective when there is no structure in the data which would lead to unavoidable transactions. To test this hypothesis, we performed additional experiments in which we gradually sparsified two dense datasets, given in Fig. 12. The sparsification was performed by randomly removing items uniformly from the transaction database, until a predefined sparsity threshold was reached. Averaging runtimes over 10 different samples for each setting, we ran our CP4IM system using three different propagators: CP4IM(4) and CP4IM(2), as explained above, and CP4IM(1) which uses the simple frequency based propagator used in [14].

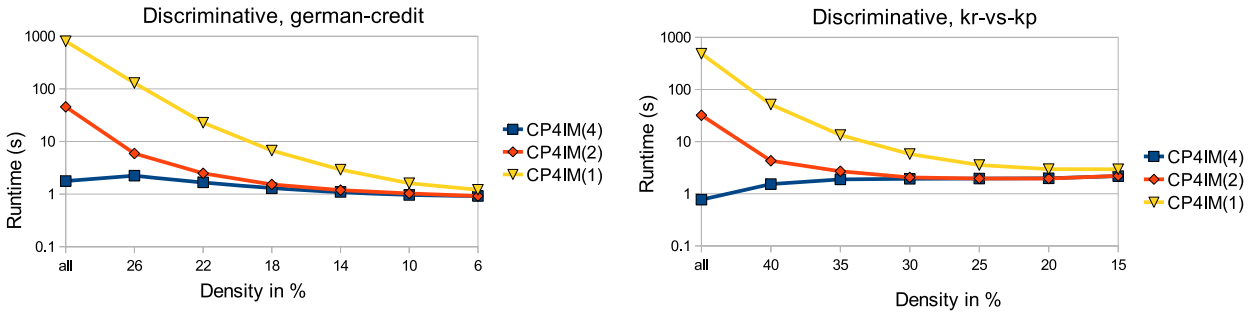


Fig. 12. Correlated itemset mining when sparsifying the data.

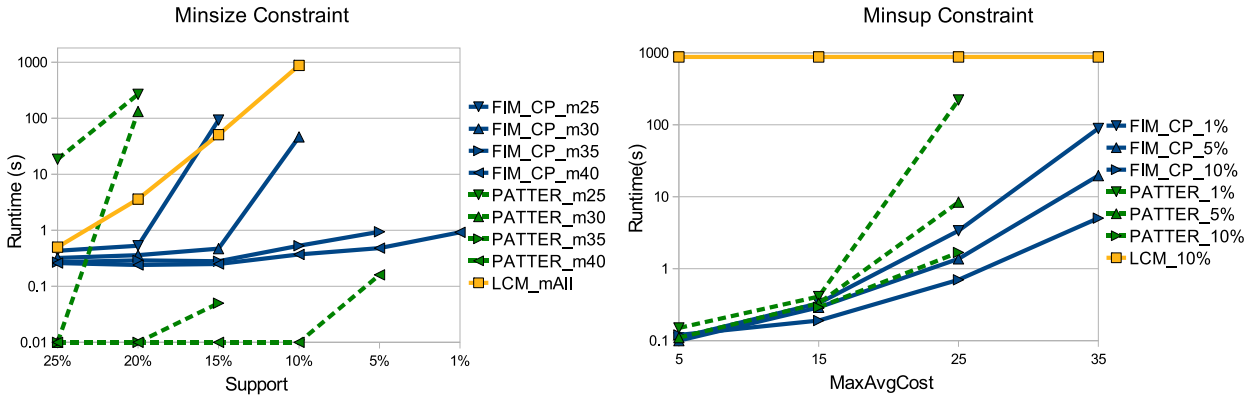


Fig. 13. Runtimes of itemset miners on Segment data under constraints. In the left figure, the suffix _m25, _m30, etc. of the name indicates the minimum size threshold. In the right figure, the suffix _1%, etc. represent the minimum support threshold.

The experiments show that when the density is decreased, and hence the sampling removes structure from the data, the advantage of the more advanced pruning method over the more simple ones disappears. However, within the CP framework the 4-bound method is often better and at worse equivalent to the 2- and 1-bound pruning.

7.8. Q4: cost-based itemset mining

In this experiment we determine how CP4IM compares with other systems when additional cost constraints are employed. Results for two settings are given in Fig. 13, where our system is indicated by FIM_CP. In the first experiment we employed a (monotonic) minimum size constraint in addition to a minimum frequency constraint; in the second a (convertible) maximum average cost constraint. The results are positive: even though for small minimum size constraints the brute force mining algorithms, such as LCM, outperform CP4IM, CP4IM does search very effectively when this constraint selects a small number of very large itemsets (30 items or more); in extreme cases CP4IM finishes within seconds while other algorithms do not finish within our cut-off time of 30 minutes. Patternist, being a breadth-first algorithm, was unable to finish some of these experiments due to memory problems. This indicates that CP4IM is a competitive system when the constraints require the discovery of a small number of very large itemsets. The results for convertible constraints are particularly promising, as we did not optimize the item order in any of our experiments, as is usually done when dealing with convertible constraints.

8. Conclusions

We started this paper by raising the question as to whether constraint programming can be used for solving itemset mining problems in a declarative way. Our results show that the answer to this question is indeed positive and that the use of constraint programming offers several advantages as well as new insights. Perhaps the more important advantage is that constraint programming systems are general purpose systems supporting many different types of constraints. In this regard, we showed that it is possible to incorporate many well-known constraints, such as cost constraints, closedness or discriminative measures as defined above, as well as their combinations in the constraint programming system. The advantage of the resulting declarative approach to data mining is that it is easy to extend or change in order to accommodate new constraints, and that all constraints can automatically be combined with one another. Furthermore, a detailed analysis of the solution strategy of constraint programming systems showed that there are many similarities between these systems and specialized itemset mining systems. Therefore, the constraint programming system arguably generalizes these systems,

not only from a theoretical perspective but also from a practical one. This was confirmed in our experiments: for problems such as frequent and closed itemset mining, for which fast implementation contests were organized, these specialized systems usually outperform CP4IM; however the runtime behavior of our constraint programming approach is similar to that of the specialized systems. The potential of the CP approach from a performance perspective was demonstrated on the problem of discriminative itemset mining. We showed that by rigorously using the principles of constraint programming, more effective propagation is obtained than in alternative state-of-the-art data mining algorithms. This confirms that it is also useful in an itemset mining context to take propagation as a guiding principle. In this regard, it might be interesting to investigate the use of alternative search strategies that have been developed in the constraint programming community [53,44].

Continuing this research, we are currently studying the application of our approach to problems arising in bioinformatics. For instance, itemset mining has commonly been applied to the analysis of microarray data; our hope is that constraint programming may offer a more general and more flexible approach to analyze such data. Whereas the above work is still restricted to the discovery of patterns in binary data, the use of constraint programming in other pattern mining related problems is also a promising direction of future research. A problem closely related to pattern mining is that of pattern set mining [19], where one does not only impose constraints on individual patterns, but also on the overall set of patterns constituting a solution [29]. Constraints that can be imposed include, for instance, the requirement that patterns do not overlap too much, or that they cover the complete set of transactions together. Another related problem is that of finding patterns in continuous data. This requirement is in particular relevant to deal with problems in bioinformatics. Likewise, there are many approaches to mining structured data, such as sequences, trees and graphs. It is an interesting open question as to whether it is possible to represent such problems using constraint programming too. One of the challenges here is that such structured data can no longer be represented using a fixed number of features or variables.

In addition to pattern mining, other areas of machine learning and data mining may also profit from a closer study of constraint programming techniques. One such area is statistical machine learning, where problems are typically formulated using mathematical programming. Recently some results in the use of other types of solvers have already been obtained for certain probabilistic models [12,15]. In these approaches, however, Integer Linear Programming (ILP) or satisfiability (SAT) solvers were used. CP solvers address a more general class of problems than ILP and SAT solvers, but this generality sometimes comes at a computational cost. Current developments in CP that aim at combining ILP and SAT with CP may also help in addressing these machine learning problems.

Other topics of interest are constraint-based clustering and constraint-based classifier induction. In constraint-based clustering the challenge is to cluster examples when additional knowledge is available about these examples, for instance, prohibiting certain examples from being clustered together (so-called cannot-link constraints). Similarly, in constraint-based classifier induction, one may wish to find a decision tree that satisfies size and cost-constraints. A first study on the application of CP on this problem was recently performed by Bessiere, Hebrard, and O'Sullivan [7]. In data mining, the relationship between itemset mining and constraint-based decision tree learning was studied [36]. It is an open question as to whether this relation can also be exploited in a constraint programming setting.

Whereas the previous cases study how data mining could profit from constraint programming, the opposite direction is also a topic of interest: how can constraint programming systems be extended using techniques from data mining? For example, in constraint programming systems the data is typically spread over the constraints, and possibly multiple times in different ways. In contrast, in data mining the data is typically centrally accessed, allowing the use of different matrix representations.

To summarize, we believe that the further integration of machine learning, data mining and constraint programming may be beneficial for all these areas.

Acknowledgements

This work was supported by a Postdoc and a project grant from the Research Foundation—Flanders, project "Principles of Patternset Mining" as well as a grant from the Institute for the Promotion and Innovation through Science and Technology in Flanders (IWT-Vlaanderen).

Appendix A. Improved solving, continued

In Section 7 we empirically studied the effect of non-reified versus reified formulations and of different variable ordering heuristics. In this appendix we include some additional findings, as we have experienced that making the right low-level decisions is necessary to be competitive with the highly optimized itemset mining implementations. We start by studying the differences between using boolean variables and integers with a domain of $\{0, 1\}$. We continue by studying two implementation alternatives for an essential constraint shared by all models: the coverage constraint. We end with a comparison of different value ordering heuristics; to explain and improve the results we have to provide some additional details about the Gecode system.

Apart from Gecode specific remarks, which are applicable only to solvers that do copying and cloning, the results presented in this appendix are also valid and applicable to other solvers. In fact, parts of the work studied here are now by default in the aforementioned Gecode system.

Table A.6
Comparison in propagations, peak memory and time (in seconds) of using boolean variables and their respective constraints versus using integer variables and constraints.

Dataset	Original boolean			Integers			Gain
	Peak mem.	# props	Time	Peak mem.	# props	Time	
1. Soybean	2820	5 909 592	1.4	2436	1 839 932	0.8	1.7
2. Splice-1	147 280	23 708 807	129	142 032	9 072 323	57	2.3
3. Anneal	3140	1 121 904 924	279	2564	273 248 618	136	2.1
4. Mushroom	45 636	2 989 128 466	1447	39 940	862 480 847	508	2.9

Table A.7
Comparison in propagations, peak memory and time (in seconds) of the channeled integer formulation of the base model and the boolean formulation with the dedicated propagator.

Dataset	Integers			Dedicated boolean			Gain
	Peak mem.	# props	Time	Peak mem.	# props	Time	
1. Soybean	2436	1 839 932	0.8	1796	1 058 238	0.5	1.6
2. Splice-1	142 032	9 072 323	57	123 279	6 098 820	68	0.8
3. Anneal	2564	273 248 618	136	2500	121 495 848	74	1.8
4. Mushroom	39 940	862 480 847	508	30 852	520 928 196	387	1.3

A.1. Booleans vs integers

Finite domain integer solvers can choose to represent a boolean as an integer with a domain of $\{0, 1\}$, or to implement a specific boolean variable.

An essential constraint in our models is the reified summation constraint. Such a sum can be expressed both on boolean and integer variables, but the reification variable always has a boolean domain. Using boolean variables should be equally or more efficient than integer variables, especially since in our model, the integer variables need to be 'channeled' to boolean variables for use in the reification part. However, in our experiments (Table A.6) the model using booleans was slower than the one using integers. The reason is that a given reified summation constraint on booleans B_i and boolean variable C,

$$\sum_i B_i \geqslant v \leftrightarrow C$$

was decomposed into two constraints: $S = \sum_i B_i$ and $S \geqslant v \leftrightarrow C$, where S is an additional integer variable; separate propagators were used for both constraints. For integers on the other hand, a single propagator was available. Our experiments show that decomposing a reified sum constraints over booleans into a sum of booleans and reifying the integer variable is not beneficial.

We implemented a dedicated propagator for a reified sum of boolean variables constraint, which includes an optimization inspired by SAT solvers [26]. A propagator is said to *watch* the variables on which it depends. A propagator is activated when the domain of one of its watched variables changes. To improve efficiency, the number of watched variables should not be larger than necessary. Assume we have a sum $\sum_{i=1}^{n} B_i \geqslant v \leftrightarrow C$, where all B_i and C are boolean variables, then it is sufficient to watch $\max(v, n - v + 1)$ (arbitrary) variables B_i not fixed yet: the propagator can not succeed (v variables true) or fail ($n - v + 1$ variables false) without assigning at least one of the watched variables. In Table A.7 we compare the formulation of the basic frequent itemset mining problem using integers and channeling, to using boolean variables and the new dedicated propagator. The peak amount of memory needed when using only booleans is naturally lower. The amount of propagations is also decreased significantly, leading to lower runtimes for all but one dataset. Hence it is overall recommended to use boolean variables with dedicated propagators.

A.2. Coverage constraint: propagators versus advisers

When a variable's domain changes, the propagators that watch this variable can be called with different amounts of information. To adopt the terminology of the Gecode system, we differentiate between classic 'propagators' and 'advisers':

- propagators: when the domain of at least one variable changes, the entire propagator is activated and re-evaluated;
- advisers: when the domain of a variable changes, an adviser is activated and informed of the new domain of this variable. When the adviser detects that propagation can happen, it will activate the propagator.

Both techniques have their advantages and disadvantages: classic propagators are conceptually simpler but need to iterate over all its variables when activated; advisers are more fine-grained but require more bookkeeping. We implemented the coverage constraint using both techniques, and compare them in Table A.8. Using advisers requires more memory but reduces the overall amount of propagations, the runtimes also decrease.

Table A.8
Comparison in propagations, peak memory and time (in seconds) of formulating the coverage constraints using: the new reified sum constraint, the advisor version of the new reified sum constraint and the clause constraint.

	Boolean sum			Boolean sum, advisers			Clause, advisers		
	Mem.	# props	Time	Mem.	# props	Time	Mem.	# props	Time
1	1796	1 058 238	0.5	2500	799 791	0.5	1860	799 791	0.5
2	123 279	6 098 820	68	237 071	3 188 139	54	124 431	3 188 139	49
3	2500	121 495 848	74	2500	121 495 848	73	2116	121 495 848	73
4	30 852	520 928 196	387	47 172	344 377 153	372	31 236	344 377 153	365

Table A.9
Comparison of peak memory, propagations and time (in seconds) using the minimum or maximum value ordering heuristics on the frequent itemset mining problem.

Dataset	Minimum value			Maximum value		
	Mem.	Props.	Time	Mem.	Props.	Time
1. Soybean	1860	137 666	0.2	899	217 802	0.3
2. Splice-1	122 511	2 602 069	41	16 328	4 961 137	109
3. Anneal	1796	577 719	18	1412	726 308	18
4. Mushroom	26 244	6 232 932	48	20 229	9 989 882	63

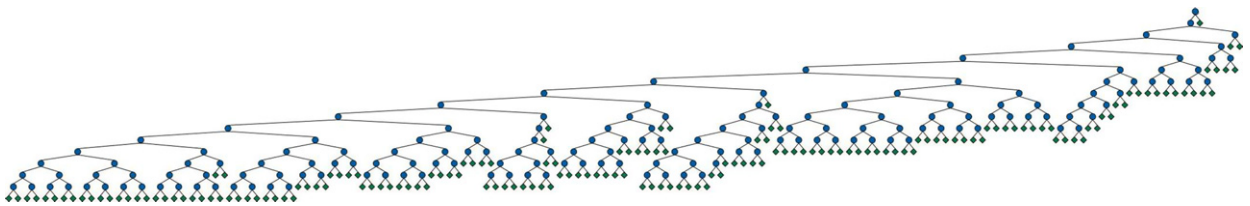


Fig. A.14. Search tree for the first 35 variables of the mushroom dataset. Every blue circle is a branchpoint over an item, every green diamond is a solution. A branch to the left assigned 0 to the item of that branchpoint, a branch to the right assigned 1. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

A.3. Coverage constraint: clauses vs sums.

As we pointed out in Property 1, Eqs. (7) and (8), on page 1957, the coverage constraint can be expressed in two equivalent ways: using reified sums or using reified clauses. Both options are evaluated in Table A.8. Overall, we find that the formulation using clauses performs best.

A.4. Value ordering

For boolean decision variables, two value ordering heuristics are meaningful: selecting the minimum value (0) or selecting the maximum value (1) first. A comparison can be found in Table A.9, where the maximum degree variable ordering is used.

The results are surprising: using the maximum value heuristic leads to more propagation and longer run times. This is counter-intuitive: the search tree is equally large in both cases and because the complete tree is searched, the total amount of propagation should be identical too. The explanation can be found in how the Gecode system stores intermediate states during the search. Gecode uses a technique called copying and recomputation [46]. In this technique, some nodes, but not necessarily all nodes, in the depth-first search tree are copied and stored. To backtrack, one retrieves the latest copied node and recomputes the propagations using the assignments leading to the desired node. This can save memory consumption and runtime for large problems [46]. The amount of copying/recomputation is set by the *copy distance* parameter. In Gecode, the default is 8, meaning that a new copy is made every 8 nodes.

When we consider a search tree using the minimum value first heuristic for our models (see Fig. A.14), we see that all variables are set to zero first, creating one long branch. The copied nodes in this branch are reused throughout the rest of the search. When using the maximum value heuristic, more propagation is possible and shorter branches are explored first. Consequently, less nodes are copied, and a lot of recomputation needs to be done in each of the short branches. In our experiments this results in increased overhead.

Table A.10 compares two values of the copy distance parameter, and how this influences the value ordering heuristic. With a distance of 0, every node in the search tree is copied. This results in a smaller amount of propagation compared to a distance of 8, independent of the value ordering heuristic used. Interestingly, the amount of runtime is also decreased compared to a larger copy distance. Using the maximum value first heuristic is about as fast as the minimum value heuristic,

Table A.10
Comparison of peak memory, propagations and time (in seconds) using the minimum or maximum value ordering heuristics. The copy distance is either the default (c–d 8) or zero (c–d 0).

	Minimum, c–d 8			Minimum, c–d 0			Maximum, c–d 0		
	Mem.	Props.	Time	Mem.	Props.	Time	Mem.	Props.	Time
1	1860	137 666	0.2	7626	64 823	0.2	1796	64 823	0.2
2	122 511	2 602 069	41	911 863	1 822 064	24	16 328	1 822 064	23
3	1796	577 719	18	4231	224 555	19	2501	224 555	19
4	26 244	6 232 932	48	148 173	2 853 248	43	20 229	2 853 248	43

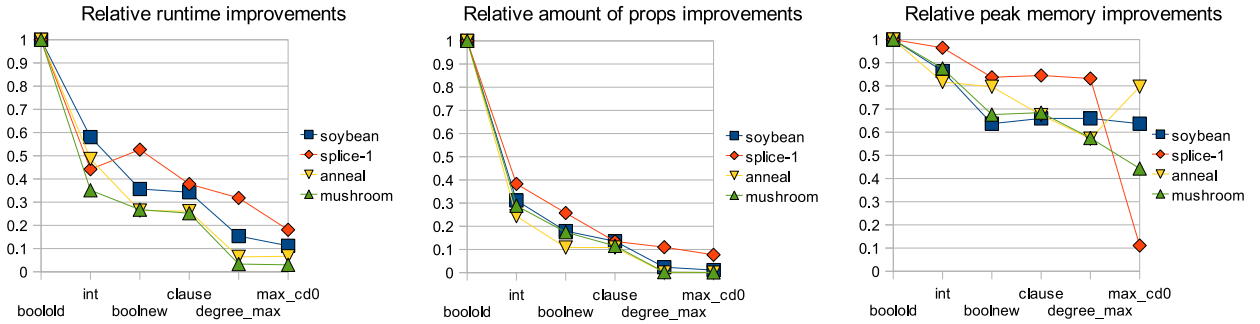


Fig. A.15. Relative improvements from applying several heuristics.

but needs significantly less memory. For our application it is thus faster and less memory intensive to clone every node in the search tree and choose the maximum value first.

A.5. Summary

An overview of the relative improvements of each step can be found in Fig. A.15. Overall, we see that even though our initial model – using reified constraints – could be specified straightforwardly, the scalability of the approach is highly depended on making the right low-level decisions, as discussed in this section. Only this modeling process as a whole can make the CP-based approach competitive with current specialized systems for constraint-based itemset mining.

References

[1] Rakesh Agrawal, Tomasz Imielinski, Arun N. Swami, Mining association rules between sets of items in large databases, in: Proceedings of the ACM SIGMOD International Conference on Management of Data, ACM Press, 1993, pp. 207–216.
[2] Rakesh Agrawal, Hiekki Mannila, Ramakrishnan Srikant, Hannu Toivonen, A. Inkeri Verkamo, Fast discovery of association rules, in: Advances in Knowledge Discovery and Data Mining, AAAI Press, 1996, pp. 307–328.
[3] Krzysztof R. Apt, Mark Wallace, Constraint Logic Programming Using Eclipse, Cambridge University Press, New York, NY, USA, 2007.
[4] Stephen D. Bay, Michael J. Pazzani, Detecting change in categorical data: mining contrast sets, in: Proceedings of the Fifth International Conference on Knowledge Discovery and Data Mining, ACM Press, 1999, pp. 302–306.
[5] Roberto J. Bayardo Jr., Rakesh Agrawal, Dimitrios Gunopulos, Constraint-based rule mining in large, dense databases, Data Mining and Knowledge Discovery 4 (2/3) (2000) 217–240.
[6] Nicolas Beldiceanu, Mats Carlsson, Sophie Demassey, Thierry Petit, Global constraint catalogue: past, present and future, Constraints 12 (March 2007) 21–62.
[7] Christian Bessiere, Emmanuel Hebrard, Barry O'Sullivan, Minimising decision tree size as combinatorial optimisation, in: Principles and Practice of Constraint Programming, in: Lecture Notes in Computer Science, vol. 5732, Springer, 2009, pp. 173–187.
[8] Francesco Bonchi, Bart Goethals, FP-bonsai: the art of growing and pruning small fp-trees, in: Advances in Knowledge Discovery and Data Mining, in: Lecture Notes in Computer Science, vol. 3056, Springer, 2004, pp. 155–160.
[9] Francesco Bonchi, Claudio Lucchese, Extending the state-of-the-art of constraint-based pattern discovery, Data and Knowledge Engineering 60 (2) (2007) 377–399.
[10] Sally C. Brailsford, Chris N. Potts, Barbara M. Smith, Constraint satisfaction problems: algorithms and applications, European Journal of Operational Research 119 (3) (1999) 557–581.
[11] Cristian Bucila, Johannes Gehrke, Daniel Kifer, Walker M. White, DualMiner: a dual-pruning algorithm for itemsets with constraints, Data Mining and Knowledge Discovery 7 (3) (2003) 241–272.
[12] Ming-Wei Chang, Lev-Arie Ratinov, Nicholas Rizzolo, Dan Roth, Learning and inference with constraints, in: Proceedings of the Twenty-Third AAAI Conference on Artificial Intelligence, AAAI Press, 2008, pp. 1513–1518.
[13] Hong Cheng, Yan Xifeng, Jiawei Han, Chih-Wei Hsu, Discriminative frequent pattern analysis for effective classification, in: Proceedings of the 23rd International Conference on Data Engineering, IEEE, 2007, pp. 716–725.
[14] Hong Cheng, Yan Xifeng, Jiawei Han, P.S. Yu, Direct discriminative pattern mining for effective classification, in: Proceedings of the 24th International Conference on Data Engineering, IEEE, 2008, pp. 169–178.
[15] James Cussens, Bayesian network learning by compiling to weighted max-sat, in: Proceedings of the 24th Conference in Uncertainty in Artificial Intelligence, AUAI Press, 2008, pp. 105–112.

[16] Luc De Raedt, Tias Guns, Siegfried Nijssen, Constraint programming for itemset mining, in: Proceeding of the 14th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, ACM, 2008, pp. 204–212.
[17] Luc De Raedt, Tias Guns, Siegfried Nijssen, Constraint programming for data mining and machine learning, in: Proceedings of the Twenty-Fourth AAAI Conference on Artificial Intelligence, AAAI Press, 2010, pp. 1513–1518.
[18] Luc De Raedt, Stefan Kramer, The levelwise version space algorithm and its application to molecular fragment finding, in: Proceedings of the Seventeenth International Joint Conference on Artificial Intelligence, Morgan Kaufmann, 2001, pp. 853–862.
[19] Luc De Raedt, Albrecht Zimmermann, Constraint-based pattern set mining, in: Proceedings of the Seventh SIAM International Conference on Data Mining, SIAM, 2007, pp. 1–12.
[20] Guozhu Dong, Jinyan Li, Efficient mining of emerging patterns: discovering trends and differences, in: Proceedings of the Fifth International Conference on Knowledge Discovery and Data Mining, ACM Press, 1999, pp. 43–52.
[21] Wei Fan, Kun Zhang, Hong Cheng, Jing Gao, Yan Xifeng, Jiawei Han, Philip S. Yu, Olivier Verscheure, Direct mining of discriminative and essential frequent patterns via model-based search tree, in: Proceedings of the 14th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, ACM, 2008, pp. 230–238.
[22] Alan M. Frisch, Matthew Grum, Christopher Jefferson, Bernadette Martínez Hernández, Ian Miguel, The design of essence: a constraint language for specifying combinatorial problems, in: Proceedings of the 20th International Joint Conference on Artificial Intelligence, Morgan Kaufmann, 2007, pp. 80–87.
[23] Johannes Fürnkranz, Peter A. Flach, ROC 'n' rule learning – towards a better understanding of covering algorithms, Machine Learning 58 (1) (2005) 39–77.
[24] Bernhard Ganter, Gerd Stumme, Rudolf Wille (Eds.), Formal Concept Analysis, Foundations and Applications, Lecture Notes in Computer Science, vol. 3626, Springer, 2005.
[25] Gecode Team, http://www.gecode.org.
[26] Ian P. Gent, Christopher Jefferson, Ian Miguel, MINION: a fast scalable constraint solver, in: Proceeding of the 17th European Conference on Artificial Intelligence, IOS Press, 2006, pp. 98–102.
[27] Bart Goethals, Mohammed J. Zaki, Advances in frequent itemset mining implementations: report on FIMI'03, in: SIGKDD Explorations Newsletter, vol. 6, 2004, pp. 109–117.
[28] Henrik Grosskreutz, Stefan Rüping, Stefan Wrobel, Tight optimistic estimates for fast subgroup discovery, in: Machine Learning and Knowledge Discovery in Databases, in: Lecture Notes in Computer Science, vol. 5211, Springer, 2008, pp. 440–456.
[29] Tias Guns, Siegfried Nijssen, Luc De Raedt, *k*-Pattern set mining under constraints, CW Reports CW596, Department of Computer Science, K.U. Leuven, October 2010.
[30] J. Han, J. Pei, Y. Yin, Mining frequent patterns without candidate generation, in: Proceedings of the ACM SIGMOD International Conference on Management of Data, ACM Press, 2000, pp. 1–12.
[31] Jiawei Han, Hong Cheng, Dong Xin, Xifeng Yan, Frequent pattern mining: current status and future directions, Data Mining and Knowledge Discovery 15 (1) (2007) 55–86.
[32] Branko Kavsek, Nada Lavrac, Viktor Jovanoski, APRIORI-SD: adapting association rule learning to subgroup discovery, in: Advances in Intelligent Data Analysis, in: Lecture Notes in Computer Science, vol. 2810, Springer, 2003, pp. 230–241.
[33] Heikki Mannila, Hannu Toivonen, Levelwise search and borders of theories in knowledge discovery, Data Mining and Knowledge Discovery 1 (3) (1997) 241–258.
[34] Yasuhiko Morimoto, Takeshi Fukuda, Hirofumi Matsuzawa, Takeshi Tokuyama, Kunikazu Yoda, Algorithms for mining association rules for binary segmentations of huge categorical databases, in: Proceedings of 24rd International Conference on Very Large Data Bases, Morgan Kaufmann, 1998, pp. 380–391.
[35] Shinichi Morishita, Jun Sese, Traversing itemset lattice with statistical metric pruning, in: Proceedings of the Nineteenth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, ACM, 2000, pp. 226–236.
[36] Siegfried Nijssen, Élisa Fromont, Optimal constraint-based decision tree induction from itemset lattices, Data Mining and Knowledge Discovery 21 (1) (2010) 9–51.
[37] Siegfried Nijssen, Tias Guns, Integrating constraint programming and itemset mining, in: Machine Learning and Knowledge Discovery in Databases, in: Lecture Notes in Computer Science, vol. 6322, Springer, 2010, pp. 467–482.
[38] Siegfried Nijssen, Tias Guns, Luc De Raedt, Correlated itemset mining in ROC space: a constraint programming approach, in: Proceedings of the 15th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, ACM, 2009, pp. 647–656.
[39] Siegfried Nijssen, Joost N. Kok, Multi-class correlated pattern mining, in: Knowledge Discovery in Inductive Databases, in: Lecture Notes in Computer Science, vol. 3933, Springer, 2005, pp. 165–187.
[40] Nicolas Pasquier, Yves Bastide, Rafik Taouil, Lotfi Lakhal, Discovering frequent closed itemsets for association rules, in: Database Theory, in: Lecture Notes in Computer Science, vol. 1540, Springer, 1999, pp. 398–416.
[41] Jian Pei, Jiawei Han, Can we push more constraints into frequent pattern mining? in: Proceedings of the sixth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, ACM, 2000, pp. 350–354.
[42] Jian Pei, Jiawei Han, Laks V.S. Lakshmanan, Mining frequent item sets with convertible constraints, in: Proceedings of the IEEE International Conference on Data Engineering, IEEE, 2001, pp. 433–442.
[43] Jian Pei, Jiawei Han, Runying Mao, Closet: an efficient algorithm for mining frequent closed itemsets, in: ACM SIGMOD Workshop on Research Issues in Data Mining and Knowledge Discovery, ACM, 2000, pp. 21–30.
[44] Laurent Perron, Search procedures and parallelism in constraint programming, in: Principles and Practice of Constraint Programming, in: Lecture Notes in Computer Science, vol. 1713, Springer, 1999, pp. 346–360.
[45] Francesca Rossi, Peter van Beek, Toby Walsh, Handbook of Constraint Programming (Foundations of Artificial Intelligence), Elsevier Science Inc., 2006.
[46] Christian Schulte, Programming Constraint Services: High-Level Programming of Standard and New Constraint Services, Lecture Notes in Computer Science, vol. 2302, Springer, 2002.
[47] Christian Schulte, Peter J. Stuckey, Efficient constraint propagation engines, Transactions on Programming Languages and Systems 31 (1) (2008) 1–43.
[48] Jun Sese, Shinichi Morishita, Answering the most correlated *n* association rules efficiently, in: Principles of Data Mining and Knowledge Discovery, in: Lecture Notes in Computer Science, vol. 2431, Springer, 2002, pp. 410–422.
[49] Pradeep Shenoy, Jayant R. Haritsa, S. Sudarshan, Gaurav Bhalotia, Mayank Bawa, Shah Devavrat, Turbo-charging vertical mining of large databases, in: Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data, ACM, 2000, pp. 22–33.
[50] Arnaud Soulet, Bruno Crømilleux, An efficient framework for mining flexible constraints, in: Advances in Knowledge Discovery and Data Mining, in: Lecture Notes in Computer Science, vol. 3518, Springer, 2005, pp. 43–64.
[51] Takeaki Uno, Masashi Kiyomi, Hiroki Arimura, LCM ver.3: collaboration of array, bitmap and prefix tree for frequent itemset mining, in: Proceedings of the 1st International Workshop on Open Source Data Mining, ACM, 2005, pp. 77–86.
[52] Pascal Van Hentenryck, Yves Deville, in: The Cardinality Operator: A New Logical Connective for Constraint Logic Programming, MIT Press, Cambridge, MA, USA, 1993, pp. 383–403.

[53] Pascal Van Hentenryck, Laurent Perron, Jean-Francois Puget, Search and strategies in OPL, ACM Transations Computational Logic 1 (2) (2000) 285–320.
[54] Pascal Van Hentenryck, Vijay A. Saraswat, Yves Deville, Design, implementation, and evaluation of the constraint language cc(FD), Journal of Logic Programming 37 (1–3) (1998) 139–164.
[55] Stefan Wrobel, An algorithm for multi-relational discovery of subgroups, in: Principles of Data Mining and Knowledge Discovery, in: Lecture Notes in Computer Science, vol. 1263, Springer, 1997, pp. 78–87.
[56] Mohammed Javeed Zaki, Karam Gouda, Fast vertical mining using diffsets, in: Proceedings of the Ninth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, ACM, 2003, pp. 326–335.