MiningZinc: A Modeling Language for Constraint-based Mining

Tias Guns[1], **Anton Dries**[1], **Guido Tack**[2], **Siegfried Nijssen**[1,3] **and Luc De Raedt**[1]
[1] Department of Computer Science, KU Leuven {*firstname.lastname*}@cs.kuleuven.be
[2] Caulfield School of Information Technology, Monash University guido.tack@monash.edu
[3] LIACS, Universiteit Leiden snijssen@liacs.nl

Abstract

We introduce MiningZinc, a general framework for constraint-based pattern mining, one of the most popular tasks in data mining. MiningZinc consists of two key components: a language component and a toolchain component.

The language allows for high-level and natural modeling of mining problems, such that MiningZinc models closely resemble definitions found in the data mining literature. It is inspired by the *Zinc* family of languages and systems and supports user-defined constraints and optimization criteria.

The toolchain allows for finding solutions to the models. It ensures the solver independence of the language and supports both standard constraint solvers and specialized data mining systems. Automatic model transformations enable the efficient use of different solvers and systems.

The combination of both components allows one to rapidly model constraint-based mining problems and execute these with a wide variety of methods. We demonstrate this experimentally for a number of well-known solvers and data mining tasks.

1 Introduction

The field of data mining has seen enormous progress, evidenced by numerous successful applications and systems. Nevertheless, it remains hard and cumbersome to tackle new data mining problems and applications. The reasons are that different problems are governed by different requirements and constraints, and that most mining problems are computationally hard to solve; hence naïve algorithms typically do not work well. This explains why data mining has focussed so much on developing specific algorithms, solutions and constraints, and has given less attention to the development of generic solution strategies. There is little support for formalising a mining task and capturing a problem specification in a declarative way. Developing and implementing the algorithms is labor intensive with only limited re-use of software. The typical iterative nature of the knowledge-discovery cycle [Han and Kamber, 2000] further complicates this process.

The data mining practice contrasts sharply with that of constraint programming, where high-level languages such as Zinc [Marriott *et al.*, 2008], Essence [Frisch *et al.*, 2008] and OPL [Van Hentenryck, 1999] are used to *model* problems, and general purpose *solvers* are provided to compute solutions. Motivated by the success of this declarative approach in constraint programming, we propose a *modeling + solving* approach for data mining. This makes data mining more flexible, as it becomes easy to change the model and to select the best solvers to get solutions.

As the field of data mining is diverse, we focus in this paper on one of the most popular tasks, namely constraint-based pattern mining. Even for the restricted data type of binary databases, many settings (supervised and unsupervised) and corresponding systems have been proposed in the literature; this makes it an adequate showcase for a declarative approach to data mining. Dealing with a diverse set of constraint-based pattern mining problems remains an unsolved and important challenge in data mining.

The key contribution of this paper is the introduction of a general-purpose, declarative mining framework called MiningZinc. The design criteria for MiningZinc are:

- to support the *high-level and natural modeling* of pattern mining tasks; that is, MiningZinc models should closely correspond to the definitions of data mining problems found in the literature;
- to support *user-defined constraints and criteria* such that existing problem formulations can be extended and modified, and novel mining tasks can be specified;
- to be *solver-independent*, such that the best solving method can be selected for the problem and data at hand. Supported methods should include both *general purpose solvers* and *specialized efficient mining algorithms*;
- to *build on* and *extend* existing constraint programming and data mining techniques, capitalizing on and extending the state-of-the-art in these fields.

In data mining, to date there is no framework that supports these four design criteria. Especially the combination of user-defined constraint and solver-independence is uncommon (we defer a detailed discussion of related work to Section 5). In the constraint programming community, however, the design of the Zinc [Marriott *et al.*, 2008; Nethercote *et al.*, 2007] family of languages and frameworks is in line with the

above criteria. The main question in this paper is hence how to extend this framework to support constraint-based pattern mining. We contribute: 1) a novel library of functions and constraints to support modelling pattern mining tasks in the MiniZinc language; 2) multiple instantiations of the library to enable efficient solving of models; and 3) the extension of the MiniZinc toolchain to support specialized mining algorithms and to recognize when these algorithms can be used.

The MiningZinc framework builds on our earlier CP4IM framework [Guns *et al.*, 2011], which showed the feasibility of constraint programming for pattern mining. Compared to this earlier framework, key novelties in MiningZinc are: 1) the present framework supports a wide variety of different solvers (including DM algorithms and general purpose solvers); 2) a significantly more high-level modeling language is employed.

Our experiments show that MiningZinc can be highly effective for established mining tasks, while still supporting user-defined constraints. This is possible thanks to the use of both generic (constraint) solving algorithms and highly specialized mining algorithms. We also demonstrate how the integration of different solvers and algorithms in one framework enables a systematic comparison of such methods.

The rest of this paper is organized as follows. In Section 2 we describe how constraint-based mining problems can be modeled in MiningZinc. Section 3 shows how such models can be analysed and transformed so that they can be solved efficiently using existing constraint solvers and specialized algorithms. In Section 4 we analyze the performance of the MiningZinc toolchain and show that it can be used to solve different constraint-based modeling problems and to compare different solving strategies. Finally, Sections 5 and 6 provide related work and conclusions.

2 Modeling constraint-based mining

In this section, we introduce constraint-based itemset mining, show how it can be modeled in standard MiniZinc and how MiningZinc can ease modelling. We then show a number of different tasks modeled in MiningZinc to illustrate the generality of the approach.

2.1 Constraint-based itemset mining

Itemset mining was introduced by [Agrawal *et al.*, 1993] and can be defined as follows. Given is an itemset database, containing a set of *transactions* each consisting of an identifier and a set of *items*. We denote the set of transaction identifiers as $\mathcal{S} = \{1, \ldots, n\}$ and the set of all items as $\mathcal{I} = \{1, \ldots, m\}$. An itemset database $\mathcal{D}$ maps transaction identifiers $t \in \mathcal{S}$ to sets of items: $\mathcal{D}(t) \subseteq \mathcal{I}$.

Definition 1 (Frequent Itemset Mining)**.** Given an itemset database $\mathcal{D}$ and a threshold *Freq*, the frequent itemset mining problem consists of finding all itemsets $I \subseteq \mathcal{I}$ such that $|\phi_{\mathcal{D}}(I)| > Freq$, where $\phi_{\mathcal{D}}(I) = \{t | I \subseteq \mathcal{D}(t)\}$.

The set $\phi_{\mathcal{D}}(I)$ is called the *cover* of the itemset, and the threshold *Freq* the *minimum frequency* threshold. An itemset I which has $|\phi_{\mathcal{D}}(I)| \geq Freq$ is called a *frequent itemset*.

Constraint-based pattern mining methods can leverage additional constraints during the pattern discovery process. One of the most fundamental and best studied problems is that of *constraint-based itemset mining* [Boulicaut and Jeudy, 2010]. A lot of research has focused on how constraints, such as the minimum frequency constraint, can be incorporated efficiently in the search process; cf. [Agrawal *et al.*, 1993; Mannila and Toivonen, 1997; Bonchi and Lucchese, 2007].

Listing 1: "Constraint-based mining"

```
1 int: NrI; int: NrT; int: Freq;
2 array[1..NrT] of set of 1..NrI: TDB;
3 var set of 1..NrI: Items;
4 constraint card(cover(Items,TDB)) >= Freq;
5 solve satisfy;
```

Listing 2: "Constraint-based mining - cover"

```
1 function var set of int: cover(
2   var set of int: Items,
3   array[int] of var set of int: D) = let {
4     var set of index_set(D): Trans;
5     constraint forall (t in index_set(D))
6         ( t in Trans <-> Items subset D[t] );
7   } in Trans;
```

2.2 Itemset mining in MiniZinc

For modelling, MiningZinc builds on the MiniZinc language. MiniZinc [Nethercote *et al.*, 2007] is a popular restricted version of Zinc [Marriott *et al.*, 2008]. It it is powerful enough to support the built-in types *bool, int, set* and *float*, user defined-predicates, and more recently, user-defined functions [Stuckey and Tack, 2013], while still being executable.

Pattern mining problems can be modeled directly in MiniZinc. A MiniZinc model of the frequent itemset mining problem is shown in Listing 1. Lines 1 and 2 represent the parameters and data that can be provided through a separate data file. It represents the items and transaction identifiers in $\mathcal{I}$ and $\mathcal{S}$ by natural numbers (from 1 to NrI and 1 to NrT respectively) and the dataset $\mathcal{D}$ by the array TDB, mapping each transaction identifier to the corresponding set of items. The set of items we are looking for is modelled on line 3 as a *set variable* with elements between value 1 and NrI. The minimal frequency constraint is posted on line 4, which naturally corresponds to the formal notation $|\phi_{\mathcal{D}}(I)| \geq Freq$.

The cover function on line 4 corresponds to $\phi_{\mathcal{D}}(I)$. An implementation of this function is shown in Listing 2, and closely matches the formal definition $\phi_{\mathcal{D}}(I) = \{t | I \subseteq \mathcal{D}(t)\}$.

This example demonstrates the appeal of using a modeling language like MiniZinc for pattern mining: the formulation is high-level, declarative and close to the mathematical notation, it allows for user-defined constraints like the *cover* relation between items and transactions, and it is solver-independent.

2.3 MiningZinc

In the example above we defined the cover function using the primitives present in MiniZinc. An important feature of MiniZinc is that common functions and predicates can be put into libraries, which makes it possible to reuse them in different tasks. In this way, MiniZinc can be extended to different application domains, without the need for developing a new language. The language component of the MiningZ-

Listing 3: "Itemset Mining constraints"

```
1 % Closure
2 constraint
3     Items = cover_inv(cover(Items,TDB),TDB);
4 % Minimum cost
5 array[1..NrI] of int: item_c; int: Cost;
6 constraint sum(i in Items) (item_c[i]) >= Cost;
```

Listing 4: "High Utility mining"

```
1 % Utility data
2 array [1..NrI] of int: ItemPrice; int: Util;
3 array [1..NrT, 1..NrI] of int: UTDB;
4 constraint
5   sum (t in cover(Items,TDB), i in Items) (
6     ItemPrice[i]*UTDB[t,i] ) >= Util;
7 constraint card(Items) >= 1;
8 solve satisfy;
```

Listing 5: "Discriminative itemset mining (accuracy)"

```
1 array[int] of set of int: D_fraud;
2 array[int] of set of int: D_ok;
3 var set of 1..NrI: Items;
4 constraint Items = cover_inv(
5              cover(Items,D_fraud),D_fraud);
6 % Optimisation function
7 var int: Score = card(cover(Items, D_fraud)) -
8                  card(cover(Items, D_ok));
9 solve maximize Score;
```

inc framework is such a library that extends MiniZinc with functions and predicates commonly used in pattern mining.

We will often write "MiningZinc model" to describe a MiniZinc model that uses elements of the MiningZinc library.

2.4 More examples

Let us now illustrate the use of the MiningZinc library on three diverse but representative tasks.

Itemset Mining with constraints. Listing 3 contains some examples of constraint-based mining constraints that can be added to the frequent itemset mining model in Listing 1. Line 3 represents the popular closure constraint $I = \psi_{\mathcal{D}}(\phi_{\mathcal{D}}(I))$, where $\psi_{\mathcal{D}}(T) = \{i \in \mathcal{I} | \forall t \in T : i \in \mathcal{D}(t)\}$. This closure constraint, together with a minimum frequency constraint, represents the *closed itemset mining* problem [Pasquier *et al.*, 1999]. "cover_inv" is part of our MiningZinc library and corresponds to $\psi_{\mathcal{D}}(T)$.

Lines 5/6 represent a common cost-based constraint [Bonchi and Lucchese, 2007]; it constrains the itemset to have a cost of at least `Cost`, with `item_cost` and `Cost` being a user-supplied array of costs and a cost threshold.

We can use the full expressive power of MiniZinc to define constraints. This includes constraints in propositional logic, for example expressing implications between (groups of) items/transactions, or inclusion/exclusion relations between elements. Constraints can involve external data too, as exemplified with a constraint on item *costs*.

High Utility mining In high utility mining [Chan *et al.*, 2003] the goal is to search for itemsets with a total utility above a certain threshold. Assumed given is an *external* utility e for each individual item, for example its price, and a utility matrix U that contains for each transaction a *local* utility of each item in the transaction, for example the quantity of that item in that transaction. The total utility of an itemset is $\sum_{t \in \phi_{\mathcal{D}}(I)} \sum_{i \in I} e(i)U(t,i)$.

Listing 4 shows the MiningZinc model corresponding to this task. Lines 2 and 3 declare the data, and lines 4 to 6 constrain the utility.

Discriminative itemset mining is the task of finding the itemset for which the coverage shows the strongest correlation with a given class attribute. Consider for example a transaction database with fraudulent transactions and non-fraudulent ones, and the task of finding itemsets correlating with fraudulent behaviour. Many different measures can be used to define what a good correlation is. This has led to tasks known as discriminative itemset mining, correlated itemset mining, subgroup discovery, contrast set mining and more [Novak *et al.*, 2009]; a modeling language would be useful to allow for the implementation of these variants.

A discriminative itemset mining task is shown in Listing 5. This is an optimisation problem, and the score to optimise is defined on line 7. The score is $p - n$ where p is the number of positive transactions covered and n the number of negative ones. Optimising this score (line 9) corresponds to optimising the *accuracy* measure; see [Fürnkranz and Flach, 2005] for more details. An additional constraint ensures that the patterns are closed, but only on the transactions in the fraudulent transactions (one can show that there must always be a closed itemset that maximizes this core). Note how we reuse the functions from Listing 1 for this different task.

3 Solving MiningZinc models

The language discussed in the previous section is solver-independent; at no point did we discuss how to find solutions for the models. In principle, we could now use the existing MiniZinc toolchain to find solutions for these models. This can be achieved by adding an implementation of the MiningZinc library using standard MiniZinc constructs. However, this rather straight-forward approach does not result in very effective solving.

For example, the models in the previous section are formulated with constraints on set variables. This is the most natural approach to model constraint-based mining problems. However, from past work [Guns *et al.*, 2011] it is known that a Boolean representation of the sets is more efficient. More is known about how to efficiently solve such problems, such as reformulations of constraints. Furthermore, for certain solvers it might be better to encode some constraints in a different way.

Hence, a more powerful solution would be one in which different low-level representations can be used for the same high-level model; furthermore, it would be desirable to include specialized data mining algorithms in the toolchain. Our extensions achieve this.

Specifically, we employ three approaches to improve performance:

1. *Model transformations:* By providing different versions of the MiningZinc library we can transform the given models into optimized variants for different types of solvers.

2. *Use of specialized solvers:* There exist highly optimized and scalable algorithms for constraint-based itemset mining. The MiningZinc toolchain can analyze the models provided, and determine whether there are specialized algorithms that can solve (part of) the given model.
3. *A portfolio of solvers:* Using static analysis techniques, MiningZinc can determine what transformation/solver combinations are capable of solving the model. Hence, it offers a portfolio of solvers, including general solvers and specialised algorithms, from which to choose.

3.1 Model transformations

An overview of the MiningZinc toolchain is given in Figure 1.

It makes extensive use of the existing MiniZinc toolchain. The main benefit of the standard MiniZinc toolchain (top-right of Figure 1) is that it transforms a model to FlatZinc, a solver-specific language that is supported by a large number of constraint solvers. This solver-specificity is obtained by including a solver-specific library. During the transformation from MiniZinc to FlatZinc, the MiniZinc toolchain replaces the functions and predicates in the libraries by their corresponding body. This includes the solver-specific transformations ($\mathcal{T}_{solver}$ in Figure 1). More details on these transformations can be found in [Stuckey and Tack, 2013].

Remember that the MiningZinc language is MiniZinc with the addition of the MiningZinc library. Given a MiningZinc model, data and the MiningZinc library, the MiniZinc toolchain will transform the model into FlatZinc, which can then be fed into a constraint solver. Any constraint solver that supports FlatZinc with integer variables can then be used. For many mining tasks, it is also required that the solver can enumerate all solutions.

The MiningZinc library embeds best-practices in constraint-based mining, as found in the CP4IM framework [Guns *et al.*, 2011]. For instance, in this earlier work it was found that the minimum frequency constraint can be specified as card(cover(Items,TDB)) >= Freq but it can also be equivalently formulated over every item separately, resulting in better propagation.

To also support the use of a Boolean representation of the sets, we have developed two instantiations of the MiningZinc library: one that operates on the set variables ($\mathcal{T}_{set}$ in Figure 1) and one that converts the sets into Boolean arrays and posts the constraints on these Boolean arrays ($\mathcal{T}_{bool}$).

For $\mathcal{T}_{bool}$, we exploit MiniZinc's ability to redefine built-in predicates in a library. In the redefinition, each set variable is linked to an array of Booleans, and the equivalent predicate on the Boolean representation is called. By virtue of *common subexpression elimination*, each set variable will only have one corresponding Boolean array of variables.

Apart from this generic set-to-boolean transformation, the Boolean instantiation of the MiningZinc library takes care not to introduce unnecessary copies of decision variables. For example, for the constraint card(Items intersect {1,...}) there is no need to represent the intersection using a new copy of Items, rather, a subset of the Boolean variables representing Items can be used directly to calculate the cardinality. The search strategy is also instantiated to an appropriate heuristic over the Boolean variables.

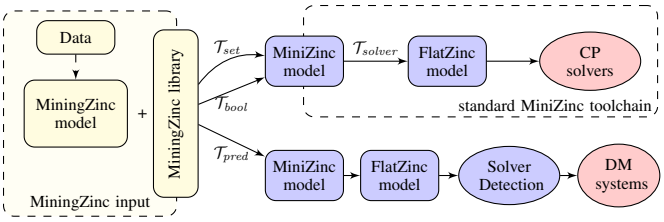


Figure 1: Overview of the MiningZinc toolchain

Section 4 will show how MiningZinc models that are automatically transformed to Boolean variables match the performance of hand-made MiniZinc models over Boolean variables.

3.2 Use of specialized algorithms

Data mining algorithms are typically highly efficient on one specific task. Even constraint-based mining algorithms only support a small number of related tasks. MiningZinc on the other hand is a general, solver-independent language for expressing constraint-based mining problems. Hence, we want to be able to plug in specialized mining algorithms as backend solvers. The main challenge is to determine whether a MiningZinc model can be solved (partially) using such an existing data mining algorithm.

Our approach is depicted in Figure 1 (bottom). We first transform the model into FlatZinc predicates. In the solver detection step, we convert the predicates to (Prolog) facts over which we then reason using resolution. As we would like to reason over the high-level predicates and functions in the model, we want to avoid decomposing the constraints in the conversion to FlatZinc. To this end, we provide a third instantiation of the MiningZinc library, $\mathcal{T}_{pred}$. It transforms all the functions to predicates, and leaves all predicates unimplemented. Furthermore, we do not specify any solver-specific library, thereby again avoiding the decomposition of predicates.

Converting the FlatZinc into facts is then a matter of filtering out the constraints and removing FlatZinc keywords and possible annotations. For example, a model describing the frequent itemset mining problem (e.g. Listing 1 with Freq set to 20) can be transformed into the following set of facts.

```
cover(Items, TDB, Aux1).
card_set(Aux1, Aux2).
int_lq(20, Aux2).
```

To reason over the model, we describe the capabilities of a data mining algorithm by specifying which combinations of constraints (facts) they can handle. For example, an algorithm capable of solving the frequent itemset mining problem can be described by the following Prolog clause:

```
im_frequent(TDB, Freq, Items, Cover) :-
    cover(Items, TDB, Cover),
    card_set(Cover, Size),
    int_lq(Freq, Size).
```

Table 1: Overhead of solver detection, runtime in seconds. #Tr=nr. of transactions, #It = nr. of items, %D = density

Dataset	#Tr	#It	%D	Freq	Fr+Clo	Fr+Clo+ Mincost
zoo-1	101	36	44%	0.049	0.051	0.085
primary-tumor	336	31	48%	0.057	0.060	0.090
soybean	630	50	32%	0.090	0.090	0.159
german-credit	1000	110	35%	0.142	0.150	0.273
hypothyroid	3247	86	50%	0.337	0.347	0.451
mushroom	8124	112	19%	0.424	0.471	0.551
pumsb_star	49046	2088	2.4%	8.513	8.710	21.842
retail	88162	16470	0.1%	2.549	2.765	57.205
T10I4D100K	100000	870	1.2%	2.689	3.471	3.006
T40I10D100K	100000	942	4.2%	8.952	9.306	10.146

By evaluating this clause on the set of facts above, we see that (part of) the model matches with the frequent itemset mining problem. Furthermore, we can extract parameters from the model (e.g. the frequency threshold `Freq`), to provide it to a mining algorithm for example.

Note that many constraints can be specified in different ways. For example, the `cover` constraint could be formulated over every item separately, instead of using the MiningZinc function. A set of background predicates that contain alternative formulations of the MiningZinc predicates, as well as alternative formulations of FlatZinc predicates, is used to make the detection mechanism more robust.

A data mining algorithm can only be used directly, without post-processing, if the rewritten query conforms exactly to the original conjunctive query, that is, there are no extra constraints (facts). However, the ability to detect whether a part of a MiningZinc model conforms to a supported mining task can also be exploited to create a hybrid approach.

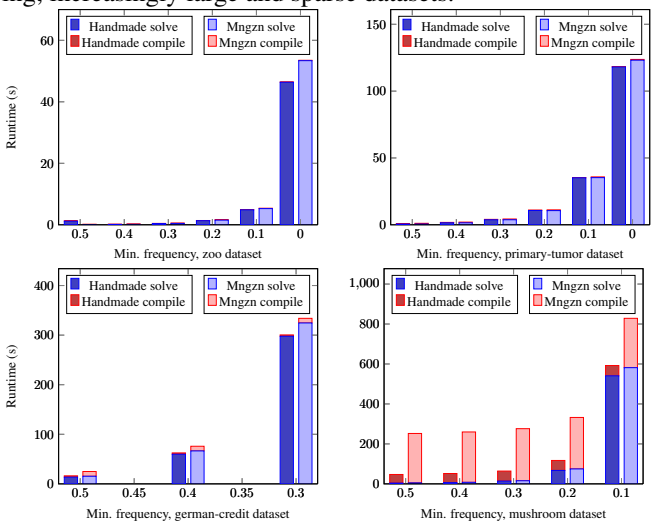
3.3 A portfolio of solvers

Using the above transformations, it is possible to use both generic constraint solvers (potentially with different transformations) and specialised mining algorithms. A third option is to use a hybrid approach, where part of the model is solved using one method, and another part by another method.

For example, most existing data mining algorithms are highly efficient but support only a few constraints. When faced with a new, e.g. domain specific, constraint, one has to (tediously) modify the algorithm. A popular alternative is to write a post-process routine, reusing the highly efficient algorithm without having to change it. In MiningZinc, should part of a model be supported by a mining algorithm, we have the unique ability to first use that algorithm to solve the subproblem, after which each of the solutions can be verified against the full model using a constraint solver to check the constraints. The same high-level language can thus be used to express all constraints. MiningZinc users need not even be aware which constraints are supported by which mining algorithms.

The current MiningZinc framework can detect which solvers are supported, and provide the user with a list of solving strategies. The use of automatic portfolio selection techniques [Xu *et al.*, 2008; Malitsky and Sellmann, 2012] is a promising direction for further research.

Figure 2: Solve and compile times for frequent itemset mining; increasingly large and sparse datasets.



4 Experiments

We now compare the different solving strategies that MiningZinc supports. We focus on two main questions: 1) what is the computational overhead of using MiningZinc, compared to manually writing low-level specifications or calling solvers; 2) what are the strengths and weaknesses of the different solving strategies.

The constraint solvers used are the g12 solvers that are part of the MiniZinc 1.6 distribution [Nethercote *et al.*, 2007] and Gecode 3.7.3 [Schulte *et al.*, 2013]. The constraint-based mining algorithms are LCM version 2 and 5[Uno *et al.*, 2004] and Christian Borgelt's implementations of Apriori (v5.73), Eclat (v3.74) and FPGrowth (v4.48) [Borgelt, 2012]; these are state-of-the art for efficient constraint-based mining. The datasets are from the UCI Machine Learning repository [Frank and Asuncion, 2010][1] and from the FIMI repository [Goethals and Zaki, 2004]. Experiments were run on computers with quad-core Intel i7 processors and 16Gb of ram. The MiningZinc system and datasets used can be downloaded at `http://dtai.cs.kuleuven.be/CP4IM/miningzinc/`.

4.1 Overhead of MiningZinc transformations

We first measure the difference in runtime between a fine-tuned hand-made MiniZinc model over Booleans and a MiningZinc model over sets that is transformed to Booleans ($\mathcal{T}_{bool}$ in Figure 1). The top of Figure 2 shows, for increasingly large datasets, the solve and compile times for standard frequent itemset mining. Solve times are comparable though slightly slower for MiningZinc, because the automatic transformation introduces a few more auxiliary variables. Compile times are not influenced by the frequency threshold but do increase with dataset size. We believe this is due to inefficiencies in the FlatZinc transformation tool, which has not been tested under this load before. The observed behavior is similar for more complex tasks.

[1] Downloaded from http://dtai.cs.kuleuven.be/CP4IM/datasets/

Table 2: Discriminative itemset mining runtimes

Dataset	g12CPX set	Gecode bool	Gecode set	g12CPX bool	g12Mip bool
anneal	1.1	20.61	37.25	20	ERR
audiology	0.21	2.01	0.22	2.14	72.85
australian-credit	1.22	52.74	74.15	129.51	ERR
german-credit	2.25	164.33	192.85	917.48	ERR
heart-cleveland	0.89	45.57	55.71	132.99	>1800
hepatitis	0.13	1.54	1.4	1.7	134.39
hypothyroid	15.22	485.07	917.9	674.27	ERR
kr-vs-kp	35.28	835.49	>1800	717.25	ERR
lymph	0.13	0.86	0.34	1.18	>1800
mushroom	56.64	376.3	78.32	293.84	ERR
primary-tumor	0.22	1.41	1.21	2.09	516.35
soybean	0.29	2.74	0.34	3.66	1160.70
splice-1	161.29	619.03	224.03	>1800	ERR
tic-tac-toe	0.68	2.76	0.66	3.06	>1800
vote	0.26	1.93	0.38	2.13	277.81
zoo-1	0.05	0.24	0.09	0.32	0.50
Total runtime	275.88	2612.66	3384.87	4701.63	20162.61

4.2 Overhead of DM solver detection

Table 1 shows, for increasingly large datasets, the overhead of converting the MiningZinc model into facts, and using Prolog to match the definitions of the mining tasks (Section 3.2) against it. The models used are combinations of the constraints shown in Listing 1 (Freq/Fr) and Listing 3 (Clo+MinCost).

Solving times are not shown as the input to the miners is what one would manually provide. The table shows that the overhead is rather low and increases with the complexity of task and the data. Note that mincost constrains the items, and hence is more sensitive to the number of items in the data.

4.3 Differences in CP encodings and solvers

We compare the use of set variables with the automatic translation to Boolean variables for a number of FlatZinc solvers. Figure 3 shows total running times (including compilation) on a few representable datasets for the representable task of closed frequent itemset mining. We observe that the Boolean encoding leads to lower runtimes as expected. In some cases though (as can be seen in the two plots on the right for high frequency values), the set encoding is faster, though it scales less well to lower frequency values. Using the Boolean encoding, the Gecode solver is consistently the fastest. One advantage of Gecode could be that it supports the *occurence* variable ordering, this strategy searches over the most constrained variable first, which is related to searching over the least frequent items first [Guns *et al.*, 2011], a search strategy known to perform well. The g12Lazy solver often returned duplicate solutions for low frequency thresholds.

Table 2 compares runtimes for the discriminative itemset mining task in Listing 5. It is an optimisation problem, which allows us to use the g12MIP solver. Unfortunately it suffers from memory problems on this task. g12FD and g12Lazy (not shown) have a total runtime around 8000 seconds. Interestingly, g12CPX using set variables is by far the fastest on this task, demonstrating that different solvers and encodings can be beneficial on different tasks.

4.4 Solvers across tasks

A framework such as MiningZinc allows for a systematic comparison of constraint solvers and mining algorithms across tasks and datasets. As can be witnessed in Table 3, the constraint-based itemset mining algorithms are significantly faster and more scalable than constraint solvers for the tasks they support. For other tasks, such as discriminative itemset mining, constraint solvers can be used with reasonable success, without having to implement new algorithms.

Table 3: Best solution method per task; between brackets is the sum of average runtime per dataset in seconds.

Task	Best solver	Second Best
Frequent	B-Fpgrowth (1480)	B-Eclat (1570)
Fr + closed	LCMv2 (1026)	LCMv5 (2244)
Fr + cl + minsize	LCMv5 (1820)	B-Eclat (3037)
Discriminative	g12CPX (275)	Gecode-bool (2612)

Experiments with a hybrid approach are not competitive at this point and not shown. The naive approach of checking each solution found by a mining algorithm against all constraints (including all the data) using a constraint solver requires more time than having a constraint solver search all solutions directly.

The main bottleneck for the constraint solvers remains the scalability towards large datasets. Figure 4 shows the runtimes of different algorithms for frequent itemset mining (the two plots on the left) and closed itemset mining (the two on the right). In these figures, CP4IM is a precompiled Gecode model for the task at hand.[2] Its performance is remarkably closer to that of specialised mining algorithms than the performance of using Gecode as a FlatZinc backend.

5 Related work

We have introduced MiningZinc, a framework for constraint-based pattern mining. Its key design criteria are 1) high-level and natural modeling of mining tasks, 2) support for user-defined constraints, 3) solver-independence and 4) building on existing languages and systems.

In the data mining field, our work is related to that on inductive databases [Mannila, 1997]; these are databases in which both data and patterns are first-class citizens and can be queried. Most inductive query languages, e.g., [Meo *et al.*, 1996; Imielinski and Virmani, 1999], extend SQL with primitives for pattern mining. They have only a restricted language for expressing mining problems, and are usually tied to one mining algorithm. A more advanced development is that of mining views [Blockeel *et al.*, 2012], which provides lazy access to patterns through a virtual table. Standard SQL can be used for querying, and the implementation will only materialise those patterns in the table that are relevant for the query. This is realized using a traditional mining algorithm.

Work on constraint solving for itemset mining [Guns *et al.*, 2011; Järvisalo, 2011] has used existing modeling languages. However, these approaches were *low-level* and *solver dependent*. The use of higher-level modeling languages and primitives has been studied before [Métivier *et al.*, 2012; Guns *et al.*, 2013], though again tied to one particular solving technology. MiningZinc on the other hand enables the use of both general constraint solvers and highly optimized mining algorithms. Best practices from solver-specific studies, such

[2] Obtained from http://dtai.cs.kuleuven.be/CP4IM/

Figure 3: Different CP solvers with bool/set encoding.

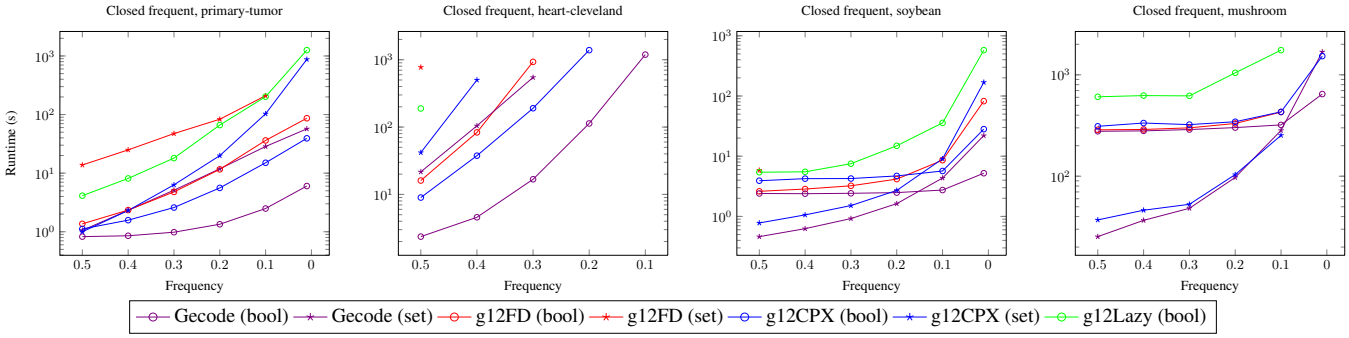
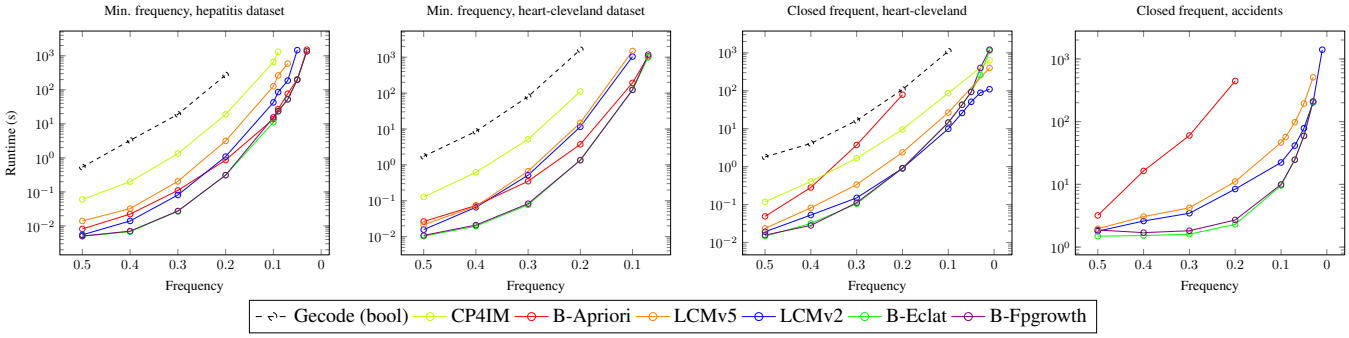


Figure 4: Different DM solvers (Gecode (bool) timed out on the right-most figure)



as the use of SAT solvers for itemset mining [Henriques *et al.*, 2012], could be incorporated into MiningZinc too.

We chose Zinc [Nethercote *et al.*, 2007] as the basis of our work because it is most in line with our design criteria. Other languages such as Essence [Frisch *et al.*, 2008], Comet [Van Hentenryck and Michel, 2005] and OPL [Van Hentenryck, 1999] have no, or only limited, support for building libraries of user-defined constraints, and/or are tied to a specific solver.

Automatic transformation is a well-studied topic in constraint programming [Flener *et al.*, 2003; Frisch *et al.*, 2005].By building on MiniZinc, we can benefit from these advancements too. To the best of our knowledge, the application of automatic model transformations to constraint-based mining has not been studied before.

In contrast to MiniZinc, full Zinc compiles a model independent of the instance data (such as the itemset database). Data-independent transformation could be a promising alternative; for example the detection of mining algorithms in Section 3.2 does not require the actual data until the final step; a data-independent transformation could also reduce the incurred overhead further.

6 Conclusions

MiningZinc is based on the declarative *modeling + solving* approach from the field of constraint programming and it promises to bring its benefits to the field of data mining.

Further benefits of the constraint programming methodology include the possible emergence of standard languages and integrated systems for modeling and solving data mining problems, which facilitate the comparison of different algorithms and the re-use of software. It also opens the way to solver selection techniques for data mining. Data mining also raises new challenges for constraint programming as the solutions offered by the *modeling + solving* approach should be competitive with that of standard data mining algorithms. This is non-trivial because data mining algorithms are highly optimized for specific tasks and large datasets, while generic constraint solvers may struggle in particular with the size of the problems.

Acknowledgements. This work was supported by the Research Foundation—Flanders by means of a Postdoc grant and the project "Principles of Patternset Mining" and by the European Commission under the project "Inductive Constraint Programming", contract number FP7-284715, as well as the KU Leuven GOA 13/010 "Declarative Modeling Languages for Machine Learning and Data Mining". NICTA is funded by the Australian Department of Broadband, Communications and the Digital Economy and the Australian Research Council.

References

[Agrawal *et al.*, 1993] Rakesh Agrawal, Tomasz Imielinski, and Arun N. Swami. Mining association rules between sets of items in large databases. In *SIGMOD*, pages 207–216. ACM Press, 1993.

[Blockeel *et al.*, 2012] Hendrik Blockeel, Toon Calders, Élisa Fromont, Bart Goethals, Adriana Prado, and Céline Robardet. An inductive database system based on virtual mining views. *Data Min. Knowl. Discov.*, 24(1):247–287, 2012.

[Bonchi and Lucchese, 2007] Francesco Bonchi and Claudio Lucchese. Extending the state-of-the-art of constraint-based pattern discovery. *Data Knowl. Eng.*, 60(2):377–399, 2007.

[Borgelt, 2012] Christian Borgelt. Frequent item set mining. *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery*, 2(6):437–456, 2012.

[Boulicaut and Jeudy, 2010] J.-F. Boulicaut and B. Jeudy. Constraint-based data mining. In *Data Mining and Knowledge Discovery Handbook*, pages 339–354. 2010.

[Chan *et al.*, 2003] Raymond Chan, Qiang Yang, and Yi-Dong Shen. Mining high utility itemsets. In *ICDM*, pages 19–26, November 2003.

[Flener *et al.*, 2003] Pierre Flener, Justin Pearson, and Magnus Ågren. Introducing ESRA, a relational language for modelling combinatorial problems. In *LOPSTR*, volume 3018 of *LNCS*, pages 214–232. Springer, 2003.

[Frank and Asuncion, 2010] A. Frank and A. Asuncion. UCI machine learning repository, 2010. Available from `http://archive.ics.uci.edu/ml`.

[Frisch *et al.*, 2005] Alan Frisch, Christopher Jefferson, Bernadette Martínez Hernández, and Ian Miguel. The rules of constraint modelling. In *IJCAI*, pages 109–116. Professional Book Center, 2005.

[Frisch *et al.*, 2008] Alan Frisch, Warwick Harvey, Christopher Jefferson, Bernadette M. Hernández, and Ian Miguel. Essence: A constraint language for specifying combinatorial problems. *Constraints*, 13(3):268–306, 2008.

[Fürnkranz and Flach, 2005] Johannes Fürnkranz and Peter A. Flach. ROC 'n' rule learning – towards a better understanding of covering algorithms. *Machine Learning*, 58(1):39–77, 2005.

[Goethals and Zaki, 2004] Bart Goethals and Mohammed J. Zaki. Advances in frequent itemset mining implementations: report on FIMI'03. In *SIGKDD Explorations*, volume 6, pages 109–117, 2004.

[Guns *et al.*, 2011] Tias Guns, Siegfried Nijssen, and Luc De Raedt. Itemset mining: A constraint programming perspective. *Artif. Intell.*, 175(12-13):1951–1983, 2011.

[Guns *et al.*, 2013] Tias Guns, Siegfried Nijssen, and Luc De Raedt. k-pattern set mining under constraints. *IEEE TKDE*, 25(2):402–418, 2013.

[Han and Kamber, 2000] Jiawei Han and Micheline Kamber. *Data Mining: Concepts and Techniques*. Morgan Kaufmann, 2000.

[Henriques *et al.*, 2012] Rui Henriques, Inês Lynce, and Vasco M. Manquinho. On when and how to use SAT to mine frequent itemsets. *CoRR*, abs/1207.6253, 2012.

[Imielinski and Virmani, 1999] Tomasz Imielinski and Aashu Virmani. MSQL: A query language for database mining. *Data Mining and Knowledge Discovery*, 3:373–408, 1999.

[Järvisalo, 2011] Matti Järvisalo. Itemset mining as a challenge application for answer set enumeration. In *Logic Programming and Nonmonotonic Reasoning*, volume 6645 of *LNCS*, pages 304–310. Springer, 2011.

[Malitsky and Sellmann, 2012] Yuri Malitsky and Meinolf Sellmann. Instance-specific algorithm configuration as a method for non-model-based portfolio generation. In *CPAIOR*, pages 244–259, 2012.

[Mannila and Toivonen, 1997] Heikki Mannila and Hannu Toivonen. Levelwise search and borders of theories in knowledge discovery. *Data Min. Knowl. Discov.*, 1(3):241–258, 1997.

[Mannila, 1997] Heikki Mannila. Inductive databases and condensed representations for data mining. In *ILPS*, pages 21–30, 1997.

[Marriott *et al.*, 2008] Kim Marriott, Nicholas Nethercote, Reza Rafeh, Peter J. Stuckey, Maria Garcia De La Banda, and Mark Wallace. The design of the Zinc modelling language. *Constraints*, 13(3):229–267, September 2008.

[Meo *et al.*, 1996] Rosa Meo, Giuseppe Psaila, and Stefano Ceri. A new SQL-like operator for mining association rules. In *VLDB*, pages 122–133, 1996.

[Métivier *et al.*, 2012] Jean-Philippe Métivier, Patrice Boizumault, Bruno Crémilleux, Mehdi Khiari, and Samir Loudni. A constraint language for declarative pattern discovery. SAC '12, pages 119–125. ACM, 2012.

[Nethercote *et al.*, 2007] Nicholas Nethercote, Peter J. Stuckey, Ralph Becket, Sebastian Brand, Gregory J. Duck, and Guido Tack. Minizinc: Towards a standard CP modelling language. In *CP*, volume 4741 of *LNCS*, pages 529–543. Springer, 2007.

[Novak *et al.*, 2009] Petra K. Novak, Nada Lavrac, and Geoffrey I. Webb. Supervised descriptive rule discovery: A unifying survey of contrast set, emerging pattern and subgroup mining. *J. Mach. Learn. Res.*, 10:377–403, 2009.

[Pasquier *et al.*, 1999] Nicolas Pasquier, Yves Bastide, Rafik Taouil, and Lotfi Lakhal. Discovering frequent closed itemsets for association rules. In *Database Theory*, volume 1540 of *LNCS*, pages 398–416. Springer, 1999.

[Schulte *et al.*, 2013] Christian Schulte, Guido Tack, and Mikael Lagerkvist. Gecode, a generic constraint development environment, 2013. `www.gecode.org`.

[Stuckey and Tack, 2013] PeterJ. Stuckey and Guido Tack. Minizinc with functions. In Carla Gomes and Meinolf Sellmann, editors, *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*, volume 7874 of *Lecture Notes in Computer Science*, pages 268–283. Springer Berlin Heidelberg, 2013.

[Uno *et al.*, 2004] Takeaki Uno, Masashi Kiyomi, and Hiroki Arimura. Lcm ver. 2: Efficient mining algorithms for frequent/closed/maximal itemsets. In *FIMI*, volume 126 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2004.

[Van Hentenryck and Michel, 2005] Pascal Van Hentenryck and Laurent Michel. *Constraint-Based Local Search*. MIT Press, 2005.

[Van Hentenryck, 1999] Pascal Van Hentenryck. *The OPL optimization programming language*. MIT Press, 1999.

[Xu *et al.*, 2008] Lin Xu, Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown. SATzilla: Portfolio-based algorithm selection for SAT. *J. Artif. Intell. Res.*, 32:565–606, 2008.