MiningZinc: a Declarative Framework for Constraint-based Mining

Tias Guns[a], Anton Dries[a], Siegfried Nijssen[a,b], Guido Tack[c], Luc De Raedt[a]

[a] *Department of Computer Science, KU Leuven* {*firstname.lastname*}*@cs.kuleuven.be*
[b] *LIACS, Universiteit Leiden s.nijssen@liacs.leidenuniv.nl*
[c] *Faculty of IT, Monash University, Australia and National ICT Australia (NICTA) guido.tack@monash.edu*

Abstract

We introduce MiningZinc, a declarative framework for constraint-based data mining. MiningZinc consists of two key components: a language component and an execution mechanism.

First, the MiningZinc language allows for high-level and natural modeling of mining problems, so that MiningZinc models are similar to the mathematical definitions used in the literature. It is inspired by the Zinc family of languages and systems and supports user-defined constraints and functions.

Secondly, the MiningZinc execution mechanism specifies how to compute solutions for the models. It is solver independent and supports both standard constraint solvers and specialized data mining systems. The high-level problem specification is first translated into a normalized constraint language (FlatZinc). Rewrite rules are then used to add redundant constraints or solve subproblems using specialized data mining algorithms or generic constraint programming solvers. Given a model, different execution strategies are automatically extracted that correspond to different sequences of algorithms to run. Optimized data mining algorithms, specialized processing routines and generic solvers can all be automatically combined.

Thus, the MiningZinc language allows one to model constraint-based itemset mining problems in a solver independent way, and its execution mechanism can automatically chain different algorithms and solvers. This leads to a unique combination of declarative modeling with high-performance solving.

Keywords: Constraint-based mining, Itemset Mining, Constraint Programming, Declarative modeling, Pattern Mining
2010 MSC: 70, 90

1. Introduction

The fields of data mining and constraint programming are amongst the most successful subfields of artificial intelligence. Significant progress in the past few years has resulted in important theoretical insights as well as the development

of effective algorithms, techniques, and systems that have enabled numerous applications in science, society, as well as industry. In recent years, there has been an increased interest in approaches that combine or integrate principles of these two fields [1]. This paper intends to contribute towards bridging this gap.

It is motivated by the observation that the methodologies of constraint programming and data mining are quite different. Constraint programming has focused on a declarative modeling and solving approach of constraint satisfaction and optimisation problems. Here, a problem is specified through a so-called model consisting of the variables of interest and the possible values they can take, the constraints that need to be satisfied, and possibly an optimization function. Solutions are then computed using a general purpose solver on the model. Thus the user specifies what the problem is and the constraint programming system determines how to solve the problem. This can be summarized by the slogan *constraint programming = model + solver(s).*

The declarative constraint programming approach contrasts with the typical procedural approach to data mining. The latter has focussed on handling large and complex datasets that arise in particular applications, often focussing on special-purpose algorithms to specific problems. This typically yields complex code that is not only hard to develop but also to reuse in other applications. Data mining has devoted less attention than constraint programming to the issue of general and generic solution strategies. Today, there is only little support for formalizing 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 [2] further complicates this process, as the problem specification may change between iterations, which may in turn require changes to the algorithms.

The aim of this paper is to contribute to bridging the methodological gap between the fields of data mining and constraint programming by applying the *model + solver* approach to data mining.

In constraint programming, high-level languages such as Zinc [3], Essence [4] and OPL [5] are used to *model* the problem while general purpose *solvers* are used to compute the solutions. Motivated in particular by solver-independent modeling languages, we devise a modeling language for data mining problems that can be expressed as constraint satisfaction or optimisation problems. Furthermore, we contribute an accompanying framework that can infer efficient execution strategies involving both specialized mining systems, and generic constraint solvers. This should contribute to making data mining approaches more flexible and declarative, as it becomes easy to change the model and to reuse existing algorithms and solvers.

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 sets and binary databases, many settings (supervised and unsupervised) and corresponding systems have been proposed in the literature; this makes itemset mining an ideal showcase for a declarative approach to 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 common elements and building blocks can be abstracted away, easing the formulation of existing problems and variations thereof;
- to be *solver-independent*, such that the best execution strategy can be selected for the problem and data at hand. Supported methods should include both *general purpose solvers*, *specialized efficient mining algorithms* and combinations thereof;
- 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 other framework that supports these four design criteria. Especially the combination of user-defined constraints and solver-independence is uncommon (we defer a detailed discussion of related work to Section 6). In the constraint programming community, however, the design of the Zinc [3, 6] family of languages and frameworks is in line with the above criteria. The main question that we answer 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 in the MiniZinc language, to support modeling itemset mining tasks in terms of set operations and constraints;
2. the ability to define the capabilities of generic solvers and specialized algorithms in terms of constraints, where the latter can solve a predefined combination of constraints over input and output variables;
3. a rewrite mechanism that can be used to add redundant constraints and determine the applicability of the defined algorithms and solvers;
4. and automatic composition of execution strategies involving multiple such specialized or generic solving methods.

The language used is MiniZinc [7] version 2.0, extended with a library of functions and constraints tailored for pattern mining. The execution mechanism, however, is much more elaborate than that of standard MiniZinc. For a specific constraint solver, it will translate each constraint individually to a constraint supported by said solver. Our method can automatically compose execution strategies with multiple solvers.

The MiningZinc framework builds on our earlier CP4IM framework [8], which showed the feasibility of constraint programming for pattern mining. This work started from the modeling experience obtained with CP4IM, but the latter

contained none of the above contributions as it was tied to the Gecode solver and consisted of a low-level encoding of the constraints.

The present paper extends our earlier publication on MiningZinc [9] in many respects. It considers the modeling and solving of a wider range of data mining tasks including numeric and probabilistic data, multiple databases and pattern sets. The biggest change is in the execution mechanism, which is no longer restricted to using a single algorithm or generic solver. Instead, it uses rewrite rules to automatically construct execution plans consisting of multiple solver/algorithm components. We also perform a more elaborate evaluation, including a comparison of automatically composed execution strategies on a novel combination of tasks.

Structure of the text. Section 2 introduces modeling in MiningZinc using the basic problem of frequent itemset mining. Section 3 illustrates how a wide range of constraint-based mining problems can be expressed in MiningZinc. In Section 4 the execution mechanism behind MiningZinc is explained, and Section 5 experimentally demonstrates the capabilities of the approach. Section 6 describes related work and Section 7 concludes.

2. Modeling

MiningZinc builds on the MiniZinc modeling language and is hence suitable for data mining problems that can be expressed as constraint satisfaction/enumeration or optimisation problems. We first introduce itemset mining and constraint-based mining. Using frequent itemset mining as an example, we demonstrate how this can be formulated as a constraint satisfaction problem in MiniZinc; and how this relates to the MiningZinc framework.

More advanced problem formulations and related tasks are given in the next section.

2.1. Pattern mining and itemset mining

Pattern mining is a subfield of data mining concerned with finding patterns, regularities, in data. Examples of patterns include products that people often buy together, words that appear frequently in abstracts of papers, recurring combinations of events in log files, common properties in a large number of observations, etcetera. Typical in pattern mining is that the pattern is a *substructure* appearing in the data, so not single words or events but collections thereof; and that there is a measure for the interestingness of a pattern, often based on how frequently it appears in the data.

We will focus on pattern mining problems where the patterns are expressible as sets, also called itemsets. Itemset mining was introduced by Agrawal et al. [10] as a technique to mine customer transaction databases for sets of items (products) that people often buy together. From these, unexpected associations between products can then be discovered.

Since its introduction, itemset mining has been extended in many directions, including more structured types of patterns such as sequences, trees and graphs.

A common issue with pattern mining techniques is that the number of patterns found can be overwhelming. In this respect, there has been much research on the use of constraints to avoid finding uninteresting patterns, on ways of removing redundancy among patterns, as well as different interestingness measures to be used. An overview can be found in a recent book [11].

The input to an itemset mining algorithm 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 $|\phi_{\mathcal{D}}(I)| \geq \mathit{Freq}$, where $\phi_{\mathcal{D}}(I) = \{t | I \subseteq \mathcal{D}(t)\}$.

The set $\phi_{\mathcal{D}}(I)$ is called the *cover* of the itemset. It contains all transaction identifiers for which the itemset is a subset of the respective transaction. The threshold *Freq* is often called the *minimum frequency* threshold. An itemset I which has $|\phi_{\mathcal{D}}(I)| \geq \mathit{Freq}$ is called a *frequent itemset*.

Example 1. Consider a transaction database from a hardware store:

t	$\mathcal{D}(t)$	t	$\mathcal{D}(t)$
1	{Hammer, Nails, Saw}	4	{Nails, Screws, Wood}
2	{Hammer, Nails, Wood}	5	{File, Saw}
3	{File, Saw, Screws, Wood}	6	{Hammer, Nails, Pliers, Wood}

With a minimum frequency threshold of 3, the frequent patterns are: $\emptyset$, {Hammer}, {Nails}, {Hammer,Nails}, {Wood}, {Nails,Wood}.

Constraint-based pattern mining methods can leverage additional constraints during the pattern discovery process; cf. [10, 12, 13]. This has lead to the research topic of *constraint-based itemset mining* [14]. Section 3 will present different constraint-based mining problems in the context of MiningZinc.

2.2. Constraint Programming

Constraint Programming (CP) is a generic method for solving combinatorial constraint satisfaction (and optimisation) problems. It is a declarative method, in that it separates the specification of the problem from the actual search for a solution. On the language side, a number of declarative and convenient languages have been developed. On the solver side, many generic constraint solvers are available, including industrial ones. We refer to the Handbook of Constraint Programming for an extensive overview of technologies and applications [15].

More formally, a Constraint Satisfaction Problem (CSP) is characterized by a declarative specification of constraints over variables.

Definition 2 (Constraint Satisfaction Problem (CSP))**.** A CSP $\mathcal{P} = (\mathcal{V}, \mathcal{D}, \mathcal{C})$ is specified by

- a finite set of variables $\mathcal{V}$;
- a domain $\mathcal{D}$, mapping each variable $V \in \mathcal{V}$ to a set of possible values $\mathcal{D}(V)$;
- a finite set of constraints $\mathcal{C}$.

A variable $V \in \mathcal{V}$ is called *fixed* if $|\mathcal{D}(V)| = 1$. An assignment to the variables $\mathcal{V}$ is a domain $\mathcal{D}$ for which all its variables are fixed. A domain $\mathcal{D}'$ is called *stronger* than a domain $\mathcal{D}$ if $\mathcal{D}'(V) \subseteq \mathcal{D}(V)$ for all $V \in \mathcal{V}$. A *constraint* $C(V_1, \ldots, V_k) \in \mathcal{C}$ is an arbitrary Boolean function on variables $\{V_1, \ldots, V_k\} \subseteq \mathcal{V}$. A solution to a CSP is an assignment to the variables such that all constraints are satisfied, where the domain D' of the assignment must be stronger than D, e.g. in D' each variable V can only be assigned to an element of $D(V)$.

Example 2. Imagine going on a boat trip. There is room to take 2 friends. Of 4 sailing friends, Sjarel and Kaat are better not put on a boat together; Kaat only wants to go if Nora goes; for Raf anything is fine. This can be modelled with a set variable F with domain $\{Sjarel, Kaat, Nora, Raf\}$ and constraints $|F| = 2, \{Sjarel, Kaat\} \nsubseteq F, (Kaat \in F) \rightarrow (Nora \in F)$.

A range of practical modeling languages exist that aid a user in formulating a CSP. Example languages are MiniZinc [3], Essence [4] and OPL [5]. Such languages define variable types, such as Booleans, integers, sets and floats; and define a large number of constraints that can be specified. They typically provide a number of modeling conveniences such as syntactic sugar for accessing an element of an array, for looping over sets (e.g. forall, exists) and for using mathematical-like operators such as sums and products.

2.3. MiniZinc and itemset mining in MiniZinc

We build on the MiniZinc [6] modeling language, version 2.0. A MiniZinc *model* describes a constraint problem as a sequence of expressions, which can include parameter declarations, declarations of decision variables, function and predicate declarations, and constraints. A model describes a parametric problem class, and it is instantiated by providing values for all the parameters, typically in a separate data file. An important feature of MiniZinc is that models are *solver-independent*. They can be translated in a non-parameterized (instantiated) low-level format called FlatZinc that can contain solver-dependent constructs. This format is understood by a wide range of different types of solvers [16], such as CP solvers, MIP (Mixed Integer Linear Programming) solvers, SAT (Boolean Satisfiability) and SMT (SAT-Modulo-Theories) solvers.

The solver reads and interprets the FlatZinc and computes solutions. The compiler achieves the specialization for a particular solver through the use of a solver-specific *library* of predicate declarations. Such a library declares each basic constraint as either a solver *builtin*, which is understood natively by the target solver, or as a decomposition into simpler constraints that are supported by the solver.

Listing 1: "A simple MiniZinc model"

```
1 int: n;
2 array[1..n] of var 1..n: queens;
3 constraint all_different(queens)
4         /\ all_different([queens[i]-i | i in 1..n])
5         /\ all_different([queens[i]+i | i in 1..n]);
6 solve satisfy;
7 output [show(queens)];
```

Listing 2: "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;
6 output [show(Items)];
```

Listing 1 shows a MiniZinc model of the n-Queens problem (the *"Hello World"* of constraint programming). The task is to place n queens on an $n \times n$ chess board so that no two queens attack each other. Line 1 declares n as a parameter of the model. Line 2 declares an array of n decision variables, each corresponding to one row of the chessboard. Each decision variable has domain $1..n$, which represents the column in which the queen in that row is placed (no two queens can be on the same row by definition). The requirement to not attack is implemented by a conjunction (written /\) of three calls to the all_different predicate, which constrain their arguments, arrays of expressions, to be pairwise different. The second and third constraints use array comprehensions as a way of compactly constructing arrays corresponding to the diagonals of the chess board, it is derived from the observation that $Q_i - Q_j \neq i - j$ forbids left-to-right diagonals and $Q_i - Q_j \neq j - i$ forbids right-to-left diagonals. Finally, the solve and output items instruct the solver to find one solution (satisfy) and output the found values for the queens array. Note that a CP solver might declare that it supports the all_different constraint natively in its library, whereas e.g. the library for a MIP solver would define a decomposition into linear inequalities.

Itemset mining in MiniZinc. Pattern mining problems can be modeled directly in MiniZinc. A MiniZinc model of the frequent itemset mining problem is shown in Listing 2. Lines 1 and 2 define the parameters and data that can be provided through a separate data file. The model represents the item and transaction identifiers in $\mathcal{I}$ and $\mathcal{S}$ by natural numbers from 1 to NrI and 1 to NrT respectively.

Listing 3: "Constraint-based mining - cover"

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

The dataset $\mathcal{D}$ is implemented by the array TDB, mapping each transaction identifier to the corresponding set of items. The set of items we are looking for is modeled on line 3 as a *set variable* with an upper bound restricted to the set $\{1,\ldots,$NrI$\}$. The minimum frequency constraint is posted on line 4, which corresponds closely to the formal notation $|\phi_{\mathcal{D}}(I)| \geq Freq$.

The cover function on line 4 corresponds to $\phi_{\mathcal{D}}(I)$. A distinguishing feature of MiniZinc is its support for user defined-predicates, and since version 2.0, user-defined functions [7]. A MiniZinc predicate is a parametric constraint specification that can be instantiated with concrete variables and parameters, like in the call to all_different in Listing 1. A MiniZinc function a generalisation to allow for arbitrary return values.

A declaration of the cover function is shown in Listing 3. Recall that the formal definition of cover is $\phi_{\mathcal{D}}(I) = \{t | I \subseteq \mathcal{D}(t)\}$. The implementation achieves this function by introducing an auxiliary set variable Cover (line 3) and constraining it to contain exactly those transactions that are subsets of Items. The let { ... } in ... construct is used to introduce auxiliary variables and post constraints, before returning a value, in this case the newly introduced Cover, after the in keyword (line 6). Other MiniZinc functions used here include index_set, which returns a set of all the indices of an array (similarly index_set_1of2 returns the index set of the first dimension of a two-dimensional array), and ub, which returns a valid upper bound for a variable. In this particular case, since Cover is a set variable, ub(Cover) returns a fixed set that is guaranteed to be a superset of any valid assignment to the Cover variable. Documentation on MiniZinc's constructs is available online[1].

In the cover() function of Listing 3, the introduced Cover variable is constrained to be equal to the cover (in the let statement, lines 4–5). This constraint states that for all values t in the declared upper bound of Cover, i.e., all values that are *possibly* in Cover, the value t is included in Cover if and only if it is an element of the cover, i.e. the set Items is contained in transaction t. While the implementation of cover is not a verbatim translation of the mathematical definition, MiniZinc enables us to define this abstraction in a library and hide its implementation details from the users.

This example demonstrates the appeal of using a modeling language like

[1] http://www.minizinc.org/2.0/doc-lib/doc.html

Listing 4: "Key abstractions provided by MiningZinc." For brevity we write 'set' for 'set of int' and 'array[]' for 'array[int]'.

```
function var set: cover(var set: Items,
                        array[] of set: TDB);
function var set: cover(var set: Items,
                        array[,] of int: TDB);
function var set: cover_inv(var set: Cover,
                            array[] of set: TDB);

function var int: weighted_sum(array[] of var int: Weights,
                               var set: Items);
function array[] of string: print_itemset(var set: Items);
function array[] of string: print_itemsetWcover(var set: Items,
                                                array[] of set: TDB);

predicate minfreq_redundant(var set: Items,
                            array[] of set: TDB,
                            int: Freq);

function ann: itemset_search(var set: Items);
predicate ann: enumerate;
function ann: query(string db, string sql);
```

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 independent of the actual solution method.

2.4. 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 placed into libraries, to facilitate their reuse in different models. In this way, MiniZinc can be extended to different application domains without the need for developing a new language. The language component of the MiningZinc framework is such a library. Listing 4 lists the signatures of the key functions and predicates provided by the MiningZinc library; we discuss each one in turn.

The two key building blocks of the MiningZinc library are the cover and cover_inv functions. Given a dataset, the cover function determines for an itemset the transaction identifiers that cover it: $\phi_{\mathcal{D}}(I) = \{t | I \subseteq \mathcal{D}(t)\}$ and was already given in Listing 3.

The cover function is also defined over numeric data following the Boolean interpretation: a transaction is covered by an itemset if each item has a non-zero value in that transaction. Listing 5 shows the MiniZinc specification, which uses a helper function to determine the items in a transaction with non-zero value. This can be used together with other constraints on the actual numeric data, as we will show in the following section.

Listing 5: "cover constraint over numeric data"

```
function var set of int: cover(var set of int: Items,
                               array[int,int] of int: DN)
  = let { var set of index_set_1of2(DN): Cover;
      constraint forall (t in ub(Cover))
          ( t in Cover <-> Items subset row(DN,t) )
  } in Cover;

function set of int: row(array[int,int] of int: DN, int: t)
    = {i | i in index_set_2of2(DN) where DN[t,i] != 0}
```

Listing 6: "cover_inv function and the col helper function"

```
function var set of int: cover_inv(var set of int: Cover,
                                   array[int] of set of int: D)
    = let { var set of min(D)..max(D): Items,
        constraint forall(i in ub(Items))
            ( i in Items <-> Cover subset col(D,i) )
    } in Items;

function set of int: col(array[int] of set of int: D, int: i)
    = {t | t in index_set(D) where i in D[t]}
```

Listing 7: "Redundant minimum frequency constraint"

```
predicate minfreq_redundant(var set of int: Items,
                            array[int] of set of int: D,
                            int: Freq)
    = let { var set of int: Cover = cover(Items,D) } in
    forall(i in ub(Items)) (
        i in Items -> card(Cover intersect col(D,i)) >= Freq
    );
```

The cover_inv function computes for a set of transaction identifiers, the items that are common between all transactions identified. Let $\mathcal{D}'$ be the transpose of $\mathcal{D}$, that is, the database that maps items to sets of transaction identifiers. $\mathcal{D}'(i)$ consists of the transactions in which item i appears, that is $\mathcal{D}'(i) = \{t \in \mathcal{S} | i \in \mathcal{D}(t)\}$. cover_inv can now be defined similarly to cover as follows: $\psi_{\mathcal{D}}(T) = \{i | T \subseteq \mathcal{D}'(i)\}$. The MiningZinc specification is similar to that of cover and given in Listing 6; it includes a helper function to calculate $\mathcal{D}'(i)$.

The library includes other helper functions such as weighted_sum and different ways to print item and transaction sets. The itemset_search function defines a *search annotation*, which can be placed on the solve item to specify the heuristic that the solver should use. For MiningZinc, this function can be defined in solver-specific libraries, to enable the use of different search heuristics by different solvers.

Finally, the library also includes predicates that express redundant con-

straints that can be added automatically by the execution mechanism. A redundant constraint is already implied by the model – it does not express an actual restriction of the solution space – but it can potentially improve solver performance, e.g. by contributing additional constraint propagation. A predicate implementing a redundant constraint for minimum frequent itemset mining is shown in Listing 7. It uses the insight that if an itemset must be frequent, then each item must be frequent as well; hence, items that appear in too few transactions can be removed without searching over them. This can be encoded with a constraint that performs *look-ahead* on the items (Listing 7, line 6). See [8] for a more detailed study of this constraint. Another type of redundant information available in the library is a search annotation (Listing 4, line 15). This is an annotation that can be added to the search keyword, and that specifies the order in which to search over the variables. An example of a search order that has been shown to work well for itemset mining is *occurrence* [8]. We also added the *enumerate* search annotation to differentiate, in the model, between satisfaction (one solution) and enumeration (all solutions) problems. The last annotation is the *query* keyword, which can be added to a variable declaration, for example `array [] of set: TDB :: query("mydb.sql", "SELECT tid,item FROM purchases");`. The execution mechanism will automatically typecheck the expression, execute the query and add the data as an assignment to that variable. In this way, one can directly load data from a database, as is common in data mining.

A second instance of the above library exists, with the same signatures, but where all set variables are internally rewritten to arrays of Boolean variables, and all constraints and functions are expressed over these Boolean variables. This alternative formulation can sometimes improve solving performance, and it enables the use of CP solvers that do not support set constraints natively.

The MiningZinc library, without needing too many constructs, offers modeling convenience for specifying itemset mining problems; this will be demonstrated in the next section. Its elements will also be used by the framework described in Section 4 for detecting known mining tasks and when adding redundant constraints.

3. Example problems

Modeling a mining problem in MiningZinc follows the same methodology as modeling a constraint program: one has to express a problem in terms of variables with a domain, and constraints over these variables; for example, a set variable with a minimum frequency constraint over data.

From a data mining perspective, the kind of problem that can be expressed are enumeration or optimization problems that can be formulated using the variable types available in MiniZinc: Booleans, integers, sets and floats, and constraints over these variables. Many itemset mining problems fit this requirement (with the exception of greedy post-processing mechanisms such as Krimp [17] and other incomplete methods). We now illustrate how to model a range of diverse but representative itemset mining problems in MiningZinc.

Listing 8: "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  % Closure
6  constraint Items = cover_inv(cover(Items,TDB),TDB);
7  % Minimum cost
8  array[1..NrI] of int: item_c; int: Cost;
9  constraint sum(i in Items) (item_c[i]) >= Cost;
10 solve satisfy :: enumerate;
```

3.1. Itemset Mining with constraints

Listing 8 contains some examples of constraint-based mining constraints that can be added to the frequent itemset mining model in Listing 2. Line 6, Items = cover_inv(cover(Items,TDB),TDB), represents the popular closure constraint $I = \psi_{\mathcal{D}}(\phi_{\mathcal{D}}(I))$. This closure constraint, together with a minimum frequency constraint, represents the *closed itemset mining* problem [18].

Lines 8/9 represent a common cost-based constraint [13]; 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 other constraints. This includes constraints in propositional logic, for example expressing dependencies between (groups of) items/transactions, or inclusion/exclusion relations between elements. Two more settings involving external data are studied in the next section.

3.2. Itemset mining with additional numeric data

Additional data can be available from different sources, such as quantities of products (items) in purchases (transactions), measurements of elements (items) in physical experiments (transactions) or probabilistic knowledge about item/transaction pairs. We look at two settings in more detail: high utility itemset mining and probabilistic itemset mining in uncertain data.

In high utility mining [19] 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 9 shows the MiningZinc model corresponding to this task. Lines 3 and 4 declare the data, and lines 5 to 8 constrain the utility.

Listing 9: "High Utility itemset mining"

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

Listing 10: "Itemset mining with uncertain data"

```
1 int: NrI; int: NrT; var set of 1..NrI: Items;
2 array [1..NrT, 1..NrI] of float: ProbTDB; float: Expected;
3 constraint
4   sum (t in cover(Items,ProbTDB)) (
5     product (i in Items) (
6       ProbTDB[t,i] ) ) >= Expected;
7 solve satisfy :: enumerate;
```

It uses the cover function over numeric data (Listing 5, checks that a covered item is not 0), as well as the actual data to compute the utility.

Another setting is that of probabilistic itemset mining in uncertain data [20]. Listing 10 shows the MiningZinc model for this task. It is similar to the problem above, with the exception that the data is now real valued (including the numeric cover function), and the constraint (lines 3 to 6) is a sum-product.

3.3. Multiple databases

When dealing with multi-relational data, one can consider each relation as a separate transaction database. We distinguish two different cases: one where the relations are in a star schema, for example, items that are related to different transaction databases, and the more traditional multi-relational setting in which one can identify *chains* of relations.

When dealing with multiple transaction databases over the same set of items, one can impose constraints on each of the databases separately. For example, searching for the itemset with a minimum frequency of α in one database and a maximum frequency of β in another.

A more advanced setting is that of **discriminative itemset mining**. This is the task of, given two databases, finding the itemset whose appearance in the data is strongly correlated to one of the databases. Consider for example a transaction database with fraudulent transactions and non-fraudulent ones, and the task of finding itemsets correlating with fraudulent behavior. Many

Listing 11: "Discriminative itemset mining (accuracy)"

```
1  int: Nrl;
2  array[int] of set of 1..Nrl: D_fraud;
3  array[int] of set of 1..Nrl: D_ok;

4  var set of 1..Nrl: Items;
5  constraint Items = cover_inv(
6                cover(Items,D_fraud),D_fraud);

7  % Optimization function
8  var int: Score = card(cover(Items, D_fraud)) -
9                   card(cover(Items, D_ok));
10 solve maximize Score;
```

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, etc [21].

A discriminative itemset mining task is shown in Listing 11. This is an optimization problem, and the score to optimize is defined on line 8. One could also constrain the score instead of optimizing it; that is, add a threshold on the score and enumerate all patterns that score better. The score is $p - n$ where p is the number of positive transactions covered and n the number of negative ones. Optimizing this score (line 10) corresponds to optimizing the *accuracy* measure; see [22] 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 score). Note how we reuse the cover and cover_inv functions that were also used in Listing 8.

Multi-relational itemset mining consists of the extraction of patterns across multiple relations. Consider a database with authors writing papers on certain topics. A multi-relational mining task would be, for example, to mine for popular related topics, e.g. topics for which more than α authors have written a paper covering all topics. Listing 12 lists the MiningZinc model for this problem. Line 5 constrains the set of papers to those covering all topics, and line 8 states that authors must have at least one such paper.

Note how the existential relation on line 8 can be changed to variations of this setting, for example, requiring that authors have at least β such papers: Authors[a] <-> sum(p in Papers) (AuthorPapers[p]) >= Beta. Other (constraint-based) multi-relational mining tasks [23] can be expressed as well.

3.4. Mining pattern sets

Instead of mining for individual patterns, we can also formulate mining problems over a *set* of patterns. For example, Listing 13 shows the specification of *concept learning* in a k-pattern set mining setting [24]. The goal is to find the k patterns that together best describe the fraudulent transactions, while covering

Listing 12: "Multi-relational itemset mining across Authors Papers and Topics"

```
1  int: NrA; int: NrP; int: NrT; int: Alpha;
2  array[1..NrA] of set of 1..NrP: AuthorPapers;
3  array[1..NrP] of set of 1..NrT: PaperTopics;
4  var set of 1..NrT: Topics;
5  var set of 1..NrP: Papers = cover(Topics, PaperTopics);
6  var set of 1..NrA: Authors;
7  constraint forall(a in 1..NrA) (
8        Authors[a] <-> exists(p in Papers) (AuthorPapers[p]));
9  constraint card(Authors) >= Alpha;
10 solve satisfy :: enumerate;
```

few other transactions. It is very similar to the discriminative itemset mining setting, with the difference that it is assumed that multiple patterns are needed to find good descriptions of the fraudulent transactions.

Other problems studied in the context of k-pattern set mining and n-ary pattern mining [24, 25] can be formulated in a similar high-level way.

Pattern sets can have many symmetric solutions [24], which can slow down search. Symmetry breaking constraints can be added to overcome this, for example by lexicographically ordering the itemsets. MiniZinc offers predicates for lexicographic constraints between arrays, but not between an array of sets. Using the helper function in Listing 14, one could add a symmetry breaking constraint to the concept learning problem of Listing 13 as follows: `constraint lex_less(Items);`.

3.5. Combinations

Using MiningZinc, we can also formulate more complex models. Listing 15 shows an example of such a model where we combine discriminative pattern mining (Listing 11) and high-utility mining (Listing 9).

We can use this model to find itemsets that both have a high utility and discriminate well between positive and negative transactions. To our knowledge, this combined problem has never been studied and no specialized algorithm exists. In Section 5 we show that several strategies exist to solve this model using MiningZinc.

4. MiningZinc execution mechanism

The MiniZinc language used in the previous section is declarative and solver-independent, as we did not impose what kind of algorithm or solving technique must be used. We now discuss how solving is done in MiningZinc.

Figure 1 shows an overview of the overall execution mechanism. The starting point of the process is a MiningZinc model. When using a high-level language like MiniZinc, it is often possible to model a problem in various equivalent ways,

Listing 13: "Concept learning"

```
int: Nrl; int: K;
array[int] of set of 1..Nrl: D_fraud;
array[int] of set of 1..Nrl: D_ok;
array [1..K] of var set of 1..Nrl: Items;
% closed on D_fraud
constraint forall(k in 1..K) (
     Items[k] = cover_inv( cover(Items[k],D_fraud), D_fraud) );
var int: Score = % total fraud - total tok
           card( union(k in 1..K) (cover(Items[k], D_fraud)) )
         - card( union(k in 1..K) (cover(Items[k], D_ok)) );
solve maximize Score;
```

Listing 14: "Lex_less for array of sets (lex_less of two arrays is a MiniZinc global constraint)"

```
predicate lex_less(array[int] of var set of int: S) =
    forall(k in 1..length(S)-1) (
        let { set: elems = union(ub(S[k]),ub(S[k+1])) } in
        lex_less([i in S[k] | i in elems)],
                 [i in S[k+1] | i in elems]) );
```

Listing 15: "High Utility Discriminative itemset mining"

```
int: Nrl; int: NrPos; int: NrNeg; var set of 1..Nrl: Items;
% Transaction data
array[1..NrPos] of set of 1..Nrl: Pos;
array[1..NrNeg] of set of 1..Nrl: Neg;
% Utility data for positives
array [1..Nrl] of int: ItemPrice;
int: MinUtil; int: MinAcc;
array [1..NrPos, 1..Nrl] of int: UPos;
constraint
  sum (t in cover(Items,UTDB)) (
    sum (i in Items) (
      ItemPrice[i]*UTDB[t,i] ) ) >= MinUtil;
constraint Items = cover_inv(cover(Items,Pos),Pos);
constraint card(cover(Items, Pos)) -
           card(cover(Items, Neg)) >= MinAcc;
solve satisfy :: enumerate;
```

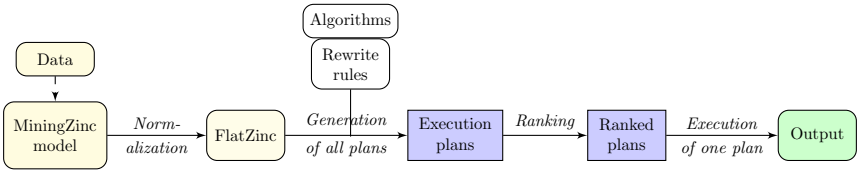

Figure 1: Overview of the MiningZinc toolchain

differing only in syntactic constructs used. The first step in the analysis process is hence to transform this model into a medium-level FlatZinc program (see Section 2.3), which we use as a normalized form for the analysis. This FlatZinc program is not suitable for solving, since it still uses the high-level MiningZinc predicates and functions. Given a set of algorithms and rewrite rules, the FlatZinc program is transformed into all possible sequences of algorithms that can solve the original problem; one such sequence of algorithms is called an execution plan. Multiple execution plans are generated and ranked using a simple heuristic ranking scheme. When a plan is chosen (by the user, or by automatic selection of the highest ranked plan), each of the algorithms in that plan is executed to obtain the required output.

We now describe each of the components in turn.

4.1. Normalization

The purpose of converting to high-level FlatZinc is to enable reasoning over the set of constraints and to simplify the detection of equivalent or overlapping formulations.

FlatZinc is a flattened, normalized representation of a MiniZinc instance (a model and all its data). A MiniZinc instance is transformed into a FlatZinc program by operations such as loop unrolling, introduction of (auxiliary) variables in one global scope, simplifying constraints by removing constants, and rewriting constraints in terms of simple built-ins, where possible. It also performs common subexpression elimination at the global scope: if two identical calls to an expression are present, one will be eliminated. For example, Listing 15 contains twice the function cover(Items,Pos). In the FlatZinc code, only one call X = cover(Items,Pos) will remain, and variable X will be shared by all expressions that contained that call. More details of this procedure are described in [7].

Finally, FlatZinc also supports annotations, for example *enumerate* in Listing 4. During the flattening process, any annotation written in MiniZinc is passed to FlatZinc. Furthermore, any variable mentioned in the **output** statement receives a *output_var* annotation. We will use the concept of output variables versus non-output variables later.

For the purpose of normalization, we use a special version of the MiningZinc library that defines all MiningZinc predicates and functions as builtins, i.e. without giving a definition in terms of simpler expressions. The resulting FlatZinc is therefore not suitable for solving (no solver supports these builtins natively), but it can be analyzed much more easily than the original MiniZinc. It is also

much more compact than the FlatZinc generated for a CP solver, since in many cases the constraints do not need to be unrolled for every row in the dataset.

An important observation is that a FlatZinc program can be seen as a CSP $(\mathcal{V}, \mathcal{D}, \mathcal{C})$, where the possible form of constraints in $\mathcal{C}$ is limited. More specifically, constraints are either of the form:

- $p(X_1, \ldots X_n)$, where $p \in \mathcal{P}$ is a predicate symbol, and each X_i is either a variable in the CSP or a constant. Examples include $\mathit{int\_le}(Y, 1)$, $\mathit{set\_subset}(S, \{2, 4\})$; for notational convenience, in the examples we will represent some of these constraints in infix notation, i.e. $Y \leq 1$, $S \subseteq \{2, 4\}$.
- $(X = f(X_1, \ldots X_n))$, where $f \in \mathcal{F}$ is a function symbol and each X_i is either a variable in the CSP or a constant. Examples are $Y = \mathit{set\_card}(S)$ and $T = \mathit{cover}(I, D)$, where $\mathit{set\_card}(S)$ is a function that calculates the cardinality of the set S, and $\mathit{cover}(I, D)$ is as defined in Section 2. We refer to these constraints as functional definitions.

Consequently, in FlatZinc functions can only occur on the right-hand side of an equality constraint. Function and predicate symbols can be either built-in symbols or user-defined.

Example 3. Consider the problem of finding frequent itemsets on a given dataset, with minimum frequency of 20 and containing at least 3 items (Listing 2 + `constraint card(Items) >= 3;`). We will represent the datasets by $\{\ldots\}$ to indicate that it is a constant. After the automatic flattening process, the FlatZinc model obtained is the following (leaving the domain implicitly defined):

$\mathcal{V} = \{Items, T, SI, ST\}$
$\mathcal{C} = \{T = \mathit{cover}(Items, \{...\}), ST = \mathit{card}(T), SI = \mathit{card}(Items), ST \geq 20, SI \geq 3\}$
$\mathit{annotations} = \{(Items, \text{“output\_var”})\}$

This normalized model then needs to be transformed into an execution plan.

4.2. Generation of all plans

An execution plan specifies which parts of a MiningZinc model are handled by which algorithms or solving techniques. Concretely, an execution plan is a sequence of algorithms that together can handle all the constraints that are present in the model and hence produce the required output.

For example, a possible execution plan for the model of Example 3 could be to run the *LCM* algorithm [26] for frequent itemset mining to generate all itemsets of the given frequency, and then use a general purpose CP solver such as Gecode to filter the solutions further, only allowing itemsets with at least three elements.

This section will first describe the different algorithms that can be made part of MiningZinc, the rewrite rules to construct the plans and finally how all plans are generated using those rules.

4.2.1. Algorithms

We can distinguish two types of algorithms:

- specialized algorithms, which are only capable of solving specific combinations of constraints over input/output variables;
- CP systems, which are capable of solving arbitrary combinations of constraints expressed in FlatZinc.

In an execution plan, the execution of an algorithm will be represented by an atom, consisting of a predicate applied to variables or constants.

Specialized algorithms. These are represented by predicates of fixed arity. Such predicates are declared through mode statements of the kind

$$p(\pm_1 V_1, \ldots, \pm_n V_n),$$

where p is the algorithm name, $\pm_i \in \{+, -\}$ indicates the *mode* of a parameter and V_i is a variable identifier for the parameter.

The interpretation of the modes is as follows:

- the input mode "+" indicates that the algorithm evaluating the predicate can only be run when this parameter is grounded, that is, its value is known;
- the output mode "-" indicates that the algorithm evaluating the predicate will only produce groundings for this parameter.

Example 4. The LCM algorithm for frequent itemset mining is characterized by the mode statement $LCM(+F, +D, -I)$, where F represents a support threshold, D a dataset, and I an itemset. The predicate $LCM(F, D, I)$ is true for any itemset I that is frequent in dataset D under support threshold F. A specific atom expressed using this predicate is $LCM(10, \{\ldots\}, Items)$.

In the MiningZinc framework, each specialised algorithm is registered with the following information:

- its predicate definition $p(\pm_1 V_1, \ldots, \pm_n V_n)$;
- the set of FlatZinc constraints over $V_1, \ldots, V_n$ that specify the problem this algorithm can solve (multiple sets can be given in case of multiple equivalent formulations);
- the binary executable of the algorithm;
- a way to map the input parameters of the predicate (in FlatZinc) to command line arguments and input files for the algorithm;
- a way to map the output of the algorithm to output parameters (in FlatZinc).

Example 5. Version 5 of the LCM algorithm implements the frequent and closed itemset mining problems with additional support for a number of constraints such as constraints on the size of an itemset and the size of the support (minimum and maximum). We can consider each combination of constraints as a different algorithm. For example, the instance of LCMv5 for closed itemset mining with an additional constraint on the minimum size of the itemset can be specified as follows:

predicate `lcm5_closed_minsize( +TDB, +MinFreq, +MinSize, -Items )`

constraints:

- `C = cover(Items,TDB)`
- `S = card(C)`
- `int_le(MinFreq, S)`
- `iC = cover_inv( C, TDB )`
- `set_eq(Items,iC)`
- `Sz = card(Items)`
- `int_le(MinSize, Sz)`

command

```
/path/to/lcm5 C -l MinSize infile(TDB) MinFreq outfile(Items)
```

conversion `infile`, `outfile`: convert between FIMI format and FZN format

In practice we provide syntax for describing multiple instances of the same algorithm in a more succinct way.

CP systems. In contrast to specialized algorithms, CP systems can operate on an arbitrary number of variables and constraints. A predicate representing a CP system can therefore take an arbitrary number of variables as parameter; furthermore, we assume that it is parameterized with a set of constraints.

Example 6. A predicate $Gecode_C(V_1, \ldots, V_n)$ represents the Gecode CP system, where $V_1, \ldots V_n$ are all variables occurring in the constraint set C with which the system is parameterized. A specific atom expressed using this predicate is $Gecode_C(I, T, ST)$, where $C = \{T = cover(I, D), ST = card(T), ST \geq 20\}$; this predicate is true for all combinations of I, T and ST for which the given constraints are true.

A CP system is capable of finding groundings for all variables that are not grounded, and hence there are no mode restrictions on the parameters. Typically, some of the variables occurring in the constraints are not yet grounded, requiring the CP system to search over them. In case all variables are already grounded when calling the CP system, the system only has to *check* whether the constraints are true.

In the MiningZinc framework, each CP system is registered with the following information:

- its predicate name (e.g. *Gecode*);
- the set of FlatZinc constraints it supports, including *global* constraints;
- the binary executable of the CP system;
- optionally, whether set variables must be translated to a Boolean encoding before executing the CP system (more on this in Section 4.2.3).

Execution plans. With the two types of algorithm predicates introduced, we can now define an execution plan as a sequence of atoms over algorithm predicates. Sequences have to be *mode conform*, that is, an algorithm must have its input variables instantiated when it is called.

Example 7. For the model of Example 3 the following is a valid execution plan that uses the LCM and Gecode predicates of Example 4 and 6:

$$[LCM(10, \{\ldots\}, I), Gecode_{(SI=card(I), SI\geq 3)}(I)].$$

The main challenge is now how to transform the initial FlatZinc program into an execution plan. The rewrite rules used to do so are described in the next section.

4.2.2. Rewrite rules

We use rewriting to transform a FlatZinc program into an execution plan. Specifically, we describe a *state* of the rewrite process with a tuple

$$(L, C),$$

where L is an execution plan, and C is a set of constraints and annotations.

The initial state in the rewrite process is $(\emptyset, C)$, where C is the set of all FlatZinc constraints and the empty set indicates the initially empty execution plan; the final state in the rewrite process is $(L, \emptyset)$, where L represents a valid execution plan for the initial set of constraints C, and the empty set indicates that all constraints have been evaluated in the execution plan (modulo the optional *annotations*). Rewrite rules will transform states into other states; an exhaustive search over all possible rewrites will produce all possible execution plans.

A key concept in these rewrite rules are substitutions. Formally, a *substitution* $\theta = \{V_1/t_1, \ldots\}$ is a function that maps variables to either variables or constants. If C is an expression, by $C\theta$ we denote the expression in which all variables V_i have been replaced with their corresponding values t_i according to θ. If for substitution θ it holds that $C\theta \subseteq C'$, the set of constraints C is said to θ *subsume* the set of constraints C'. In the exposition below predicates and variables are untyped for ease of presentation. FlatZinc is a typed language, so in practice we only allow variables of the same type to be mapped to each other.

We now define three types of rewrite rules: rules for adding redundant constraints, for executing specialized algorithms and for executing CP systems.

Rules for redundant constraints. Let C_1 and C_2 both be sets of constraints over the same variables, let $C_1 \rightarrow C_2$ hold (C_1 entails C_2). Since C_2 will be true whenever C_1 is, the set of constraints C_2 can be added as redundant constraints to any $C \supseteq C_1$. Taking substitutions into account, we have the following rewrite rule:

IF $C_1\theta$ subsumes the set of constraints C,
THEN $(L, C) \vdash (L, C \cup C_2\theta)$.

Example 8. Past work showed that the execution of the frequent itemset mining task is more efficient in some CP systems if a look-ahead constraint is added. Let $C_2 = \{\mathit{minfreq\_redundant}(I, D, V)\}$ represent this look-ahead constraint (see Section 2.4). This constraint set is entailed by the set of constraints $C_1 = \{A = \mathit{cover}(I, D), B = \mathit{card}(A), V \leq B\}$. Then for the model of Example 3 (depicted by C_M) we have the following rewrite:

$$(\emptyset, C_M) \vdash (\emptyset, C_M \cup \{\mathit{minfreq\_redundancy}(I, \{\ldots\}, 20)\}).$$

Redundant constraints are registered in the system with the following information:

- a set of constraints C_1;
- a function that takes as input the substitution θ and produces a constraint set C_2 over the variables in θ as output.

Rules for specialized algorithms. Recall that all specialised algorithms are registered with a predicate definition $p(\pm_1 V_1, \ldots, \pm_n V_n)$ and a set of constraints C that define the problem being solved by this algorithm. Note that not all variables in C need to be a parameter of the predicate; there can be *auxiliary* variables.

Let (L, C_M) be a state in the rewriting of an execution plan. The key idea is that if the set of constraints C of an algorithm subsumes the given set of constraints C_M, then we wish to append its predicate (p) to the execution plan L. More formally, if L is the current plan, and C subsumes C_M with substitution θ, we can add $p(V_1\theta, \ldots, V_n\theta)$ to L.

Example 9. If our model has constraints $\{T = \mathit{cover}(I, \{\ldots\}), ST = \mathit{card}(T), SI = \mathit{card}(I), ST \geq 20, SI \geq 3\}$, and our current execution plan is empty ($\emptyset$); LCM's constraint set $\{T' = \mathit{cover}(I', D'), ST' = \mathit{card}(T'), ST' \geq V'\}$ subsumes the model with the substitution $\{T' \mapsto T, I' \mapsto I, D' \mapsto \{\ldots\}, V' \mapsto 20\}$. Hence, we may add $\mathit{LCM}(20, \{\ldots\}, I)$ to the execution plan.

Removing subsumed constraints from C_M. The next important step is to remove as many subsumed constraints as possible from C_M, to avoid them being unnecessarily recomputed or verified again. Indeed, running the algorithm will ensure that these constraints are satisfied, but unfortunately, we cannot always remove all subsumed constraints.

Example 10. If our model has constraints $\{T = cover(I, \{\ldots\}), ST = card(T), ST \geq 20, ST \leq 40\}$, and we again use the LCM algorithm to solve part of this model, we cannot remove the constraints $ST = card(T)$ and $T = cover(I, D)$, even though they are subsumed; the reason is that the constraint $ST \leq 40$, which is not subsumed, requires the ST value, which is not in the output of the LCM algorithm.

This problem is caused by auxiliary variables, which occur in the constraint definition of the algorithm but not in the mode definition.

When $C\theta$ is the set of constraints subsumed by the algorithm, the set $C_M \backslash C\theta$ contains all remaining constraints; if among these remaining constraints there are constraints that rely on the functionally defined by variables that are not in the mode definition of the algorithm, we can remove all constraints except the functional definition constraints necessary to calculate these variables.

More precisely, assume we are given a state (L', C') for a FlatZinc program with constraints $\mathcal{C}$ and output variables $\mathcal{V}_{\texttt{output}}$ (that is, variables in $\mathcal{V}$ that have a *output_var* annotation). Then let $V_{(L',C')}$ be the set of those variables occurring in C' or in $\mathcal{V}_{\texttt{output}}$ that do not occur in L', e.g. the free variables that will still be used later in the execution plan. Let $F(L', C', \mathcal{C}) = (L', C' \cup \{c \in \mathcal{C} \mid c \text{ functionally defines a variable in } V_{(L',C')}\})$, i.e., this function adds the functional definitions for variables for which the definitions are missing from C'. The inclusion of missing definitions may trigger a need to include further definitions; the repeated application of this function will yield a fixed point $F^*(L', C', \mathcal{C})$.

This function can be used to define the following rewrite rule:

IF	$C\theta$ subsumes constraints C_M for mode declaration $p(\pm_1 V_1, \ldots, \pm_n V_n)$, and $[L, p(V_1\theta, \ldots, V_n\theta)]$ such that the substitution is conform to the modes of p,
THEN	$(L, C_M) \vdash F^*([L, p(V_1\theta, \ldots, V_n\theta)], C_M \backslash C\theta, C_M)$.

Continuing on Example 10, we can see that the functional definition constraints for the variables ST and T that are still used will not be removed. The constraint $ST \geq 20$ will be removed though.

Examples of specialized algorithms. The following are additional examples of rewriting for specific algorithms available in MiningZinc.

Example 11. Let $calcfreq(+I, -F, +D)$ be a specialized algorithm that calculates the frequency F of an itemset I in a database D. The constraint set corresponding to this algorithm is $\{T = cover(I, D), F = card(T)\}$. Assume we have the following state:

$$([LCM(20, D, I)], \{T = cover(I, D), F = card(T), F \leq 40\}),$$

then with the empty substitution the constraint set $\{T = cover(I, D), F = card(T)\}$ subsumes the model. Furthermore, as the variable F is in the output

of the algorithm $calcfreq(I, F, D)$, and the variable T is not used outside the subsumed set of constraints, we can remove these functional definition constraints. Hence, we can rewrite this state to

$$([LCM(20, D, I), calcfreq(I, F, D)], \{F \leq 40\}),$$

Example 12. Let $maxsup(+I, +V, +D)$ be a specialized algorithm that determines whether the frequency of an itemset I in a database D is lower than a given threshold V, i.e., the algorithm checks a constraint and has no output. The constraint set corresponding to this algorithm is $\{T = cover(I, D), F = card(T), F \leq V\}$. Assume we have the following state:

$$([LCM(20, D, I)], \{T = cover(I, D), F = card(T), F \leq 40\}),$$

Then we can rewrite this state into:

$$([LCM(20, D, I), maxsup(I, 40, D)], \emptyset),$$

where every solution found by LCM will be checked by the specialized *maxsup* algorithm.

Rules for CP systems. The final rewrite rule is the one for the registered CP systems. We use the following rewrite rule for a state (L, C):

IF	*cp* is a CP system and all constraints in C are supported by CP system *cp*
THEN	let $V_1, \ldots V_n$ be the variables occurring in C, $(L, C) \vdash ([L, cp_C(V_1, \ldots, V_n)], \emptyset)$.

Currently, a CP system will always solve all of the remaining constraints. An alternative rule could be one in which only a subset of the remaining constraints is selected for processing by a CP system; this would enable more diverse combinations where specialized algorithms are used after CP systems. However, for reasons of simplicity and by lack of practical need, we do not consider this option further.

Translating set variables to Boolean variables. MiningZinc models are typically expressed over set variables, however, some CP solvers do not support constraints over set variables. In previous work, we found that solvers that do support set variables are usually more efficient on a Boolean encoding of the set variables and constraints (for example, constraining the cardinality of a subset of a variable requires 2 constraints when expressed over sets, yet only 1 linear constraint in the Boolean encoding).

Hence, for CP solvers we provide a transformation that translates all set variables into arrays of Boolean variables. For each potential value in the original set, we introduce a Boolean variable that represents whether that value is included in the set or not. Constraints over these set variables are translated accordingly, e.g. replacing a subset constraint by implications between

every pair of corresponding Boolean variables. This Boolean transformation is done directly on the FlatZinc representation, and transparently to the execution mechanism.

When registering a CP system in the MiningZinc framework, one can hence indicate whether set variables must be translated to Booleans just before execution of the CP system. For systems that support set variables we typically register two system predicates, one without and one with the Boolean transformation flag.

4.2.3. Generation of all plans

So far we have focussed on individual rewrite rules and how they can be used to rewrite a set of constraints C and possibly add a step to an execution plan L. We now show how different rewrite rules can be combined to create complete execution plans.

As mentioned before, sequences of execution steps have to be *mode conform*. More specifically, for each parameter of an atom the following needs to hold:

- *input conform*: when the parameter has an input mode, it must either be ground or instantiated with a variable that has an output mode in an earlier atom in the sequence;
- *output conform*: when the parameter has an output mode, it must be instantiated with a variable that does not have an output mode in any earlier atom in the sequence.

The search for all execution plans operates in a depth-first manner. In each node of the search tree, the conditions of all rewrite rules are checked (including mode conformity). Rules with substitutions that are identical to a rule applied in one of the parents of the node are ignored. The search then branches over each of the applicable rules. This continues until no more rules are applicable. If at that point the set of constraints C in the state (L, C) is empty (modulo annotations), then L is a valid execution plan.

In practice, as rules for redundant constraint can only add constraints in our framework, we can restrict them to only be considered if the current plan L is empty. Furthermore, as rules for specialized algorithms can only remove constraints, if such a rule is not applicable in a node of the search tree it must not be considered for any of the descendant nodes either.

One can observe that in the presence of rewrite rules for redundant constraints, this process is not guaranteed to terminate for all sets of rewrite rules. One could use a bound on the depth of search. Currently, we work under the simplified assumption that the rewrite rules provided to the system by the user do not lead to an infinite rewrite process. This assumption holds for the examples used in this article.

4.3. Ranking plans

In the previous step, all possible execution plans are enumerated, leaving the choice of which execution plan to choose open to the user.

In relational databases, a query optimizer attempts to select the most efficient execution plan from all query plans. Typically, a cost (e.g. number of tuples produced) is calculated for each step in the plan, and the plan with overall smallest cost is selected [27].

In MiningZinc, this is more complicated as we are dealing with combinatorial problems, for which computing or estimating the number of solutions is a hard problem in itself. Furthermore, different algorithms have different strengths and weaknesses, leading to varying runtimes depending on the size and properties of the input data at hand. This has been studied in the algorithm selection and portfolio literature [28].

In MiningZinc this is further complicated by having chains of algorithms. A MiningZinc formulation can lead to new execution plans that have never been observed before, complicating an approach in which each plan is treated as one meta-algorithm for which we could learn the performance. Additionally, different chaining of algorithms can again lead to differences in runtime, depending on the data generated by the previous algorithms. Nevertheless, the input to the next algorithm in a chain is not known until all its predecessor algorithms are run.

The purpose of this paper is not to solve this hard problem. However, MiningZinc is built around the idea that specialized algorithms should be used whenever this will be more efficient than generic systems. Hence, we can discriminate between three types of execution plans:

1. *Specialized plans*: plans consisting of only specialized algorithms
2. *Hybrid plans*: plans consisting of a mix of both specialized and generic CP systems
3. *Generic plans*: plans consisting of only generic CP systems.

We hence propose a heuristic approach to ranking that assumes specialized plans are always preferred over hybrid ones, and that hybrid ones are preferred over generic plans. Once all plans are categorized in one of these groups, we can rank the plans within each group (an example is provided below).

For *specialized plans* we adopt the simple heuristic that plans with fewer algorithms are to be preferred over plans with more. The idea is that with fewer algorithms, probably more of the constraints are pushed into the respective algorithms. Ties in this ranking are typically caused by having multiple algorithms that solve the same problem (e.g. frequent itemset mining). One could use an algorithm selection approach for choosing the plan with the best 'first' algorithm in the chain. We did not investigate this further; instead we assume a global ordering over all algorithms (e.g. the order in which they are registered in the system), and break ties based on this order.

Hybrid plans are first ordered by number of constraints handled by generic systems (fewer is better), then by number of algorithms (fewer is better), and finally we break ties using the global order of the algorithms. Choosing plans with fewer CP constraints first will prefer solutions where specialized algorithms solve a larger part of the problem, but it also penalizes the use of redundant constraints unfortunately. Note that we assume that there is also a global

order of all CP systems (for example, based on the latest MiniZinc competition results).

Finally, *generic plans* consist of one CP system that solves the entire problem. Differences in this category come from the use of different redundant constraints and different CP systems. As this involves only one CP system, one could very well apply algorithm selection techniques here. As the ranking of generic plans is only important in case there are no specialized or hybrid plans, we rather use a simple ranking, first based on number of constraints (with the naive assumption that more redundant constraints are better), then on the global order of the CP systems.

Example 13. Assume we wish to solve the earlier model

$$\{T = cover(I, D), S = card(T), 20 \leq S, S \leq 40\},$$

where the available algorithms are the *LCM* and *FPGrowth* specialized itemset mining algorithms, the *maxsup* and *frequency* specialized algorithms and the *Gecode* generic CP system (in that order). Then these are the ranked execution plans:

Specialized:
  $[LCM(20, D, I), maxsup(I, 40, D)]$
  $[FPGrowth(20, D, I), maxsup(I, 40, D)]$
Hybrid:
  $[LCM(20, D, I), frequency(I, S, D), Gecode_{(S\leq 40)}(S)]$
  $[FPGrowth(20, D, I), frequency(I, S, D), Gecode_{(S\leq 40)}(S)]$
  $[LCM(20, D, I), Gecode_{(T=cover(I,D),S=card(T),S\leq 40)}(I, S, T)]$
  $[FPGrowth(20, D, I), Gecode_{(T=cover(I,D),S=card(T),S\leq 40)}(I, S, T)]$
Generic:
  $[Gecode_{(T=cover(I,D),S=card(T),20\leq S,S\leq 40,minfreq\_redundant(I,D,20)}(I, S, T)]$
  $[Gecode_{(T=cover(I,D),S=card(T),20\leq S,S\leq 40)}(I, S, T)]$

In the above we assume that LCM and FPGrowth are aware of the redundant constraint *minfreq_redundant*. If not, there would be variants of each of the specialized and hybrid strategies with redundant constraints too (they would be ranked below their non-redundant equivalent as they would have more constraints for the CP system).

Note that we proposed just one heuristic way of ordering the strategies, based on common sense principles. In the experiments, we will investigate the difference in runtime of the different strategies in more detail.

4.4. Execution of a plan

One plan is executed in a similar way as a Prolog query. The execution proceeds left-to-right. If all variables are ground then the algorithm simply checks whether the current grounding (assignment to variables) satisfies the inherent

constraints of the algorithm, and if so outputs the same grounding. Otherwise, the algorithm is used to find all groundings for non-grounded variables. Each grounding will be passed in turn to the next algorithm. The evaluation backtracks until all groundings for all predicates have been evaluated.

Note that before executing a specialized algorithm, the accompanying mapping from ground FlatZinc variables to input files and command line arguments is applied. After execution, the mapping from output of the algorithm to (previously non ground) FlatZinc variables is also performed.

Example 14. In the execution plan of Example 7, $[LCM(10, \{\ldots\}, I)$, $Gecode_{(SI=card(I),SI\geq 3)}(I)]$, the database $\{\ldots\}$ is transformed into a LCM's file format and the minimum frequency threshold 10 is given as argument to the LCM executable. LCM then searches for all groundings of the I variable, that is, all frequent itemsets. Each such itemset is processed using the Gecode system; variable I is already grounded so it will simply check the constraints $(SI = card(I), SI \geq 3)$ for each of the giving groundings of I to a specific itemset. All assignments to the I variable that satisfy all constraints hence constitute the output of the execution plan.

5. Experiments

In the experiments we make use of the ability of MiningZinc to enumerate all execution strategies, and to compare the different strategies that MiningZinc supports. We focus on the following main questions: 1) what is the computational overhead of MiningZinc's model analysis and execution plan generation; 2) what are the strengths and weaknesses of the different solving strategies?

The MiningZinc framework is implemented in Python with key components, such as `libminizinc`[2] for the MiniZinc to FlatZinc conversion, written in C++. Our implementation supports multiple algorithms for executing parts of a model. All CP solvers have a corresponding rewrite rule, and the specialized algorithms have one rewrite rule for each task they support. We also use one rewrite rule for redundant constraints in case of a minimum frequency constraint. The constraint solvers used are Gecode [29], Opturion's CPX [30], Google or-tools [31] and the g12 solvers from the MiniZinc 1.6 distribution [6]. We also provide a custom version of Gecode for fast checking of given solutions against a FlatZinc model. We use this solver as our default solver (it is the highest ranked solver) in case a generic CP system is needed merely for constraint checking. The constraint-based mining algorithms are LCM version 2 and 5 [26] and Christian Borgelt's implementations of Apriori (v5.73), Eclat (v3.74) and FPGrowth (v4.48) [32]; these are the state-of-the art for efficient constraint-based mining. In our experiments we also used the HUIMine algorithm [33] for high utility mining as found in the SPMF framework [34]. For correlated itemset mining, the corrmine algorithm is used [35]. Input/output

[2] http://www.minizinc.org/2.0/

mapping for these algorithms is written in Python, as are specialized checking algorithms like *calcfreq* and *maxsup*.

The datasets are from the UCI Machine Learning repository [36][3] and from the FIMI repository [37]. Experiments were executed on Linux computers with quad-core Intel i7 processors. Unless stated otherwise we used a timeout of 900 seconds, and a limit on memory usage of 6Gb. The MiningZinc system and datasets used can be downloaded at `http://dtai.cs.kuleuven.be/CP4IM/miningzinc/` *the version described in this work will be made available upon acceptance of the paper.*

5.1. Computing all execution strategies

An important part of MiningZinc consists of analyzing a given model and determining all available execution plans. In our first experiment we focus on this part of the execution and we analyze the time needed for this process.

Table 1 shows, for increasingly large datasets, the time of (1) normalizing a model (+ data) to the intermediate FlatZinc representation and (2) generating and ranking all available execution plans. The models used are combinations of the constraints shown in Listing 2 (Freq/Fr) and Listing 8 (Clo+MinCost). Solving times are not shown as they depend on the threshold supplied (and typically range from seconds to hours, see the following sections).

The table shows that the normalization is quick for small datasets, but can take some seconds for large datasets. In fact, most time is spent on reading the data; the *libminizinc* tool uses the standard MiniZinc parser to read data from quite verbose text files, it is not optimized for parsing large matrices of data. This can be sidestepped by loading in the data directly from a database with the *query* annotation.

The actual plan generation time is rather low but can increase with the size of the data and the complexity of the task. The key part here is the subsumption check of the rewrite rules, which, in case of complex constraint networks over large datasets, can take a bit of time.

5.2. CP Solver performance

MiningZinc has the ability to automatically add redundant constraints or transform set variables to Boolean variables. From earlier work [8], we know that such reformulations can improve the runtime behavior of solvers.

The solver-independence of MiningZinc allows for an easy comparison of different solvers and reformulations. We compare the different reformulations on three state-of-the-art CP solvers that won medals in the 2013 MiniZinc challenge: Gecode, Opturion's CPX, Google's or-tools. The comparison is done on a range of datasets for the standard mining tasks of frequent itemset mining and closed itemset mining. Or-tools does not support constraints over set variables and hence requires the set to Boolean transformation.

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

Dataset	#Tr	#It	%D	Freq	Fr+Clo	Fr+Clo+ Mincost
zoo-1	101	36	44%	0.052/0.007	0.052/0.009	0.094/0.010
primary-tumor	336	31	48%	0.069/0.007	0.069/0.009	0.109/0.011
soybean	630	50	32%	0.098/0.008	0.091/0.010	0.131/0.012
german-credit	1000	110	35%	0.202/0.012	0.195/0.014	0.229/0.015
hypothyroid	3247	86	50%	0.578/0.024	0.578/0.027	0.594/0.027
mushroom	8124	112	19%	0.780/0.032	0.800/0.036	0.799/0.035
pumsb_star	49046	2088	2.4%	13.489/0.546	13.895/0.563	13.459/0.541
retail	88162	16470	0.1%	5.132/0.235	5.290/0.243	5.086/0.233
T10I4D100K	100000	870	1.2%	5.692/0.262	5.821/0.271	5.633/0.259
T40I10D100K	100000	942	4.2%	16.285/0.746	16.707/0.763	16.057/0.723

Table 1: Left, dataset statistics: #Tr=nr. of transactions, #It = nr. of items, %D = density. Right, time taken by MiningZinc analysis for 3 different tasks; normalization to FlatZinc/execution plan generation, in seconds.

	Gecode				CPX				or-tools	
	set	set/red	bool	bool/red	set	set/red	bool	bool/red	bool	bool/red
soybean	2.65	4.02	2.04	**2.01**	**0.58**	3.20	3.08	4.62	1.89	**1.55**
primary-tumor	2.14	11.61	**1.30**	1.61	**2.65**	33.20	9.66	6.48	**1.07**	1.57
vote	3.32	13.48	2.34	**2.24**	**4.16**	39.40	20.33	7.28	2.05	**2.01**
lymph	**92.85**	220.76	216.53	211.36	**110.85**	241.37	216.18	222.01	211.25	**205.22**
mushroom	580.67	308.30	604.41	**81.52**	**46.42**	239.37	258.88	259.02	257.27	**90.31**
hepatitis	386.88	446.16	385.57	**376.07**	415.24	573.33	478.14	**402.42**	381.10	**374.99**
german-credit	523.58	582.50	446.38	**310.30**	**384.35**	587.66	422.15	471.99	401.28	**366.82**
heart-cleveland	568.66	595.44	482.94	**425.74**	**544.58**	693.37	556.37	561.27	456.69	**428.64**
australian-credit	755.83	757.56	674.09	**580.25**	**565.78**	761.22	642.44	735.72	643.24	**585.40**

Table 2: Frequent itemset mining, runtime in seconds averaged over different thresholds. */red* indicates redundant constraints were added. Lowest average runtime per solver in bold.

	Gecode				CPX				or-tools	
	set	set/red	bool	bool/red	set	set/red	bool	bool/red	bool	bool/red
soybean	**0.62**	1.54	2.36	2.25	**0.54**	1.92	4.06	3.98	**2.16**	2.40
primary-tumor	**1.86**	9.15	3.20	3.31	**1.94**	22.92	8.97	7.75	**3.27**	3.69
vote	**2.85**	11.67	5.67	5.50	**3.42**	30.67	23.02	10.73	**5.76**	5.97
lymph	**1.90**	11.68	4.50	3.59	**5.31**	70.16	8.16	7.63	**3.55**	3.86
mushroom	177.92	138.42	432.67	**127.03**	**21.82**	59.20	273.64	168.14	**111.87**	135.79
hepatitis	**78.58**	312.16	118.29	106.23	384.87	559.34	405.83	**247.12**	**97.59**	129.55
german-credit	273.61	443.12	332.36	**260.93**	**353.88**	564.18	446.07	415.82	**291.00**	312.55
heart-cleveland	364.57	464.67	380.53	**293.25**	427.42	599.48	556.28	**410.23**	**315.58**	317.50
australian-credit	489.28	610.58	509.54	**422.79**	**551.05**	738.46	618.00	579.33	**444.50**	455.61

Table 3: Closed frequent itemset mining, runtime in seconds averaged over different thresholds. */red* indicates redundant constraints were added. Lowest average runtime per solver in bold.

Table 2 shows average runtimes for the different reformulations. One can immediately see that the type of reformulation to use can depend on the solver used; for Gecode and or-tools, using Boolean variables with redundant constraints yields lower average runtimes, while for CPX not using any reformulation is often fastest. CPX uses a lazy clause generation technique that includes its own lazy transformation from set to Boolean variables.

Looking in more detail at the difference between adding or not adding redundant constraints, we can see that in case of set variables adding these constraints slows down the process. The redundant constraints added compute a subset of the itemset for every transaction, which requires each time that an auxiliary set variable is created. This overhead seems to overshadow the potential gain in propagation. This is not the case when using the Boolean transformation; no subset variables need to be created but instead a slice of the Boolean array representing the itemset is directly used.

Table 3 show the results for closed itemset mining. Interestingly, for closed itemset mining or-tools consistently performs faster without the redundant constraints while Gecode performs faster with them for most of the larger datasets.

As is known in constraint programming, reformulation and adding redundant constraints may or may not be beneficial, depending on the problem and instance at hand. This may depend on the solver used as well, as we observed for a number of typical mining problems. These experiments suggest the potential of algorithm selection techniques to help the user in choosing the best generic solver for a problem.

In the next section, we will compare the different solvers with each other and with specialized mining algorithms.

5.3. Standard tasks

On well-studied data mining tasks, one can expect specialized mining algorithms to be more efficient than generic CP solvers. Indeed, this is also the case for frequent and closed itemset mining.

Figure 2 shows a number of representative datasets with results for Gecode, or-tools and CPX compared to mining algorithms Apriori, Eclat and LCMv2. One can observe the gap in runtime between the CP solvers and mining algorithms. Among the solvers, CPX is faster for high threshold values, but its

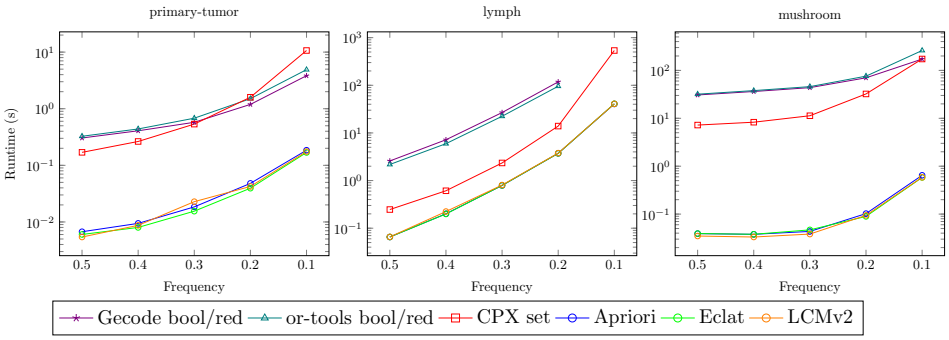

Figure 2: Comparison of single algorithm strategies for frequent itemset mining.

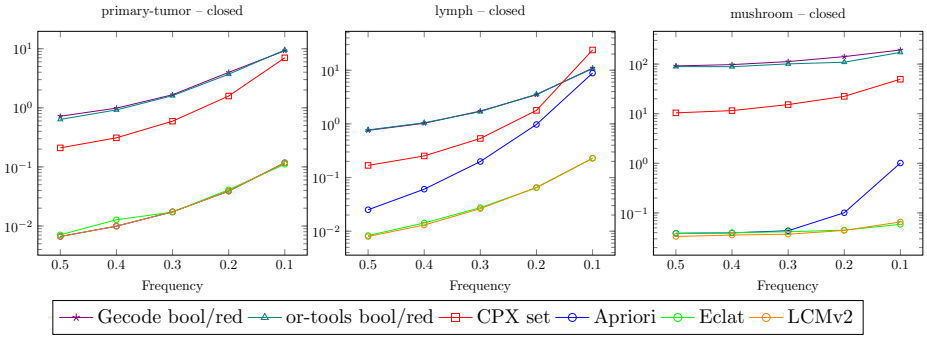

Figure 3: Comparison of single algorithm strategies for closed frequent itemset mining.

runtime grows somewhat quicker than the other solvers for lower values. The miners used are all highly optimized and there is little difference in their runtime to be noticed.

Figure 3 plots a comparison for the task of closed frequent itemset mining. Again the gap between solvers and miners can be observed, as can CPXs faster growth for lower thresholds. Note that the traditional Apriori algorithm performs gradually worse than the depth-first Eclat and LCM algorithms for certain datasets.

While MiningZinc is a convenient tool to perform comparisons between algorithms, its novelty lies in its ability to combine different algorithms in an automatic way. This is experimentally investigated in the next two sections.

5.4. Variations of standard tasks

In the previous experiments we compared the behavior of different execution plans that consisted of a single stage. Now we focus on execution plans that consist of multiple stages.

First, we address the problem of finding all closed itemsets that also satisfy a minimum size constraint on the size of the itemset.

Figure 4 shows a comparison of 5 approaches to solve this problem:

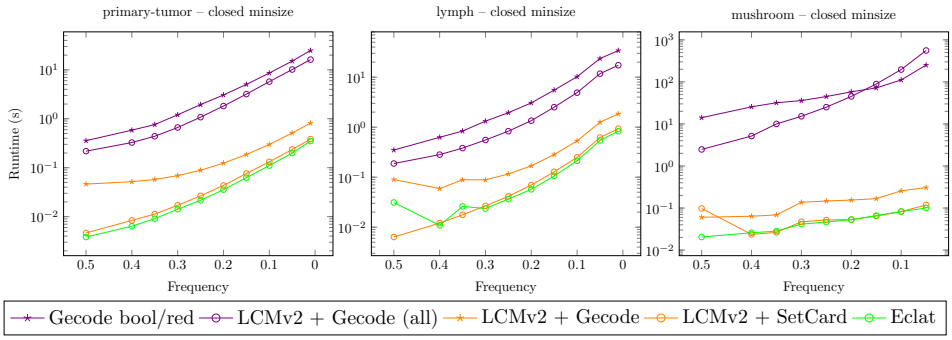

Figure 4: Comparison of hybrid solve strategies for closed frequent itemset mining with a size constraint (size ≥ 4).

1. Using the CP solver Gecode (with redundant constraints and Boolean encoding).
2. Using the specialized algorithm LCMv2, followed by a Gecode-based checking CP system that checks *all* constraints in the model.
3. Using the specialized algorithm LCMv2, followed by a Gecode-based checking CP system that only checks constraints that have not been checked before (i.e. the size constraint).
4. Using the specialized algorithm LCMv2, followed by a specialized algorithm for post-processing the set cardinality constraints.
5. Using the specialized algorithm Eclat, which is capable of solving the complete problem.

Across all datasets, we can observe the following trends.

Unsurprisingly, the specialized algorithm is usually the fastest, followed closely by the specialized post-processor. The difference is negligible in most cases, and in a few cases the specialized post-processor is faster.

In most cases, a specialized algorithm followed by a generic CP checker is faster than the pure CP approach, especially when already satisfied constraints are removed.

We experimented with several larger datasets than mentioned here, for instance, the 'accidents' dataset available at the FIMI repository[4]. For these datasets the pure CP approach does not work due to the complexity of flattening the model and data, while the post-processing based approaches are still able to solve the problem.

Hence, even for large datasets for which a CP-based approach is not possible, MiningZinc can still be used and can even continue to take advantage of generic CP technology in the post-processing step (in case only a subset of the constraints need checking).

[4] http://fimi.ua.ac.be/

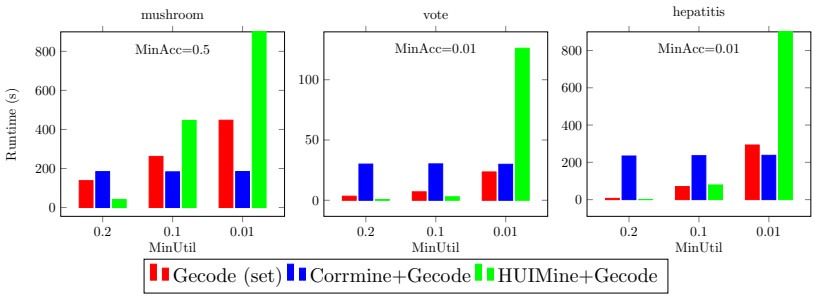

Figure 5: Comparison of runtimes of different execution plans on a selection of representative datasets.

5.5. Combinations of tasks

In the previous experiments we focussed mainly on solving models for which a specialized algorithm exists. In order to analyze the power of MiningZinc, we now focus on a more complex model that involves solving two well-studied problems from constraint-based mining: high-utility mining and discriminative pattern mining. This combined problem was introduced in Section 3, Listing 15.

MiningZinc provides us with three strategies for solving this model: (1) use a CP solver to solve the entire model, (2) use a specialized algorithm for mining discriminative itemsets and post-process its results using a CP solver, and (3) use a specialized high-utility mining algorithm and post-process its result using a CP solver.

For the data we used a selection of UCI datasets as before, augmented with randomly generated utilities. These utilities were generated using the procedure described in [33], that is, the item utilities were generated using a lognormal distribution ($\mu = 0, \sigma = 1$), and the amounts were generated uniformly between 1 and 10.

Results are shown in Figure 5 for three datasets. In general, there is no single best strategy. However, the strategy with the high utility mining algorithm typically performs worse than the other two. This is not surprising because (1) there are typically much more high utility itemsets than there are discriminative ones, and (2) the remaining constraints for the high-utility strategy are computationally harder to verify since they include calculating the cover set of the itemset, whereas the other case only requires computing a simple cost function. Overall, the direct CP solver approach offers a trade-off between the two other strategies.

To conclude, choosing the best strategy for solving complex models is a non-trivial task and for this problem it depends on the number of solutions to the subproblem we are solving first. In many settings, using the most restrictive algorithm first (in this case discriminative pattern mining) will be the best choice. When little is known about the expected output of the subproblems, the pure CP approach can offer a good trade-off for this problem.

6. Related Work

In the data mining field, our work is related to that on inductive databases [38]; these are databases in which both data and patterns are first-class citizens and can be queried. Most inductive query languages, e.g., [39, 40], 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 [41], which provides lazy access to patterns through a virtual table. Standard SQL can be used for querying, and the implementation will only materialize those patterns in the table that are relevant for the query. This is achieved using a traditional mining algorithm.

Work on constraint solving for itemset mining [8, 42] 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 [43, 24], 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 as the use of SAT solvers for itemset mining [44, 45] or ASP [46] could be incorporated into MiningZinc too. This would require adding rewrite rules for specific encodings of the constraints into SAT/ASP. Other pattern mining settings that have been studied in a CP framework such as sequences with wildcards [47, 48, 49] and sequential patterns [50] can be expressed in MiningZinc too. The main difference is in the definition of the cover relation, and one could add such relations to the MiningZinc library as reusable functions.

Recently, a range of constraint solving techniques for pattern mining have been developed that incorporate an order or preference over patterns, such as top-k preferred patterns [51], (soft) sky patterns [52] and dominance relations [53]. These typically employ a form of dynamic CSP solving, where after each solution found by a solver, new constraints are added to the problem. This requires a finer-grained reasoning system than the one investigated in this paper.

Other tasks such as clustering have been studied in a declarative constraint-based setting as well [54, 55]. While being a very different type of data mining problem, it would be interesting to incorporate these in MiningZinc as well.

We chose Zinc [6] as the basis of our work because it is most in line with our design criteria. Other modeling languages such as Essence [4], Comet [56] and OPL [5] have no, or only limited, support for building libraries of user-defined constraints, and/or are tied to a specific solver.

We employ automatic model transformations such as the MiniZinc to FlatZinc transformation [7] and a set to Boolean transformation. Model transformations are a well-studied topic in constraint programming [57, 58], see the ModRef workshop series [59]. Even for the Zinc family of languages, a range of transformations exist [60], ranging from the ones we employ to transformations into other solving technology like SAT [61] and SMT [62], and compilation to executable machine code [63]. By building on the MiniZinc framework, we will be able to take advantage of future developments in model transformations for this

language.

Rewrite rules have been used in the context of constraint modeling before, such as Constraint Handling Rules [64]. For Zinc, an ACD term rewrite system is introduced [63] for specifying model transformations. The difference with our approach is that we must make a distinction between rewriting the model (e.g. adding redundant constraints) and using specialized algorithms to solve some of the constraints. In the latter case, some but certainly not all constraints may need to be recomputed later. Conjure [58] also uses rewrite rules to transform CP specifications, with the ability to add redundant symmetry breaking constraints and perform algorithm/model selection; the main difference is that our rewrite process creates compositions of different algorithms instead of a single CP model.

Finally, the use of multiple algorithms for solving constraint satisfaction problems has been studied in constraint programming before, most notable in the hybridization of both generic constraint programming and optimization/OR techniques [65]. Several approaches have been studied including to incorporate algorithms in CP solvers by means of global constraints, to incorporate information of one algorithm (such as relaxations of IP solvers) as constraints in a CP system [66] or to add CP propagation inside an integer programming system [67]; see [68] for a nice overview of hybrid algorithms in CP. The idea of *chaining* specialized algorithms for enumeration problems and the automatic detection of such execution strategies has, to the best of our knowledge, not been studied before.

7. Conclusions

MiningZinc aims at bringing the benefits of the declarative modeling + solving approach from the field of constraint programming to the field of data mining. It extends the modeling language MiniZinc with a library for constraint-based itemset mining. Furthermore, an execution mechanism has been designed to support a wide range of general purpose and specialized solvers and to allow for different execution strategies. Its implementation incorporates many state-of-the-art constraint programming and data mining solvers, as well as specialized algorithms. The resulting architecture is flexible and extensible. The experiments have shown that this leads to execution strategies with state-of-the-art performance for many known as well as novel constraint-based mining tasks.

The benefits of the modeling + solving approach to data mining are manyfold. First, it allows for the natural and declarative modeling and solving of many constraint-based data mining problems. Second, it provides a uniform interface to many data mining systems. It is very expressive through its use of constraint programming technology and can tackle a wide spectrum of constraint-based mining tasks. This should facilitate the comparison of different algorithms as well as the re-use of software. A potential benefit of the modeling + solving approach to data mining is also the possible emergence of standard languages and (integrated) systems for data mining, as in constraint programming. Current limitations are that the problem must be expressed over integer

and set variables (no other explicit data structures), and that search is assumed to be exhaustive (no greedy/heuristic approximations).

Data mining also raises new challenges for constraint programming as the solutions offered by the modeling + solving approach need to 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.

The approach taken in MiningZinc is to provide a rewrite mechanism for generating different execution plans that may involve different algorithms, each addressing part of the task. Experiments have shown that the performance of different execution plans can vary greatly, which invites for automatic algorithm selection techniques. However, the breadth of problems that can be formulated, different execution plans that can be produced and the sensitivity of some problems towards a single threshold raises challenges. Other directions for future work include the investigation of heuristic search strategies and the extension of the framework towards more complex pattern types and other data mining tasks such as clustering [55].

Acknowledgements. This work was supported by the Research Foundation—Flanders by means of a Postdoc grant 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

[1] L. De Raedt, S. Nijssen, B. O'Sullivan, P. Van Hentenryck, Constraint programming meets machine learning and data mining (dagstuhl seminar 11201)., Dagstuhl Reports 1 (5) (2011) 61–83.

[2] J. Han, M. Kamber, Data Mining: Concepts and Techniques, Morgan Kaufmann, 2000.

[3] K. Marriott, N. Nethercote, R. Rafeh, P. J. Stuckey, M. Garcia De La Banda, M. Wallace, The design of the Zinc modelling language, Constraints 13 (3) (2008) 229–267.

[4] A. Frisch, W. Harvey, C. Jefferson, B. M. Hernández, I. Miguel, Essence: A constraint language for specifying combinatorial problems, Constraints 13 (3) (2008) 268–306.

[5] P. Van Hentenryck, The OPL optimization programming language, MIT Press, 1999.

[6] N. Nethercote, P. J. Stuckey, R. Becket, S. Brand, G. J. Duck, G. Tack, MiniZinc: Towards a standard CP modelling language, in: CP, Vol. 4741 of LNCS, Springer, 2007, pp. 529–543.

[7] P. Stuckey, G. Tack, MiniZinc with functions, in: Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems, Vol. 7874 of LNCS, Springer Berlin Heidelberg, 2013, pp. 268–283.

[8] T. Guns, S. Nijssen, L. De Raedt, Itemset mining: A constraint programming perspective, Artif. Intell. 175 (12-13) (2011) 1951–1983.

[9] T. Guns, A. Dries, G. Tack, S. Nijssen, L. De Raedt, MiningZinc: A modeling language for constraint-based mining, in: Proceedings of the Twenty-Third International Joint Conference on Artificial Intelligence, AAAI Press, 2013, pp. 1365–1372.

[10] R. Agrawal, T. Imielinski, A. N. Swami, Mining association rules between sets of items in large databases, in: Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data, ACM Press, 1993, pp. 207–216.

[11] C. C. Aggarwal, J. Han (Eds.), Frequent Pattern Mining, Springer, 2014.

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

[13] F. Bonchi, C. Lucchese, Extending the state-of-the-art of constraint-based pattern discovery, Data Knowl. Eng. 60 (2) (2007) 377–399.

[14] J.-F. Boulicaut, B. Jeudy, Constraint-based data mining, in: Data Mining and Knowledge Discovery Handbook, 2nd Edition, Springer, 2010, pp. 339–354.

[15] F. Rossi, P. v. Beek, T. Walsh, Handbook of Constraint Programming (Foundations of Artificial Intelligence), Elsevier Science Inc., New York, NY, USA, 2006.

[16] P. J. Stuckey, R. Becket, J. Fischer, Philosophy of the MiniZinc challenge, Constraints 15 (3) (2010) 307–316.

[17] J. Vreeken, M. Leeuwen, A. Siebes, Krimp: Mining itemsets that compress, Data Min. Knowl. Discov. 23 (1) (2011) 169–214.

[18] N. Pasquier, Y. Bastide, R. Taouil, L. Lakhal, Discovering frequent closed itemsets for association rules, in: Database Theory, Vol. 1540 of LNCS, Springer, 1999, pp. 398–416.

[19] R. Chan, Q. Yang, Y.-D. Shen, Mining high utility itemsets, in: ICDM, 2003, pp. 19–26.

[20] C. K. Chui, B. Kao, E. Hung, Mining frequent itemsets from uncertain data, in: Advances in Knowledge Discovery and Data Mining, 11th Pacific-Asia Conference, PAKDD 2007, Nanjing, China, May 22-25, 2007, Proceedings, 2007, pp. 47–58.

[21] P. K. Novak, N. Lavrac, G. I. Webb, Supervised descriptive rule discovery: A unifying survey of contrast set, emerging pattern and subgroup mining, J. Mach. Learn. Res. 10 (2009) 377–403.

[22] J. Fürnkranz, P. A. Flach, ROC 'n' rule learning – towards a better understanding of covering algorithms, Machine Learning 58 (1) (2005) 39–77.

[23] S. Nijssen, A. Jimenez, T. Guns, Constraint-based pattern mining in multi-relational databases, in: Proceedings of the 11th IEEE International Conference on Data Mining Workshops,, IEEE, 2011, pp. 1120–1127.

[24] T. Guns, S. Nijssen, L. De Raedt, k-Pattern set mining under constraints, IEEE Transactions on Knowledge and Data Engineering 25 (2) (2013) 402–418.

[25] M. Khiari, P. Boizumault, B. Crémilleux, A generic approach for modeling and mining n-ary patterns, in: ISMIS, 2011, pp. 300–305.

[26] T. Uno, M. Kiyomi, H. Arimura, LCM ver. 2: Efficient mining algorithms for frequent/closed/maximal itemsets, in: FIMI, Vol. 126 of CEUR Workshop Proceedings, CEUR-WS.org, 2004.

[27] R. Elmasri, S. Navathe, Fundamentals of Database Systems, 6th Edition, Addison-Wesley Publishing Company, USA, 2010.

[28] L. Kotthoff, Algorithm selection for combinatorial search problems: A survey, AI Magazine 35 (3) (2014) 48–60.

[29] C. Schulte, G. Tack, M. Lagerkvist, Gecode, a generic constraint development environment, `www.gecode.org` (2013).

[30] Opturion, CPX (2014).
URL `http://www.opturion.com/cpx.html`

[31] N. van Omme, L. Perron, V. Furnon, or-tools user's manual, Tech. rep., Google (2014).

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

[33] M. Liu, J. Qu, Mining high utility itemsets without candidate generation, in: Proceedings of the 21st ACM international conference on Information and knowledge management, ACM, 2012, pp. 55–64.

[34] P. Fournier-Viger, A. Gomariz, A. Soltani, T. Gueniche, Spmf: Open-source data mining platform, `http://www.philippe-fournier-viger.com/spmf/` (2013).

[35] S. Nijssen, T. Guns, L. De Raedt, Correlated itemset mining in ROC space: A constraint programming approach, in: KDD, ACM, 2009, pp. 647–656.

[36] A. Frank, A. Asuncion, UCI machine learning repository, available from `http://archive.ics.uci.edu/ml` (2010).

[37] B. Goethals, M. J. Zaki, Advances in frequent itemset mining implementations: report on FIMI'03, in: SIGKDD Explorations, Vol. 6, 2004, pp. 109–117.

[38] H. Mannila, Inductive databases and condensed representations for data mining, in: ILPS, 1997, pp. 21–30.

[39] R. Meo, G. Psaila, S. Ceri, A new SQL-like operator for mining association rules, in: VLDB, 1996, pp. 122–133.

[40] T. Imielinski, A. Virmani, MSQL: A query language for database mining, Data Mining and Knowledge Discovery 3 (1999) 373–408.

[41] H. Blockeel, T. Calders, É. Fromont, B. Goethals, A. Prado, C. Robardet, An inductive database system based on virtual mining views, Data Min. Knowl. Discov. 24 (1) (2012) 247–287.

[42] M. Järvisalo, Itemset mining as a challenge application for answer set enumeration, in: Logic Programming and Nonmonotonic Reasoning, Vol. 6645 of LNCS, Springer, 2011, pp. 304–310.

[43] J.-P. Métivier, P. Boizumault, B. Crémilleux, M. Khiari, S. Loudni, A constraint language for declarative pattern discovery, in: ACM Symposium on Applied Computing, ACM, 2012, pp. 119–125.

[44] R. Henriques, I. Lynce, V. M. Manquinho, On when and how to use SAT to mine frequent itemsets, CoRR abs/1207.6253.

[45] E. Coquery, S. Jabbour, L. Sais, Y. Salhi, et al., A SAT-based approach for discovering frequent, closed and maximal patterns in a sequence., in: European Conference on Artificial Intelligence (ECAI), Vol. 242, 2012, pp. 258–263.

[46] M. Järvisalo, Itemset mining as a challenge application for answer set enumeration, in: Logic Programming and Nonmonotonic Reasoning - 11th International Conference, LPNMR 2011, Vancouver, Canada, May 16-19, 2011. Proceedings, 2011, pp. 304–310.

[47] S. Jabbour, L. Sais, Y. Salhi, Boolean satisfiability for sequence mining, in: Proceedings of the 22nd ACM international conference on information & knowledge management, ACM, 2013, pp. 649–658.

[48] E. Coquery, S. Jabbour, L. Sais, A constraint programming approach for enumerating motifs in a sequence, in: Data Mining Workshops (ICDMW), 2011 IEEE 11th International Conference on, IEEE, 2011, pp. 1091–1097.

[49] A. Kemmar, W. Ugarte, S. Loudni, T. Charnois, Y. Lebbah, P. Boizumault, B. Cremilleux, Mining relevant sequence patterns with CP-based framework, in: Tools with Artificial Intelligence (ICTAI), 2013 IEEE 25th International Conference on, IEEE, IEEE Computer Society, 2014, pp. 552–559.

[50] J.-P. Métivier, S. Loudni, T. Charnois, A constraint programming approach for mining sequential patterns in a sequence database, in: ECML/PKDD 2013 Workshop on Languages for Data Mining and Machine Learning, 2013, also available as arXiv:1311.6907.

[51] S. Jabbour, L. Sais, Y. Salhi, The top-k frequent closed itemset mining using top-k SAT problem, in: Machine Learning and Knowledge Discovery in Databases, Springer, 2013, pp. 403–418.

[52] W. Ugarte, P. Boizumault, S. Loudni, B. Crémilleux, A. Lepailleur, Soft threshold constraints for pattern mining, in: Eleventh International Conference on Integration of Artificial Intelligence (AI) and Operations Research (OR) techniques in Constraint Programming (CPAIOR-14), 2014.

[53] B. Negrevergne, A. Dries, T. Guns, S. Nijssen, Dominance programming for itemset mining, in: 13th IEEE International Conference on Data Mining,, IEEE Computer Society, 2013, pp. 557–566.

[54] J.-P. Métivier, P. Boizumault, B. Crémilleux, M. Khiari, S. Loudni, Constrained clustering using SAT, in: Advances in Intelligent Data Analysis XI, Springer, 2012, pp. 207–218.

[55] K.-C. Duong, C. Vrain, et al., A declarative framework for constrained clustering, in: ECML/PKDD, Machine Learning and Knowledge Discovery in Databases, Springer, 2013, pp. 419–434.

[56] P. Van Hentenryck, L. Michel, Constraint-Based Local Search, MIT Press, 2005.

[57] P. Flener, J. Pearson, M. Ågren, Introducing ESRA, a relational language for modelling combinatorial problems, in: LOPSTR, Vol. 3018 of LNCS, Springer, 2003, pp. 214–232.

[58] A. Frisch, C. Jefferson, B. M. Hernández, I. Miguel, The rules of constraint modelling, in: IJCAI, Professional Book Center, 2005, pp. 109–116.

[59] A. Frisch, et al., International workshops on constraint modelling and reformulation.
URL `http://www-users.cs.york.ac.uk/~frisch/ModRef/`

[60] R. Becket, S. Brand, M. Brown, G. J. Duck, T. Feydy, J. Fischer, J. Huang, K. Marriott, N. Nethercote, J. Puchinger, R. Rafeh, P. J. Stuckey, G. Wallace, The many roads leading to rome: Solving Zinc models by various solvers (2008).

[61] J. Huang, Universal Booleanization of constraint models, in: P. Stuckey (Ed.), Principles and Practice of Constraint Programming, Vol. 5202 of LNCS, Springer Berlin Heidelberg, 2008, pp. 144–158.

[62] M. Bofill, M. Palahí, J. Suy, M. Villaret, fzn2smt, `http://ima.udg.edu/Recerca/lap/fzn2smt/index.html` (2011).

[63] G. J. Duck, L. De Koninck, P. J. Stuckey, Cadmium: An implementation of ACD term rewriting, in: ICLP, Vol. 5366 of LNCS, Springer, 2008, pp. 531–545.

[64] T. Frühwirth, Constraint Handling Rules, Cambridge University Press, 2009.

[65] F. Ajili, M. Wallace, Hybrid problem solving in eclipse, in: M. Milano (Ed.), Constraint and Integer Programming, Vol. 27 of Operations Research/-Computer Science Interfaces Series, Springer US, 2004, pp. 169–206.

[66] F. Focacci, A. Lodi, M. Milano, Exploiting relaxations in cp, in: Constraint and Integer Programming, Vol. 27 of Operations Research/Computer Science Interfaces Series, Springer US, 2004, pp. 137–167.

[67] T. Achterberg, T. Berthold, T. Koch, K. Wolter, Constraint integer programming: A new approach to integrate cp and mip, in: L. Perron, M. A. Trick (Eds.), Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems, Vol. 5015 of LNCS, Springer Berlin Heidelberg, 2008, pp. 6–20.

[68] M. Wallace, Hybrid algorithms in constraint programming, in: F. Azevedo, P. Barahona, F. Fages, F. Rossi (Eds.), Recent Advances in Constraints, Vol. 4651 of LNCS, Springer Berlin Heidelberg, 2007, pp. 1–32.