Automatic solver chaining in MiningZinc

Tias Guns[1], Anton Dries[1], Guido Tack[2], Siegfried Nijssen[1,3], and Luc De Raedt[1]

[1] KU Leuven, Belgium
{*firstname.lastname*}@cs.kuleuven.be
[2] National ICT Australia (NICTA) and Monash University, Australia
guido.tack@monash.edu
[3] LIACS, Universiteit Leiden s.nijssen@liacs.leidenuniv.nl

Abstract. We describe the reformulation and execution mechanism of MiningZinc, a declarative framework for constraint-based data mining. The MiningZinc execution mechanism determines how to compute solutions for MiningZinc 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. 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, leading to tailored high-performance solving of the declarative models.

1 Introduction

The fields of data mining and constraint programming are amongst the most successful subfields of artificial intelligence. Yet, their methodologies are quite different. Constraint programming advocates a declarative modeling and solving approach to constraint satisfaction and optimisation problems. Data mining on the other hand 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 very efficient, but hard to modify or reuse in other applications. Nevertheless, there is a need for generic techniques that can handle variations of known tasks as well as application-driven constraints [1,2].

However, when using generic declarative modeling languages, the performance of the underlying solvers is orders of magnitudes slower than state-of-the-art specialized systems.

The aim of this paper is to contribute to bridging the methodological gap between the fields of data mining and constraint programming. We do this by investigating effective ways to transform declarative specifications of mining problems, such that generic solvers, specialised algorithms and combinations thereof

can be used. 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 and corresponding systems have been proposed in the literature[3]; this makes it an ideal showcase for a declarative approach to data mining.

The language used is MiniZinc [4] version 2.0, extended with a library of functions and constraints that are common in constraint-based itemset mining. The choice of language is driven by the need for solver-independence and the ability to define user-defined functions. The execution mechanism is much more elaborate than that of standard MiniZinc. The standard mechanism translates individual constraints to constraints that are supported by a specific solver. Our method reasons over groups of constraints and can automatically compose execution strategies involving multiple solvers.

The MiningZinc framework builds on our earlier CP4IM framework [5], which showed the feasibility of constraint programming for pattern mining. The present paper also extends our earlier publication on MiningZinc [6], whose execution mechanism was restricted to using a single algorithm or generic solver. Instead, we now use rewrite rules to automatically construct execution plans consisting of multiple solver/algorithm components. This workshop paper is a summary of an upcoming journal paper.

2 Modeling constraint-based mining problems

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. [7] 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 then, itemset mining has been extended in many directions, including more structured types of patterns such as sequences, trees and graphs as well as many applications [3].

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 $|\phi_{\mathcal{D}}(I)| \geq Freq$, 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 $|\phi_{\mathcal{D}}(I)| \geq Freq$ is called a *frequent itemset*.

Listing 1.1. "Constraint-based mining"

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

Listing 1.2. "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;
```

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. [7,8,9]. This has lead to the research topic of *constraint-based itemset mining* [10]. It is in this constraint-based setting that the rich modeling facilities of constraint programming are appealing.

2.1 Itemset mining in standard MiniZinc

Itemset mining problems can be modeled directly in MiniZinc. A MiniZinc model of the frequent itemset mining problem is shown in Listing 1.1. We assume familiarity with the syntax. The model represents the item and transaction identifiers in $\mathcal{I}$ and $\mathcal{S}$ by natural numbers from 1 to Nrl ($= n$) and 1 to NrT ($= m$) respectively (lines 1 and 2). 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. The minimum frequency constraint is posted on line 4, which corresponds closely to the formal notation $|\phi_{\mathcal{D}}(I)| \geq \mathit{Freq}$, where ϕ is a function named cover.

A distinguishing feature of MiniZinc is its support for user defined-predicates, and since version 2.0, user-defined functions [4]. A declaration of the cover function is shown in Listing 1.2. Recall that the formal definition of cover is

$\phi_{\mathcal{D}}(I) = \{t | I \subseteq \mathcal{D}(t)\}$. 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 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.

2.2 MiningZinc

MiniZinc allows one to place common functions and predicates into libraries, to facilitate their reuse in different models. The language component of the MiningZinc framework is such a library.

The two key building blocks of the MiningZinc library are the cover and cover_inv functions. The cover function $\phi_{\mathcal{D}}(I) = \{t | I \subseteq \mathcal{D}(t)\}$ was already given in Listing 1.2. For brevity we ommit the cover_inv function ψ_D, but see [6]. The library includes other helper functions such as weighted_sum and different ways to print item and transaction sets.

The library also includes predicates that express redundant constraints 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.

Another type of redundant information available in the library is a search annotation. This is an annotation that can be added to the search keyword, and that specifies the order in which to search over the variables. We also added the *enumerate* search annotation to differentiate, in the model, between satisfaction (one solution) and enumeration (all solutions) problems.

Direct database access Another annotation we added is the *query* keyword, which can be added to a variable declaration, for example:

```
array [] of set of int: 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.

2.3 Constraint-based mining in MiningZinc

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

Listing 1.3. "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;
```

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 incomplete methods).

Listing 1.3 is a constraint-based mining model that extends the MiniZinc model in Listing 1.1. 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, see [11] for details. Lines 8/9 represent a common cost-based constraint [9]; 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. Many such constraints have been studied in the literature [1], though for the remaining of this paper we will use frequent, closed and cost-based itemset mining as running examples.

3 MiningZinc execution mechanism

The MiniZinc language shown 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. The first step in the analysis process is to transform this model into a FlatZinc program [17], which we use as a *normalized* form for simplifying the analysis. Given a set of algorithms and rewrite rules, the FlatZinc program is then transformed into all possible sequences of algorithms that can solve this problem; one such sequence of algorithms is called

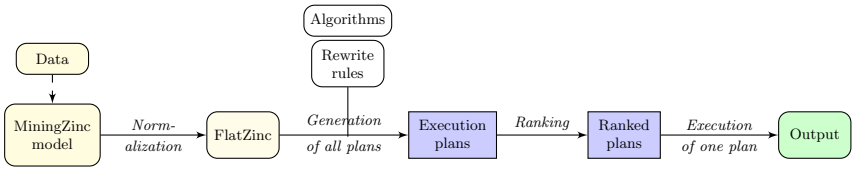

Fig. 1. Overview of the MiningZinc toolchain

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.

3.1 Normalization

The purpose of converting to FlatZinc is to enable reasoning over a *set* of constraints instead of having to reason over nested constraints and loops. This also simplifies the detection of equivalent or overlapping formulations.

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 [4]. It also performs common subexpression elimination at the global scope. For example, Listing 1.3 contains twice the function cover(Items, TDB). In the FlatZinc code, only one call X = cover(Items,TDB) will remain, and variable X will be shared by all expressions that contained that call. FlatZinc also supports annotations, for example *enumerate* in Listing 1.3.

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. This allows us to reason at the level of the library and is also more compact than the FlatZinc generated for a CP solver, since the definitions typically loop over the data.

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. $Y \leq 1$, $S \subseteq \{2, 4\}$.
- $(X_0 = f(X_1, \ldots X_n))$, 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.

Example 2. Consider the problem of finding frequent itemsets with minimum frequency of 20 and containing at least 3 items (Listing 1.1 + card(Items) >= 3). We represent the datasets by $\{\ldots\}$ to indicate that it is a constant. The normalized FlatZinc model obtained is the following (leaving the domain implicit):

$\mathcal{V} = \{Items, T, SI, ST\}$
$\mathcal{C} = \{T = cover(Items, \{...\}), ST = card(T), SI = card(Items), ST \geq 20, SI \geq 3\}$

3.2 Execution plans and algorithm predicates

An execution plan specifies which parts of a MiningZinc model are handled by which algorithms or solving techniques in sequence. In an execution plan, the execution of an algorithm or CP solver 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 3. 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. It is true for any F-frequent itemset I. A specific atom expressed using this predicate is $LCM(10, \{\ldots\}, Items)$.

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 has therefore no fixed arity, and we assume that the predicate is parameterized with a set of constraints. There are also no mode restrictions as a CP system will find groundings to all non-ground variables.

Example 4. A predicate $Gecode_C(V_1, \ldots, V_n)$ represents the Gecode solver, 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.

Execution plans 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 5. For the model of Example 2 the following is a valid execution plan that uses the LCM and Gecode predicates of Example 3 and 4:
$[LCM(10, \{\ldots\}, I), Gecode_{(SI=card(I), SI\geq 3)}(I)]$.

3.3 Building an execution plan with 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 $\vdash$ symbol denotes the transition of one state to another.

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_1 is an expression, by $C_1\theta$ 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 $C_1\theta \subseteq C$, the set of constraints C_1 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.

3.3.1 Rules for redundant constraints Let C_1 be a set of constraints and C_2 an equivalent or implied, but typically much more involved, constraint set over the same variables. We call C_2 a redundant constraint set given constraints $C \supseteq C_1$. Taking substitutions to match variables names into account, we have the following rewrite rule:

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

Example 6. Past work showed that the execution of the frequent itemset mining task is more efficient in some CP systems if the frequency constraint is verified for each item individually, as well as for the entire itemset [5]. Let $C_2 = \{minfreq\_redundant(I, D, V)\}$ represent the per-item frequency constraints. C_2 is redundant to the set of constraints $C_1 = \{A = cover(I, D), B = card(A), V \leq B\}$. Then for the model of Example 2 (depicted by C_M) we have the rewriting:

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

3.3.2 Rules for specialized algorithms All specialized algorithms have a predicate definition $p(\pm_1 V_1, \ldots, \pm_n V_n)$ and a set of constraints C that define the problem this algorithm solves. 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 7. 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 as possible of the subsumed constraints in C from C_M. Indeed, running the algorithm will ensure that these constraints are satisfied, so we can avoid unnecessarily recomputing or verifiying them again. Unfortunately, we cannot always remove all subsumed constraints.

Example 8. If our model has constraints $\{T = \mathit{cover}(I, \{\ldots\}), ST = \mathit{card}(T), ST \geq 20, ST \leq 40\}$, and we again use the LCM algorithm to solve part of this model, we cannot remove the constraints $ST = \mathit{card}(T)$ and $T = \mathit{cover}(I, D)$, 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. If these auxiliary variables are also used outside the constraint set of this algorithm, they may not be removed. Let $C' \subseteq C$ be all constraints defining the algorithm that do not contain such auxiliary variables. We can now define the following rewrite rule for specialized algorithms:

IF $C\theta$ subsumes constraints C_M for mode declaration $p(\pm_1 V_1, \ldots, \pm_n V_n)$ such that $[L, p(V_1\theta, \ldots, V_n\theta)]$ is conform the modes of p,
THEN $(L, C_M) \vdash ([L, p(V_1\theta, \ldots, V_n\theta)], C_M \setminus C'\theta)$.

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

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

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. 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 are typically more efficient on a Boolean encoding rather than reasoning over set variables directly.

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 indicate whether set variables must be translated to Booleans just before execution.

3.4 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: 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; 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. 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.

3.5 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 [12]. For combinatorial problems, computing or estimating the number of solutions produced is a hard problem in itself. Furthermore, algorithms are typically sensitive to the size and properties of the input data at hand. Choosing the best individual algorithm for a task has been studied in the algorithm selection and portfolio literature [13]. In MiningZinc on the other hand, we have to choose the best execution *plan*, which may consist of chains of algorithms.

Using an approach in which each plan is treated as one meta-algorithm is not feasible as a MiningZinc formulation can lead to new execution plans that have never been observed before. Additionally, different chaining of algorithms can also lead to differences in runtime, depending on the data generated by the previous algorithms. Furthermore, the input to the next algorithm in a chain is not known until all its predecessor algorithms are run.

This work does not aim to solve the hard problem of plan selection. 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. We assume a global ordering over algorithms to break ties.

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 a global ordering of the algorithms.

Finally, *generic plans* consist of one CP system that solves the entire problem. We rank them based on the number of constraints (more is better, e.g. redundant constraints) and break ties using a global ordering of solvers. One could apply standard algorithm selection techniques here.

We proposed just one heuristic way of ordering the strategies, based on common sense principles. We leave more advanced mechanisms for future work.

3.6 Execution of a plan

One plan is executed in a similar way as a Prolog query. The execution proceeds left-to-right. An algorithm is used to find all groundings for non-grounded variables. If all variables are ground this amounts to checking whether the constraints are satisfied. Each grounding will be passed in turn to the next algorithm. The evaluation backtracks until all groundings for all predicates have been evaluated.

Example 9. In the execution plan of Example 5, $[LCM(10, \{\ldots\}, I)$, $Gecode_{(SI=card(I),SI\geq 3)}(I)]$, 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.

4 Experiments

The MiningZinc framework is implemented in Python with key components, such `libminizinc`[4] for the MiniZinc to FlatZinc conversion, written in C++. The constraint solvers used are Gecode [14], Opturion's CPX [15], Google or-tools [16] and the g12 solvers from the MiniZinc 1.6 distribution [17]. 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 [18] and Christian Borgelt's implementations of Apriori (v5.73), Eclat (v3.74) and FP-Growth (v4.48) [19]; these are the state-of-the art for efficient constraint-based mining. Input/output 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 [20] and from the FIMI repository [21]. Experiments were run on linux computers with quad-core Intel i7 processors. The MiningZinc system and datasets used can be downloaded at `http://dtai.cs.kuleuven.be/CP4IM/miningzinc/`.

Cost of computing all execution strategies 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 1.1 (Freq/Fr) and Listing 1.3 (Clo+MinCost). Solving times are not shown as they depend on the threshold supplied (see the following experiments).

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

Dataset	#Tr	#It	Freq	Fr+Clo	Fr+Clo+ Mincost
primary-tumor	336	31	0.069 / 0.007	0.069 / 0.009	0.109 / 0.011
mushroom	8124	112	0.780 / 0.032	0.800 / 0.036	0.799 / 0.035
pumsb_star	49046	2088	13.489 / 0.546	13.895 / 0.563	13.459 / 0.541
retail	88162	16470	5.132 / 0.235	5.290 / 0.243	5.086 / 0.233
T40I10D100K	100000	942	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. Right, MiningZinc analysis for 3 different tasks (frequent, frequent+closed and frequent+closed+mincost); time taken for 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
primary-tumor	2.14	11.61	**1.30**	1.61	**2.65**	33.20	9.66	6.48	**1.07**	1.57
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**
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 the minfreq redundant constraint was added. Lowest average runtime per solver in bold.

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 and it is not optimized for parsing large matrices. This can be sidestepped by loading in the data directly from a database with the *query* annotation.

CP Solver performance MiningZinc has the ability to automatically add redundant constraints or transform set variables to Boolean variables. We compare this on three state-of-the-art CP solvers that won medals in the 2013 MiniZinc challenge: Gecode, Opturion's CPX, Google's or-tools. Or-tools does not support constraints over set variables and requires the set to Boolean transformation.

Table 2 shows average runtimes for the different reformulations. In general, different datasets have different characteristics and different numbers of frequent itemsets. However, one can immediately see that the type of reformulation to use can depends mostly on the solver used; for Gecode and or-tools, using Boolean variables with redundant frequency constraints yields lower average runtimes, while, surprisingly, 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.

In case of set variables adding redundant constraints seems to mostly slow down the process. When using the Boolean transformation it often improves efficiency, sometimes significantly.

Different execution plans We now look at the problem of finding all closed itemsets that also satisfy a minimum size constraint on the size of the itemset.

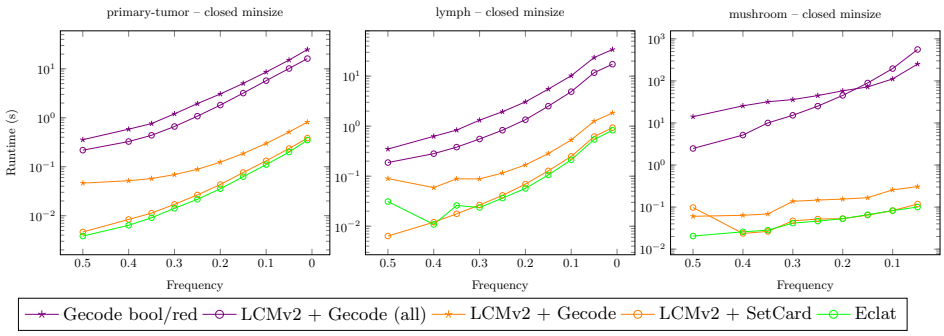

Fig. 2. Comparison of hybrid solve strategies for closed frequent itemset mining with a size constraint (size ≥ 4).

Figure 2 shows a comparison of 5 approaches to solve this problem: a) Using Gecode (with redundant frequency constraints and Boolean encoding); b) using the specialized algorithm LCMv2, followed by Gecode that checks *all* constraints in the model; c) using LCMv2, followed by Gecode checking only non-stasified constraints (i.e. the size constraint); d) LCMv2 and a specialized algorithm for post-processing the size constraint; e) using the specialized algorithm Eclat which can solve the complete problem.

Across all datasets, we observe unsurprisingly that the specialized algorithm is usually the fastest. It is followed closely by the specialized post-processor and difference is negligible in most cases. A specialized algorithm followed by a generic CP checker is often faster than the pure CP approach, especially when already satisfied constraints are removed. We also experimented with several larger datasets than shown above. 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.

5 Related Work

Work on constraint solving for itemset mining [5,22] 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 [23,24], though again tied to one particular solving technology.

We chose MiniZinc [17] as the basis of our work because it is most in line with our design criteria. Other modeling languages such as Essence [25], Comet [26] and OPL [27] 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 [4] and a set to Boolean transformation. Model transformations are a well-studied topic in constraint programming [28,29]. Even for the Zinc family of languages, a range of transformations exist [30], ranging from the ones

we employ to transformations into other solving technology like SAT [31] and SMT [32], and compilation to executable machine code [33]. By building on the MiniZinc framework, we will be able to take advantage of future developments in model transformations for this language. Conjure [29] also uses rewrite rules to transform CP specifications, with the ability to add implied symmetry breaking constraints and perform algorithm/model selection; the main difference is that our rewrite process creates compositions of different *algorithms* in addition to different CP models.

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 [34]. 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 [35] or to add CP propagation inside an integer programming system [36]; see [37] for a nice overview of hybrid algorithms in CP. The idea of *chaining* specialized algorithms for enumeration problems and the automatic detection of such strategies has, to our best knowledge, not been studied before.

6 Conclusions

MiningZinc applies the declarative modeling + solving paradigm from constraint programming to the field of data mining. Data mining raises new challenges as the solutions offered by the declarative paradigm need to be competitive with that of standard data mining algorithms. This is non-trivial because mining algorithms are highly optimized for large datasets, in contrast to generic solvers.

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. To achieve this, MiningZinc extends the solver-independent modeling language MiniZinc with a library for constraint-based itemset mining. Its execution mechanism can automatically detect and compose different solving methods, including general purpose constraint solvers, specialized mining algorithms and other specialized algorithms.

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 [38].

Acknowledgements. This work was supported by the Research Foundation—Flanders, by the European Commission project FP7-284715 "Inductive Constraint Programming", and 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. S. Dzeroski, B. Goethals, P. Panov, Inductive Databases and Constraint-Based Data Mining, 1st Edition, Springer-Verlag New York, Inc., 2010.
2. 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.
3. C. C. Aggarwal, J. Han (Eds.), Frequent Pattern Mining, Springer, 2014.
4. 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.
5. T. Guns, S. Nijssen, L. De Raedt, Itemset mining: A constraint programming perspective, Artif. Intell. 175 (12-13) (2011) 1951–1983.
6. 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.
7. 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.
8. H. Mannila, H. Toivonen, Levelwise search and borders of theories in knowledge discovery, Data Min. Knowl. Discov. 1 (3) (1997) 241–258.
9. F. Bonchi, C. Lucchese, Extending the state-of-the-art of constraint-based pattern discovery, Data Knowl. Eng. 60 (2) (2007) 377–399.
10. J.-F. Boulicaut, B. Jeudy, Constraint-based data mining, in: Data Mining and Knowledge Discovery Handbook, 2nd Edition, Springer, 2010, pp. 339–354.
11. 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.
12. R. Elmasri, S. Navathe, Fundamentals of Database Systems, 6th Edition, Addison-Wesley Publishing Company, USA, 2010.
13. L. Kotthoff, Algorithm selection for combinatorial search problems: A survey, AI Magazine 35 (3) (2014) 48–60.
14. C. Schulte, G. Tack, M. Lagerkvist, Gecode, a generic constraint development environment, `www.gecode.org` (2013).
15. Opturion, CPX (2014).
URL `http://www.opturion.com/cpx.html`
16. N. van Omme, L. Perron, V. Furnon, or-tools user's manual, Tech. rep., Google (2014).
17. 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.
18. 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.
19. C. Borgelt, Frequent item set mining, Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery 2 (6) (2012) 437–456.
20. A. Frank, A. Asuncion, UCI machine learning repository, available from `http://archive.ics.uci.edu/ml` (2010).

21. B. Goethals, M. J. Zaki, Advances in frequent itemset mining implementations: report on FIMI'03, in: SIGKDD Explorations, Vol. 6, 2004, pp. 109–117.
22. 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.
23. 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.
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. 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.
26. P. Van Hentenryck, L. Michel, Constraint-Based Local Search, MIT Press, 2005.
27. P. Van Hentenryck, The OPL optimization programming language, MIT Press, 1999.
28. 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.
29. A. Frisch, C. Jefferson, B. M. Hernández, I. Miguel, The rules of constraint modelling, in: IJCAI, Professional Book Center, 2005, pp. 109–116.
30. 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).
31. 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.
32. M. Bofill, M. Palahí, J. Suy, M. Villaret, fzn2smt, `http://ima.udg.edu/Recerca/lap/fzn2smt/index.html` (2011).
33. 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.
34. 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.
35. 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.
36. 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.
37. 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.
38. 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.