# Modeling in MiningZinc

Anton Dries[1], Tias Guns[1], Siegfried Nijssen[1,2], Behrouz Babaki[1], Thanh Le Van[1], Benjamin Negrevergne[1], Sergey Paramonov[1], and Luc De Raedt[1]

[1] Department of Computer Science, KU Leuven
[2] LIACS, Universiteit Leiden
{firstname.lastname}@cs.kuleuven.be

**Abstract.** MiningZinc offers a framework for modeling and solving constraint-based mining problems. The language used is MiniZinc, a high-level declarative language for modeling combinatorial (optimisation) problems. This language is augmented with a library of functions and predicates that help modeling data mining problems and facilities for interfacing with databases. We show how MiningZinc can be used to model constraint-based itemset mining problems, for which it was originally designed, as well as sequence mining, Bayesian pattern mining, linear regression, clustering data factorization and ranked tiling. The underlying framework can use any existing MiniZinc solver. We also showcase how the framework and modeling capabilities can be integrated into an imperative language, for example as part of a greedy algorithm.

## 1 Introduction

The traditional approach to data mining is to develop specialized algorithms for specific tasks. This has led to specialized algorithms for many tasks, among which classification, clustering and association rule discovery [19, 30, 35]. In many cases these algorithms support certain kinds of constraints as well; in particular constraint-based clustering and constraint-based pattern mining are established research areas [5, 6, 2]. Even though successful for specific applications, the downside of specialized algorithms is that it is hard to adapt them to novel tasks.

In recent years, researchers have explored the idea of using generic solvers to tackle data mining problems such as itemset mining [16, 22], sequence mining [7, 31] and clustering [13, 11]. These approaches start from the insight that many data mining problems can be formalized as either a constraint satisfaction problem, or a constrained optimization problem. The advantage is that, just as in constraint programming, new tasks can be addressed by changing the constraint specification.

Although these techniques allow flexibility in modeling new tasks, they are often tied to one particular solver. To address this shortcoming we introduced MiningZinc [15], a solver-independent language and framework for modeling constraint-based data mining tasks. That work focussed on constraint-based itemset mining problems and the solving capabilities of the framework. In this work, we focus on the modeling aspects of the MiningZinc language and we show

for a wide range of data mining tasks how they can be modeled in MiningZinc, namely sequence mining, Bayesian pattern mining, linear regression, clustering data factorization and ranked tiling. We end with a discussion of related work and conclusion.

## 2 Language

Ideally, a language for mining allows one to express the problems in a natural way, while at the same time being generic enough to express additional constraints or different optimization criteria. We choose to build on the MiniZinc constraint programming language for this purpose [32]. It is a subset of Zinc [25] restricted to the built-in types *bool, int, set* and *float*, and user-defined predicates and functions [34].

MiningZinc uses the MiniZinc language, that is, all models written for MiningZinc are compatible with the standard MiniZinc compiler and toolchain. However, MiningZinc offers additional functionality aimed towards modeling data mining problems:

- a library of predicates and functions that are commonly used in data mining
- extensions for reading data from different sources (e.g. a database)
- integration with Python for easy implementation of greedy algorithms

### 2.1 MiniZinc

MiniZinc is a language designed for specifying constraint problems. Listing 1 shows an example of a MiniZinc model for the well-known "Send+More=Money" problem. In this problem the goal is to assign distinct digits to the letters such that the formula holds.

This model starts with declaring the decision variables with their domains (Lines 1 and 2). The problem specification states that the variables S and M should not be zero which we encode in their domains. Next, we specify the constraints on these decision variables. On Line 4 we specify that all variables should take a different value. For this we use the all_different global constraint which is defined in MiniZinc's standard library. In order to use this constraint we include that library (Line 3). On Line 5 we specify that the numbers formed by the digits "SEND" and "MORE" should sum up to the number formed by the digits "MONEY". For the translation between the list of digits and the number they represent, we define a helper function on Line 6; it first creates a local parameter max_i that represents the largest index, and then sums over each variable multiplied by $1, 10, 100, ...$ depending on its position in the array (for example, number([S,E,N,D]) = 1000∗S+100∗E+10∗N+1∗D). Line 7 states that this is a constraint satisfaction problem. MiniZinc also supports optimization problems in which case this statement would be replaced by solve minimize <variable expression>, or likewise with maximize. Finally, Line 8 defines the output of the model.

Listing 1: An example MiniZinc model

```
1 var 1..9: S; var 0..9: E; var 0..9: N; var 0..9: D;
2 var 1..9: M; var 0..9: O; var 0..9: R; var 0..9: Y;

3 include "globals.mzn";
4 constraint all_different([S,E,N,D,M,O,R,Y]);

5 constraint number([S,E,N,D]) + number([M,O,R,E]) =
                                          number([M,O,N,E,Y]);

6 function var int: number(array[int] of var int: digits) =
    let { int: max_i = max(index_set(digits)) } in
    sum(i in index_set(digits))
           (pow(10,max_i-i) * digits[i]);

7 solve satisfy;

8 output [show([S,E,N,D]), "+", show([M,O,R,E]),
                                    "=", show([M,O,N,E,Y])];
```

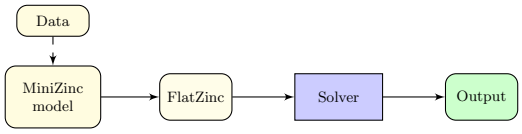


Fig. 1: Overview of the MiniZinc toolchain

Apart from the functionality demonstrated in the example, MiniZinc models can be parameterized, for example, to include external data. The values of these parameters can be loaded from an external file, or passed in through the command line.

MiniZinc is *solver-independent*, that is, MiniZinc models can be translated to the lower level language FlatZinc which is understood by a wide range of solvers. The MiniZinc toolchain does however support solver-specific optimizations through the use of solver-specific versions of the standard library and annotations. A schematic overview of the MiniZinc toolchain is shown in Figure 1.

In the following we describe how we extended MiniZinc.

### 2.2 Library

MiniZinc offers the ability to add additional libraries of commonly used predicates and functions. As part of MiningZinc, we created a minimal library for help with specifying mining and learning problems. It has two purposes: (1) to simplify modeling for the user and (2) to simplify the model analysis by the MiningZinc solver.

There are four categories of functions:

- generic helper functions
- itemset mining functions
- norms and distance functions
- extra solver annotations

There are two generic helper functions that we feel are missing in MiniZinc, namely a direct bool2float conversion `function var float: b2f(var bool: B)` and an explicit weighted sum:
`function var int: weighted_sum(array[int] of var int W, array[int] of var int X).`

Itemset mining is the best studied problem category in MiningZinc, and the key abstraction is the cover function: `cover(Items, TDB))`. Other helper functions are `coverInv(Trans, TDB))` and `frequent_items(TDB, MinFreq)`.

Many data mining and machine learning problems involve computing distances. For this reason, we added to the library functions that compute the $l_1$, $l_2$ and $l_\infty$ norms, the Manhattan, Euclidean and Chebyshev distance as well as the sum of squared errors and mean squared error:

Listing 2: "norms"

```
1  function var float: norm1(array[int] of var float: W) =
2    sum(j in index_set(W))( abs(W[j]) );
3  function var float: norm2sq(array[int] of var float: W) =
4    sum(j in index_set(W))( W[j]*W[j] );
5  function var float: norm2(array[int] of var float: W) =
6    sqrt(norm2sq(W));
7  function var float: normInf(array[int] of var float: W) =
8    max(j in index_set(W))( W[j] );
```

Listing 3: "distances"

```
1  function var float: manhDist(array[int] of var float: A,
2                               array[int] of var float: B) =
3    norm1([ A[d] - B[d] | d in index_set(A) ]);
4  function var float: euclDist(array[int] of var float: A,
5                               array[int] of var float: B) =
6    norm2([ A[d] - B[d] | d in index_set(A) ]);
7  function var float: chebDist(array[int] of var float: A,
8                               array[int] of var float: B) =
9    normInf([ A[d] - B[d] | d in index_set(A) ]);

10 function var float: sumSqErr(array[int] of var float: A,
11                              array[int] of var float: B) =
12   norm2sq([ A[d] - B[d] | d in index_set(A) ]);
13 function var float: meanSqErr(array[int] of var float: A,
14                              array[int] of var float: B) =
15   sumSqErr(A,B)/length(A);
```

Finally, the library also declares a few annotations that provide additional information to the solver such as load_data and vartype, which are discussed in the next section.

The MiningZinc library can be used by adding the following statement.

```
1 include "miningzinc.mzn";
```

The library is written in MiniZinc and is fully compatible with the standard MiniZinc toolchain.

### 2.3 Facilities for loading data

The MiningZinc library also declares a few annotations that allow us to extend the functionality of MiniZinc with respect to loading data from external sources. This consists of two components: (1) accessing data from an external data source and (2) translating it to a data structure supported by MiniZinc.

When using standard MiniZinc, if one wants to use external data, the workflow would be as follows:

1. Determine the relevant information from the database
2. Use SQL to extract this information
3. Translate the data to numeric identifiers using a script
4. Write out the data into MiniZinc format (dzn) using a script
5. Execute MiniZinc
6. Translate the results' identifiers back to the original data using a script
7. Analyze the results and, if necessary, repeat the process

Using the data loading facilities available in MiningZinc, the workflow becomes:

1. Determine the relevant information from the database
2. Update the MiningZinc model with data sources pointing to the relevant information
3. Execute MiningZinc
4. Analyze the results and, if necessary, repeat the process

*Reading data* MiningZinc facilitates loading data from external sources through the load_data( specification ) annotation. The specification is a string describing the data source. By default, MiningZinc supports the following specifications:

**sqlite;<filename>;<SQL query>** Retrieve data from an SQLite database based on an SQL query.
**arff;<filename>** Retrieve data from a file in Weka's ARFF format [18].
**csv;<filename>;<field separator>** Retrieve data from a CSV file.

The use of these annotations is illustrated in Listing 4.

Listing 4: Examples of external data loading

```
1 array [int] of set of int: TDB
      :: load_data("sqlite;data/uci.db;SELECT * FROM zoo;");

2 array [int] of set of int: TDB
      :: load_data("arff;data/zoo.arff;");

3 string: datasource;
4 array [int] of set of int: TDB :: load_data(datasource);
```

The translation process is determined based on the structure of the input data and the target type in MiniZinc. For example, given an input table with two columns and the output type `array[int] of set of int`, the first column is interpreted as the index of the array and the second column as an element of the set. This is illustrated in Figure 2.

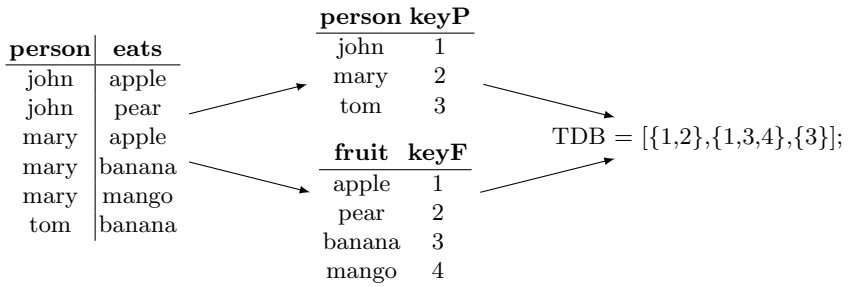


Fig. 2: Default translation to an `array[int] of set of int` from a table with two columns.

*Automatic translations* The previous example shows that during the loading of the data, we need to translate some of the data to an integer range. MiningZinc performs these translations automatically. The user can guide this translation process by adding type annotations to variable definitions. This can be done using the `vartype` annotation as illustrated in Listing 5. The additional information allows MiningZinc to translate the solutions back to the original values.

Listing 5: Examples of type annotations

```
1 array [int] of set of int: TDB
      :: load_data("sqlite;data/uci.db;SELECT * FROM zoo;")
      :: vartype("Animals","Features")

2 var set of Items: Items :: vartype("Features");
3 var set of int: Trans :: vartype("Animals") = cover(Items,TDB);
4 constraint card(cover(Items, TDB)) >= MinFreq;
```

```
5 solve satisfy;

6 output [show(Items), show(Trans)];
```

### 2.4 Python integration

MiniZinc is a language that is specifically designed for expressing constrained optimization problems. It is therefore not very suitable to write complete systems, but should be seen as a domain specific language instead. To facilitate the use of MiningZinc we provide an interface with Python through the mngzn module. This interface allows the user to parse a MiningZinc model from a Python string, provide parameters using Python's dictionaries and query its results as a Python data structure. The main interface is demonstrated in Listing 6.

Listing 6: "Python interface"

```
1 import mngzn
2 modelstr = """
  int: sum; int: max;
  var 0..max: a; var 0..max: b;
  constraint a+b == sum;
  solve satisfy;
  output [show(a), show(b)];
  """
3 params = {'sum': 3, 'max': 2}
4 model = mngzn.parseModel(modelstr, params)
5 solutions = model.solve()
6 for sol in solutions:
7     print model.format_solution(sol)
```

First, we load the MiningZinc package (Line 1) and we define a model as a string of MiniZinc code (Line 2). The model takes two parameters sum and max and finds all pairs of integers up to max that sum to sum. On Line 3 we set the values of these parameters in a Python dictionary. Next, we parse the model string together with the parameters to obtain a solvable model (Line 4). On Line 5 we solve the model and obtain the solutions. This returns a sequence of Python dictionaries containing the output variables of the model, in this case [{'a': 1, 'b': 2}, {'a': 2, 'b': 1}]. Finally, we format and print out each solution (Line 7).

In Section 3.7 (Listing 17) we show an example of how this interface can be used to implement a greedy algorithm.

## 3 Modeling data mining problems

We show how to model a variety of data mining problems, including constraint-based itemset mining, sequence mining, clustering, linear regression, ranked tiling and more.

In each case, the problem is modelled as a standard constraint satisfaction or optimisation problem, and it is modelled using the primitives available in MiniZinc, as well some common functions and predicates that we have added to the MiningZinc library.

### 3.1 Itemset mining

The MiningZinc work originates from our work on using constraint programming (solvers) for constraint-based itemset mining [16].

*Problem statement.* Itemset mining was introduced by Agrawal et al. [1] and can be defined as follows. The input consists of a set of transactions, each of which contains a set of *items*. Transactions are identified by identifiers $\mathcal{S} = \{1, \ldots, n\}$; the set of all items is $\mathcal{I} = \{1, \ldots, m\}$. An itemset database $\mathcal{D}$ maps transaction identifiers to sets of items: $\mathcal{D}(t) \subseteq \mathcal{I}$. The frequent itemset mining problem is then defined mathematically as follows.

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

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

*MiningZinc model.* Listing 7 shows the frequent itemset mining problem in MiningZinc. Lines 2 and 3 are parameters and data, which a user can provide separate from the actual model or through `load_data` statements. The model represents the items and transaction identifiers in $\mathcal{I}$ and $\mathcal{S}$ by natural numbers (from 1 to `NrI` and 1 to `NrT` respectively) and the dataset $\mathcal{D}$ by the array `TDB`, mapping each transaction identifier to the corresponding set of items. The set of items we are looking for is represented on line 5 as a *set variable* with elements between value 1 and `NrI`. The minimum frequency constraint is posted on line 7; it naturally corresponds to the formal notation $|\phi_{\mathcal{D}}(I)| \geq \alpha$. The *cover* relation used on line 7 and shown in Listing 8 is part of the MiningZinc library and implements $\phi_{\mathcal{D}}(I) = \{t | I \subseteq \mathcal{D}(t)\}$; note that this constraint is *not* a hard-coded constraint in the solver, such as in other systems, but is implemented in the MiningZinc language itself, and can hence be changed if this is desired. Finally, line 8 states that it is a satisfaction problem. Enumerating all solutions that satisfy the constraints corresponds to enumerating all frequent itemsets.

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 of the problem. Furthermore, the use of user-defined functions allows us to abstract away concepts that are common when expressing constraint-based mining problems.

Listing 7: "Frequent Itemset mining"

```
1 % Data
2 int: NrI; int: NrT; int: Freq;
3 array[1..NrT] of set of 1..NrI: TDB;
4 % Pattern
5 var set of 1..NrI: Items;
6 % Min. frequency constraint
7 constraint card(cover(Items,TDB)) >= Freq;
8 solve satisfy;
```

Listing 8: "Cover function for itemsets"

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

*Constraint-based mining* In constraint-based mining the idea is to incorporate additional user-constraints into the mining process. Such constraints can be motivated by an overwhelming number of (redundant) results otherwise found, or by application-specific constraints such as searching for patterns with high profit margins in sales data.

Listing 9 shows an example constraint-based mining setting. Compared to Listing 7, two constraints have been added: a *closure* constraint on line 6, which avoids non-closed patterns in the output, and a minimum cost constraint 9, requiring that the sum of the costs of the individual items is above a threshold. Other constraints could be added and combined in a similar way. See [16] for the range of constraints that has been studied in a constraint programming setting.

Listing 9: "Constraint-based itemset 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;
```

### 3.2 Sequence mining

Sequence mining [1] can be seen as a variation of the itemset mining problem discussed above. Whereas in itemset mining each transaction is a set of items, in sequence mining both transactions and patterns are ordered, (i.e. they are sequences instead of sets) and symbols can be repeated. For example, $\langle \mathrm{b}, \mathrm{a}, \mathrm{c}, \mathrm{b} \rangle$ and $\langle \mathrm{a}, \mathrm{c}, \mathrm{c}, \mathrm{b}, \mathrm{b} \rangle$ are two sequences, and the sequence $\langle \mathrm{a}, \mathrm{b} \rangle$ is one possible pattern included in both.

*Problem statement.* Two key concepts in any pattern mining setting are the structure of the pattern, and the *cover* relation that defines when a pattern covers a transaction.

In sequence mining, a transaction is covered by a pattern if there exists an embedding of the sequence pattern in the transaction; where an embedding is a mapping of every symbol in the pattern to the same symbol in the transaction such that the order is respected.

**Definition 2 (Embedding in a sequence).** *Let $S = \langle s_1, \ldots, s_m \rangle$ and $S' = \langle s'_1, \ldots, s'_n \rangle$ be two sequences of size $m$ and $n$ respectively with $m \leq n$. The tuple of integers $e = (e_1, \ldots, e_m)$ is an **embedding** of $S$ in $S'$ (denoted $S \sqsubseteq_e S'$) if and only if:*

$$S \sqsubseteq_e S' \leftrightarrow e_1 < \ldots < e_m \text{ and } \forall i \in 1, \ldots, m : s_i = s'_{e_i} \tag{1}$$

For example, let $S = \langle \mathrm{a}, \mathrm{b} \rangle$ be a pattern, then $(2, 4)$ is an embedding of $S$ in $\langle \mathrm{b}, \mathrm{a}, \mathrm{c}, \mathrm{b} \rangle$ and $(1, 4), (1, 5)$ are both embeddings of $S$ in $\langle \mathrm{a}, \mathrm{c}, \mathrm{c}, \mathrm{b}, \mathrm{b} \rangle$.

Given an alphabet $\Sigma$ of symbols, a *sequential* database $D$ is a set of transactions where each transaction is a sequences defined over symbols in $\Sigma$. As in itemset mining, let $\mathcal{D}$ be a mapping from transaction identifiers to transactions. The frequent sequence mining problem is then defined as follows:

**Definition 3 (Frequent Sequence Mining).** *Given a sequential database $\mathcal{D}$ with alphabet $\Sigma$ and a threshold $\alpha$, the frequent sequence mining problem consists of finding all sequences $S$ over alphabet $\Sigma$ such that $|\psi_{\mathcal{D}}(S)| > \alpha$, where $\psi_{\mathcal{D}}(S) = \{t \mid \exists e \text{ s.t. } S \sqsubseteq_e \mathcal{D}(t)\}$.*

The set $\psi_{\mathcal{D}}(S)$ is the *cover* of the sequence, similar to the cover of an itemset.

*MiningZinc model.* Modeling this in MiningZinc is somewhat more complex than itemset mining, as for itemsets we could reuse the *set* variable type, while sequences and the embedding relation need to be *encoded*. Each symbol is given an identifier (offset 1), and a transaction is represented as an array of symbol

identifiers. The data is hence represented by a two dimensional array, and all sequences are padded with the identifier 0 such that they have the same length (MiniZinc does not support an array of arrays of different length). This data is given in lines (2)–(7) in Listing 10.

The pattern itself is also an array of integers, representing symbol identifiers. The 0 identifier can be used as padding at the end of the pattern, so that patterns of any size can be represented. Line (9) represents the array of integer variables while line (11)–(13) enforce the *padding* meaning of the 0 identifier.

To encode the cover relation we can not quantify over all possible embeddings $e$ explicitly, as there can be an exponential number of them. Instead, we add one array of variables for every transaction that will represent the embedding of the pattern in that transaction, if one exists (line 15). Furthermore, we add one Boolean variable for every transaction, which will indicate whether the embedding is valid, e.g. whether the transaction is covered (line 17). Using these variables, we can encode the cover relation (line 19), explained below, as well as that the number of covered transactions must be larger than the minimum frequency threshold (line 21).

Listing 10: "Frequent sequence mining"

```
1  % Data
2  int: NrS; % number of distinct symbols (symbol identifiers)
3  int: NrPos; % number of positions = maximum transaction size
4  int: NrT;
5  int: Freq;
6  % dataset: 2D array of symbols
7  array[1..NrT,1..NrPos] of 1..NrS: data;


8  % Pattern (0 means 'end of sequence')
9  array[1..NrPos] of var 0..NrS: Seq;

10 % enforce meaning of '0'
11 constraint Seq[1] != 0;
12 constraint forall(j in 1..NrPos-1) (
13         (Seq[j] == 0) -> (Seq[j+1] == 0) );


14 % Helper variables for embeddings (0 means 'no match')
15 array[1..NrT, 1..NrPos] of var 0..NrPos: Emb;

16 % Helper variables for Boolean representation of cover set
17 array[1..NrT] of var bool: Cov;


18 % Constrain cover relation
19 constraint sequence_cover(Seq, Emb, Cov);

20 % Min. frequency constraint
```

```
21 constraint sum(i in 1..NrT) (bool2int(Cov[i])) >= Freq;

22 solve satisfy;
```

The actual formulation of the cover relation is shown in Listing 11; it could be made available as a function returning a *set* variable too. The formulation consists of three parts. In the first part (line 6) we constrain that for each transaction, the $j$th embedding variable must point to a position in the transaction that matches the symbol of the $j$th symbol in the pattern. Note that if no match is possible then the embedding variable will only have symbol 0 in its domain. The second part (line 9) requires that embedding variables must be increasing (except when 0). Finally, on line 12 we state that a transaction is covered if for every non-0 valued position in the pattern there is a matching embedding variable.

Listing 11: "Cover relation for sequences"

```
1  predicate sequence_cover(array[int] of var int: Seq,
2                           array[int,int] of var int: Emb,
3                           array[int] of var bool: Cov) =

4    % Individual positions should match (else: 0)
5    forall(i in 1..NrT, j,x in 1..NrPos) (
6      (Emb[i,j] == x) -> (Seq[j] == data[i,x]) ) /\

7    % Positions increase (except when 0)
8    forall(i in 1..NrT, j in 1..NrPos-1, x in 1..NrPos) (
9      (Emb[i,j+1] == x) -> (Emb[i,j] < x) ) /\

10   % Covered if all its positions match
11   forall(i in 1..NrT) (
12     Cov[i] <-> forall(j in 1..NrT) ( (Seq[j] != 0) ->
13                                      (Emb[i,j] != 0) ));
```

As in sequence mining, extra constraints can be added to extract fewer, but more relevant or interesting patterns. An overview of sequence mining constraints that have been studied in a sequence mining setting is available in [31].

### 3.3 Constraint-based pattern mining in Bayesian networks

Just as one can mine patterns in data, it is also possible to mine patterns in Bayesian Networks (BNs). These patterns can help in understanding the knowledge that resides in the network. Extracting such knowledge can be useful when trying to better understand a network, for example when presented with a new network, in case of large and complex networks or when it is updated frequently.

*Problem statement.* A Bayesian Network $\mathcal{G}$ defines a probability distribution over a set of random variables $\mathcal{X}$. Each variable has a set of values it can take,

called its *domain*. We define a Bayesian network pattern as an assignment to a subset of the variables:

**Definition 4 (BN pattern).** *A pattern $A$ over a Bayesian network $\mathcal{G}$ is a partial assignment, that is, an assignment to a subset of the variables in $\mathcal{G}$: $A = \{(X_1 = x_1), \ldots, (X_m = x_m)\}$, where each $X_i$ is a different variables and $x_i$ is a possible value in its domain.*

A BN pattern can be seen as an itemset, where each item is an assignment of a variable to a value. One can compute the (relative) *frequency* of an itemset in a database; related, for a BN pattern one can compute the *probability* of the pattern in the Bayesian network. The probability of a pattern $A$, denoted by $P_{\mathcal{G}}(A)$, is $P((X_1 = x_1), \ldots, (X_m = x_m))$, that is, the probability of the partial assignment marginalized over the unassigned variables.

Given this problem setting, one can define a range of constraint-based pattern mining tasks over Bayesian Networks, similar to constraint-based mining tasks over itemsets or sequences. In line with frequent itemset mining, the following defines the probable BN pattern mining problem:

**Definition 5 (Probable BN pattern Mining).** *Given a Bayesian network $\mathcal{G}$ over variables $\mathcal{X}$ and a threshold $\alpha$, the probable BN pattern mining problem consists of finding all BN patterns $A$ over $\mathcal{X}$ such that $P_{\mathcal{G}}(A) > \alpha$.*

*MiningZinc model.* We encode a BN pattern with an array of integer CP variables, as many as there are random variables in the BN. Each CP variable has $m$+1 possible values, where $m$ is the number of values in the domain of the corresponding random variable: value 0 represents that the variable is not part of the partial assignment, e.g. it should be marginalized over when computing the probability. The other values each correspond to a value the domain of the random variable.

The main issue is then to encode the probability computation. As this computation will be performed many times during search, we choose to first compile the BN into an *arithmetic circuit*. Computing the probability over the circuit is polynomial in its size (which may be worst-case exponential to the size of the origin network) [8]. Nevertheless, using ACs is generally recognized as one of the most effective techniques for exact computation of probabilities, especially when doing so repeatedly.

Figure 3 shows an example AC. It consists of product nodes, sum nodes, constants and *indicator variables* $\lambda$. The Boolean indicator variables $\lambda_{i,j}$ indicate whether $(X_i = j)$. An assignment sets the corresponding indicator variable to 1 and the other indicator variables of the random variable to 0. To marginalize a random variable away, all indicator variables of that variable must be set to 1. The value obtained in the root node after evaluating the arithmetic circuit corresponds to the probability of the pattern given the assignment to the indicator variable nodes.

To encode this in CP, we will create a float variable for every node in the AC. Listing 12 shows how to encode an AC in CP. We assume given the set of

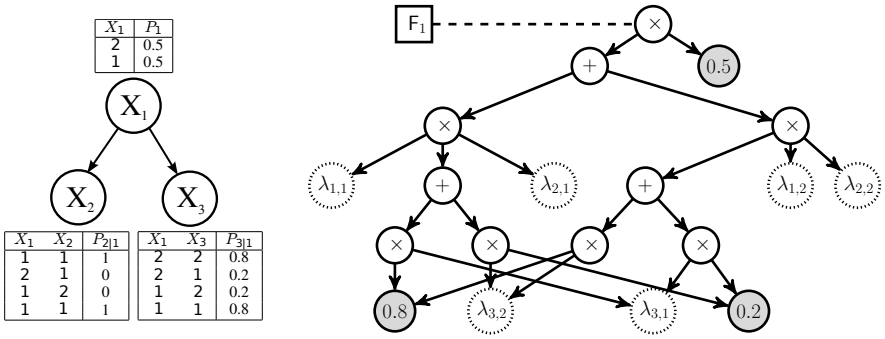


Fig. 3: Left: The Bayesian network and CPTs for a distribution with 3 variables. The domain for all variables is $\{1, 2\}$. Right: The compiled arithmetic circuit for this BN. A $\lambda_{ij}$ leaf of in the AC corresponds to assignment $X_i = j$ in the BN.

product and sum node identifiers (root node has identifier 1), an array of sets representing the 'children' relation, an array with the constants, and a 2D array that maps the indicator variables to nodes of the tree. The last two constraints in the listing provide a mapping from the $Q$ variables to indicator variables, such that they must all be set to 1 (=marginalize) or exclusively have one indicator set to 1 (=assignment).

Listing 12: Probable BN pattern mining

```
1  int: num_vars;
2  array[variables] of int: num_values;
3  float: min_prob;

4  array[1..num_vars] of int: Q;
5  var 0.0..1.0: P;

6  constraint P > min_prob;

7  constraint forall(i in 1..num_vars) (
8    0 <= Q[i] /\ Q[i] <= num_values[i]+1 );

9  % encode AC
10 int: num_ACnodes;
11 array[1..num_ACnodes] of var 0.0..1.0: F;
12 constraint P = F[1]; % root node

13 % sum and product nodes
14 array[1..num_ACnodes] of set of int: children;
15 set of int: sum_nodes;
16 constraint forall(i in sum_nodes) (
17   F[i] = sum(j in children[i]) (F[j]) );
18 set of int: prod_nodes;
19 constraint forall(i in prod_nodes) (
```

```
20   F[i] = product(j in children[i]) (F[j]) );

21 % constant nodes, -1 means non-constant
22 array[1..num_ACnodes] of int: constants;
23 constraint forall(i in 1..num_ACnodes where constants[i]!=-1) (
24   F[i] = constants[i] );

25 % Q to indicator nodes (which must take either 0 or 1)
26 array[1..num_vars,int] of int: mapQ;
27 constraint forall(i in 1..num_vars) (
28   Q[i]=0 -> forall(j in 1..num_vals[i]) (F[mapQ[i,j]] == 1) );
29 constraint forall(i in 1..num_vars, k in 1..num_vals[i]) (
30   Q[i]=k -> (F[mapQ[i,k]] == 1) /\
31             forall(j in 1..num_vals[i] where j!=k)
32               (F[mapQ[i,j]] == 0) );

33 solve satisfy;
```

Many constraints from the itemset mining literature have a counterpart over BN patterns and can be formulated as well, such as size constraints, closed/maximal/free constraints and constraints to discriminate results from two networks from each other, or to discriminate a probability in a network to relative frequency in a dataset. This is currently work in progress.

### 3.4 Linear regression

A common problem studied in data mining is regression, where the goal is to estimate the value of a variable based on the values of dependent variables.

*Problem statement.* In simple regression, the goal is to predict the value of a *target* attribute given the value of an $M$-dimensional vector of *input* variables $x = (x_1, ..., x_M)$.

We are given a set of $N$ observations $\mathbf{X}$ and their corresponding target values $y$. The goal is to find a function $\hat{y}(x)$ that approximates the target values $y$ for the given observations. In linear regression [3] we assume $\hat{y}(x)$ to be a linear function. Mathematically, such a function can be formulated as

$$\hat{y}(x) = w_1 x_1 + ... + w_M x_M + w_{M+1}$$

where $w = (w_1, ..., w_{M+1})$ is a vector of weights that minimizes a given error function. This error function is typically defined as the sum of squared errors

$$sumSqErr(\hat{y}, y) = \|\mathbf{X}w - y\|_2^2,$$

where $\mathbf{X}$ is an $N \times (M+1)$ matrix where each row corresponds to a given observation (extended with the constant 1), and $y$ is a vector of length $N$ containing the corresponding target values. The vector $\mathbf{X}w$ contains the result of computing $\hat{y}$ for each observation. The goal is then to find the vector of weights $w$ that

minimizes this error, that is,

$$\arg\min_{w} \|\mathbf{X}w - y\|_2^2.$$

*MiningZinc model.* We can formulate this problem as an optimization problem as shown in Listing 13. The model starts with defining the input data (Lines 2 and 3) and its dimensions (Lines 5-6). The input data can be specified in an external file. Line 9 specifies the weight vector that needs to be found. Based on this weight vector and the input variable $x$, we specify the estimate ($\hat{y}(x)$) of the linear model in Line 11. Finally, on Line 13 we specify the optimization criterion, i.e. to minimize the sum of squared errors. The function sumSqErr is defined in the MiningZinc library (Listing 3).

Listing 13: Model for min-squared-error linear regression

```
1  % Observations
2  array[int, int] of float: X;
3  array[int] of float: y;
4  % Data dimensions
5  set of int: Points = indexset_1of2(X);
6  set of int: Dimensions = indexset_2of2(X);
7  int: NumWeights = max(Dimensions)+1;
8  % Weights to find
9  array[1..NumWeights] of var float: w;

10 % Estimate for each data point
11 array[Points] of var float: yh =
       [ sum(j in Dimensions) (w[j]*X[i,j]) + w[NumWeights]
                                                | i in Points ];
12 % Optimization criterium
13 solve minimize sumSqErr(y, yh);
```

By replacing the error function we can easily model other linear regression tasks, for example linear regression with *elastic net* regularization [38] where the optimization criterium is defined as

$$\arg\min_{w} \frac{1}{2n_{samples}} \|\mathbf{X}w - y\|_2^2 + \alpha\rho\|w\|_1 + \frac{\alpha(1-\rho)}{2}\|w\|_2^2$$

with $\alpha$ and $\rho$ parameters that determine the trade-off between L1 and L2 regularization. Listing 14 shows the implementation of this scoring function in MiniZinc.

Listing 14: Elastic net error function for linear regression

```
1  function var float: elastic_net(
               array[int] of float: Y,
               array[int] of var float: E,
               array[int] of var float: W,
               float: Alpha, float: Rho) =
```

```
        (0.5 / int2float(length(Y))) *
          norm2([ Est[i] - Y[i] | i in indexset(Y)])
        + (Alpha*Rho) * norm2( W )
        + (0.5*Alpha*(1.0-Rho)) * norm1( W );
```

### 3.5 Clustering

The task of clustering is already discussed in Chapter **??**. We would like to point out that the clustering problems explained there can be modeled in MiningZinc too.

*Problem statement.* Let us consider the minimum sum of squared error clustering problem (which $k$-means provides an approximation of), where the goal is to group all examples into $k$ non-overlapping groups [21]. The objective to minimize is the 'error' of each cluster, that is, the distance of each point in the cluster to the mean (centroid) of that cluster.

The *centroid* of a cluster can be computed by computing the mean of the data points that belong to it:

$$z_C = \text{mean}(C) = \frac{\sum_{p \in C} p}{|C|} \tag{2}$$

The error is then measured as the sum of squared errors of the clusters:

$$\sum_{C \in \mathcal{C}} \sum_{p \in C} d^2(p, z_C) \tag{3}$$

*MiningZinc model.* The model below shows a MiningZinc specification of this problem. As variables, it uses an array of integers, one integer variable for every example. This variable will indicate which cluster the example belongs too. The optimisation criterion is specified over all clusters and examples; the b2f(Belongs[j])==i) part converts the Boolean valuation of whether point $j$ belongs to cluster $i$ into a float variable, such that this indicator variable can be multiplied by the sum of squared errors of point $j$ to cluster $i$.

The functions b2f() (bool 2 float) and sumSqErr() (sum of squared errors) are part of the MiningZinc library, see Section 2.2. The definition of mean() is shown below and follows the mathematical definition above.

```
1  % Data
2  int: NrDim; % number of dimensions
3  int: NrE; % number of examples
4  int: K; % number of clusters
5  array [1..NrE,1..NrDim] of float: Data;

6  % Clustering (each point belongs to one cluster)
7  array[1..NrE] of var 1..K: Belongs;
```

```
8  solve minimize sum(i in 1..K, j in 1..NrE) (
9                    b2f(Belongs[j] == i)*
10                   sumSqErr(Data[j], mean(Data, Belongs, i))
11                 );


12 function array[int] of var float: mean(
13                       array[int,int] of var float: Data,
14                       array[int] of var int: Belongs,
15                       int: c) =
16   let {
17     set of int: Exs = index_set_1of2(Data),
18     set of int: Dims = index_set_2of2(Data),
19     array[Dims] of var float: Mean,
20     constraint forall( d in Dims ) (
21       Mean[d] =
22         sum(i in Exs)( b2f(Belongs[i] == c) * Data[i,d] ) /
23         sum(i in Exs)( b2f(Belongs[i] == c) )
24     )
25   } in Mean;
```

More clustering problems and how to model them for use with constraint solvers can be found in chapter **??**.

### 3.6 Relational data factorization

$$\underset{\mathbf{A}}{\begin{vmatrix} 1 & 1 & 0 \\ 1 & 1 & 1 \\ 1 & 0 & 1 \end{vmatrix}} = \underset{\mathbf{B}}{\begin{vmatrix} 1 & 0 \\ 1 & 1 \\ 0 & 1 \end{vmatrix}} \times \underset{\mathbf{C}}{\begin{vmatrix} 1 & 1 & 0 \\ 0 & 1 & 1 \end{vmatrix}}$$

Fig. 4: Boolean matrix factorization.

Motivated by an analogy with Boolean matrix factorization [29] (cf. Figure 4), [9] introduces the problem of factorizing a relation in a database. In matrix factorization, one is given a matrix and has to factorize it as a product of other matrices.

*Problem statement.* In relational data factorization (RDF), the task is to factorize a given relation as a conjunctive query over other relations, i.e., as a combination of natural join operations. The problem is then to compute the extensions of these relations starting from the input relation and a conjunctive query. Thus relational data factorization is a form of both *inverse* querying and *abduction*, as one has to compute the relations in the query from the result of the query. The result of relational data factorization is not necessarily unique,

constraints on the desired factorization can be imposed and a scoring function is used to determine the quality of a factorization. Relational data factorization is thus a constraint satisfaction and optimization problem.

More specifically, relational data factorization is a generalization of abductive reasoning [10]:

1. instead of working with a single observation $f$, we now assume a set of facts $D$ for a unique target predicate $db$ is given;
2. instead of assuming any definition for the target predicate, we assume a single definite rule defines $db$ in terms of a set of abducibles $A$, the conjunctive query;
3. instead of assuming that a minimal set of facts be abduced, we score the different solutions based on the observed error.

Formally we can specify the relational factorization problem as follows:
**Given:**

- a dataset $D$ (of ground facts for a target predicate $db$);
- a factorization shape $Q$: $db(\bar{T}) \leftarrow q_1(\bar{T}_1), \ldots, q_k(\bar{T}_k)$, where some of the $q_i$ are abducibles;
- a set of rules and constraints $P$;
- a scoring function *opt*.

**Find**: the set of ground facts $F$, the extensions of relation $Q$, that scores best w.r.t. $opt(D, approx(P, Q, F))$ and for which $Q \cup P \cup F$ is consistent.

*MiningZinc model.* Listing 15 shows the model for a relational factorization problem with a ternary conjunctive query, using the sum of absolute errors as scoring function and without additional constraints.

Listing 15: Relational decomposition

```
% Input data
array[int, int, int] of int: paper;
% index set of authors
set of int: Authors = index_set_1of3(paper);
% index set of universities
set of int: Universities = index_set_2of3(paper);
% index set of venues
set of int: Venues = index_set_3of3(paper);

% Search for
array[Authors, Universities] of var bool: worksAt;
array[Authors, Venues] of var bool: publishesAt;
array[Universities, Venues] of var bool: knownAt;

solve minimize
  sum(a in Authors, u in Universities, v in Venues) (
     abs(paper[a,u,v] - bool2int(
        worksAt[a,u] /\ publishesAt[a,v] /\ knownAt[u,v])) );
```

```
14 output [ show(worksAt), show(publishesAt), show(knownAt) ];
```

### 3.7 Ranked tiling

Ranked tiling was introduced in [23] to find interesting areas in ranked data. In this data, each transaction defines a complete ranking of the columns. Ranked data occurs naturally in applications like sports or other competitions. It is also a useful abstraction when dealing with numeric data in which the rows are incomparable.

*Problem statement.* Ranked tiling discovers regions that have high average rank scores in rank matrices. These regions are called *ranked tiles*. Formally, a rank tile is defined by the following optimization problem:

**Problem 1 (Maximal ranked tile mining)** *Given a rank matrix* $\mathcal{M} \in \sigma^{m \times n}$, $\sigma \in \{1, \dots n\}$ *and a threshold* $\theta$*, find the ranked tile* $B = (R^*, C^*)$*, with* $R^* \subseteq \{1 \dots m\}$ *and* $C^* \subseteq \{1 \dots n\}$*, such that:*

$$B = (R^*, C^*) = \operatorname*{argmax}_{R,C} \sum_{r \in R, c \in C} (\mathcal{M}_{r,c} - \theta). \tag{4}$$

*where* $\theta$ *is an absolute-valued threshold.*

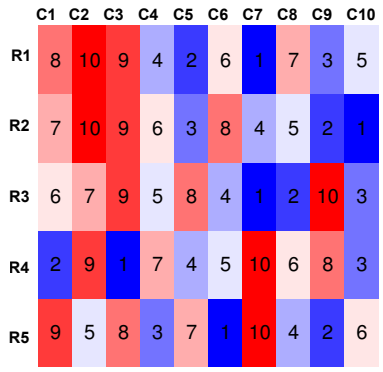


Fig. 5: Example rank matrix, with maximal ranked tile $B = (\{R1, R2, R3, R5\}, \{C1, C2, C3\})$.

*Example 1.* Figure 5 depicts a rank matrix containing five rows and ten columns. When $\theta = 5$, the maximal ranked tile is defined by $R = \{1, 2, 3, 5\}$ and $C = \{1, 2, 3\}$. The score obtained by this tile is 37, and no more columns or rows can be added without decreasing the score.

The maximal ranked tiling problem aims to find a single tile, but we are also interested in finding a set of such tiles. This maximizes the amount of information we can get from the data. In other words, we would like to discover a ranked tiling.

**Problem 2 (Ranked tiling)** *Given a rank matrix $\mathcal{M}$, a number $k$, a threshold $\theta$, and a penalty term $P$, the ranked tiling problem is to find a set of ranked tiles $B_i = (R_i, C_i),\ i = 1 \dots k$, such that they together maximize the following objective function:*

$$\underset{R_i, C_i}{\arg\max} \sum_{r \in \mathcal{R}, c \in \mathcal{C}} \mathbb{1}_{(t_{r,c} \geq 1)}((\mathcal{M}_{r,c} - \theta) - (t_{r,c} - 1)P) \tag{5}$$

*where $t_{r,c} = |\{i \in \{1, \dots, k\} \mid r \in R_i, c \in C_i\}|$ indicates the number of tiles that cover a cell, and $\mathbb{1}_\varphi$ is an indicator function that returns 1 if the test $\varphi$ is true, and 0 otherwise. $P$ indicates a penalty that is assigned when tiles overlap.*

To solve Problem 1 efficiently, we introduce two Boolean decision vectors: $T = (T_1, T_2, ..., T_m)$, with $T_i \in \{0, 1\}$, for rows and $I = (I_1, I_2, ..., I_n)$, with $I_i \in \{0, 1\}$, for columns. An assignment to the Boolean vectors $T$ and $I$ corresponds to an indication of rows and columns belonging to a tile. Then, the maximal ranked tile can be found by solving the following equivalent problem:

$$\underset{T, I}{\arg\max} \sum_{t \in \mathcal{R}} T_t * (\sum_{i \in \mathcal{C}} (\mathcal{M}_{t,i} - \theta) * I_i) \tag{6}$$

subject to

$$\forall t \in \mathcal{R} : T_t = 1 \leftrightarrow \sum_{i \in \mathcal{C}} (\mathcal{M}_{t,i} - \theta) * I_i \geq 0 \tag{7}$$

$$\forall i \in \mathcal{C} : I_i = 1 \leftrightarrow \sum_{t \in \mathcal{R}} (\mathcal{M}_{t,i} - \theta) * T_t \geq 0 \tag{8}$$

where redundant constraints (7), (8) are introduced to boost the search.

*MiningZinc model for finding a single tile.* This problem specification translates directly into the MiningZinc model shown in Listing 16 where Equation 7 and 8 correspond to lines 7 and 8, respectively, and the optimization criterion of Equation 6 corresponds to line 9.

Listing 16: MiniZinc model for finding a single best tile

```
1 array[int, int] of int: TDB;
2 int: th;  % Theta
3 set of int: Rows = index_set_1of2(TDB);
4 set of int: Cols = index_set_2of2(TDB);

5 array [Cols] of var bool: I;
6 array [Rows] of var bool: T;
```

```
7  constraint forall (r in Rows) (
     T[r] == (sum(c in Cols)((TDB[r,c]-th)*bool2int(I[c])) >= 0)
   );

8  constraint forall (c in Cols) (
     I[c] == (sum(r in Rows)((TDB[r,c]-th)*bool2int(T[r]))>=0)
   );

9  solve maximize
    sum(r in Rows)(
       bool2int(I[r])*sum(c in Cols)((TDB[r,c]-th)*bool2int(T[c])
    )
   );

10 output [ show(T), "\n", show(I), "\n"];
```

To solve Problem 2, Le Van et. al. [23] propose to approximate the optimal solution by using a greedy approach, as is common for this type of pattern set mining problem. The first tile is found by solving the optimization problem in Listing 16. Next, we remove that tile by setting all cells in the matrix that are covered to the lowest rank (or another value, depending on parameter $P$). Then, we search in the resulting matrix for the second tile. This process is repeated until the number of desired tiles is found. The sum of the scores of all discovered tiles will correspond to the score of Equation 5 for this solution. However, as the search is greedy, the solution is not necessarily optimal.

*Python wrapper for greedy tile mining.* The greedy approach cannot be modelled directly in the MiningZinc language. However, the MiningZinc framework allows direct access to the solving infrastructure from Python. The complete algorithm is shown in Listing 17. The interaction with the MiningZinc module (`mngzn`) occurs on line 7 where the model is initialized, and on line 9 where one solution of the model is retrieved. In lines 13 through 19 the obtained tile is removed from the original matrix (by setting its entries to 0). The process is repeated until the tile no longer covers a new entry of the matrix.

Listing 17: Wrapper for finding all tiles (in Python)

```
1  TDB = ...  # Load matrix
2  mznmodel = ...  # See Listing 16
3  params = {'TDB': TDB, 'th': 5}
4  tiles = []
5  stop = False
6  while not stop :
7      model = mngzn.parseModel(mznmodel, params)
8      # Solve the model to find one tile
9      solution = next(model.solve())
10     tiles.append(solution)
11     stop = True
```

```
12      # Update the ranking matrix => zero out values
13      for i,r in enumerate(solution['T']) :
14          for j,c in enumerate(solution['I']) :
15              if r and c :
16                  # Stop unless the new tile covers a new item
17                  if TDB[i][j] > 0 :
18                      TDB[i][j] = 0
19                      stop = False
20  return tiles
```

## 4 Related Work

We have shown how MiningZinc can be used to model a wide variation of data mining and machine learning tasks in a high-level and declarative way. Our modeling language is based on MiniZinc [32] because it is a well-developed existing language with wide support in the CP community, it supports user-defined constraint, and is solver-independent. Other modeling languages such as Essence [12], Comet [37] and OPL [36] have no, or only limited, support for building libraries of user-defined constraints, and/or are tied to a specific solver.

Integrating declarative modeling and data mining has been studied before in the case of itemset mining [16, 22], clustering [27, 11] and sequence mining [31]. However, these approaches were *low-level* and *solver dependent*. The use of higher-level modeling languages and primitives has been studied before [28, 17], though again tied to one particular solving technology.

The idea of combining multiple types of data mining and machine learning techniques also lies at the basis of machine learning packages such as WEKA [18] and scikit-learn [33]. However, these packages do not offer a unified declarative language and they do not support going beyond the capabilities of the algorithms offered.

In data mining, our work is related to that on inductive databases [24]; these are databases in which both data and patterns can be queried. Most inductive query languages, e.g., [26, 20], 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 [4], 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 realized using a traditional mining algorithm. In MiningZinc we support the integration of data from an external database through the use of SQL queries directly.

## 5 Solving

This chapter does not expand on solving, but the MiningZinc framework [14] supports three types of solving: 1) to use an existing MiniZinc solver; 2) to de-

tect that the specified tasks is a standard known task and to use a specialised algorithm to solve it; and 3) a hybrid solving approach that uses both specialised algorithms and generic constraint solvers, for example by solving a master problem and subproblem with different technology, or to incorporate specialised algorithms inside *global* constraint propagators. The first approach is typically least efficient but most flexible towards adding extra constraints. The second approach is least flexible but typically most scalable. The third, hybrid, approach offers a trade-off between generality and efficiency, but requires modifications to the solving process, which is hence beyond what can be expressed in a modeling language like Mini(ng)Zinc.

## 6 Conclusion

In this chapter we showed how a wide range of data mining problems can be modeled in MiningZinc. Only a minimal library of extra predicates and functions was needed to express these problems, meaning that standard MiniZinc is often sufficient to model such problem. Two additions are the ability to load data from a database, and a library of distance functions, which are often used in data mining.

The key feature of MiningZinc as a language for expressing data mining problems is the ability to add and modify constraints and objective functions. Hence constraint-based mining problems are those where the language and framework has most to offer, such as in constraint-based pattern mining and constrained clustering. Another valuable use is for prototyping new data mining problems, as was done for relational data factorization and ranked tiling. Many other problem settings are yet unexplored.

## References


1. Agrawal, R., Imielinski, T., Swami, A.N.: Mining association rules between sets of items in large databases. In: Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data. pp. 207–216. ACM Press (1993)
2. Basu, S., Davidson, I., Wagstaff, K.: Constrained clustering: advances in algorithms, theory, and applications. Chapman & Hall/CRC data mining and knowledge discovery series, CRC Press (2008)
3. Bishop, C.M.: Pattern Recognition and Machine Learning. Springer (2006)
4. Blockeel, H., Calders, T., Fromont, É., Goethals, B., Prado, A., Robardet, C.: An inductive database system based on virtual mining views. Data Min. Knowl. Discov. 24(1), 247–287 (2012)
5. Boulicaut, J.F., Dzeroski, S. (eds.): Proceedings of the Second International Workshop on Inductive Databases, 22 September, Cavtat-Dubrovnik, Croatia. Rudjer Boskovic Institute, Zagreb, Croatia (2003)
6. Boulicaut, J.F., Raedt, L.D., Mannila, H. (eds.): Constraint-Based Mining and Inductive Databases, European Workshop on Inductive Databases and Constraint Based Mining, Hinterzarten, Germany, March 11-13, 2004, Revised Selected Papers, Lecture Notes in Computer Science, vol. 3848. Springer (2005)

7. Coquery, E., Jabbour, S., Sais, L., Salhi, Y., 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, pp. 258–263 (2012)
8. Darwiche, A.: A differential approach to inference in bayesian networks. J. ACM 50(3), 280–305 (2003), http://doi.acm.org/10.1145/765568.765570
9. De Raedt, L., Paramonov, S., van Leeuwen, M.: Relational decomposition using Answer Set Programming. In: Online Preprints 23rd International Conference on Inductive Logic Programming, International Conference on Inductive Logic Programming, Rio de Janeiro, 28-30 August 2013 (Aug 2013), https://lirias.kuleuven.be/handle/123456789/439287
10. Denecker, M., Kakas, A.: Abduction in logic programming. In: Kakas, A., Sadri, F. (eds.) Computational Logic: Logic Programming and Beyond, Lecture Notes in Computer Science, vol. 2407, pp. 402–436. Springer Berlin Heidelberg (2002)
11. Duong, K.C., Vrain, C., et al.: A declarative framework for constrained clustering. In: ECML/PKDD, Machine Learning and Knowledge Discovery in Databases, pp. 419–434. Springer (2013)
12. Frisch, A., Harvey, W., Jefferson, C., Hernández, B.M., Miguel, I.: Essence: A constraint language for specifying combinatorial problems. Constraints 13(3), 268–306 (2008)
13. Gilpin, S., Davidson, I.N.: Incorporating SAT solvers into hierarchical clustering algorithms: an efficient and flexible approach. In: Proceedings of the 17th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, San Diego, CA, USA, August 21-24, 2011. pp. 1136–1144 (2011)
14. Guns, T., Dries, A., Tack, G., Nijssen, S., De Raedt, L.: MiningZinc: A modeling language for constraint-based mining. In: Proceedings of the Twenty-Third International Joint Conference on Artificial Intelligence. pp. 1365–1372. AAAI Press (Aug 2013)
15. Guns, T., Dries, A., Tack, G., Nijssen, S., Raedt, L.D.: Miningzinc: A language for constraint-based mining. In: International Joint Conference on Artificial Intelligence (2013)
16. Guns, T., Nijssen, S., De Raedt, L.: Itemset mining: A constraint programming perspective. Artif. Intell. 175(12-13), 1951–1983 (2011)
17. Guns, T., Nijssen, S., De Raedt, L.: k-Pattern set mining under constraints. IEEE Transactions on Knowledge and Data Engineering 25(2), 402–418 (Feb 2013)
18. Hall, M., Frank, E., Holmes, G., Pfahringer, B., Reutemann, P., Witten, I.H.: The weka data mining software: An update. SIGKDD Explorations 11(1) (2009)
19. Han, J., Kamber, M.: Data Mining: Concepts and Techniques. Morgan Kaufmann (2000)
20. Imielinski, T., Virmani, A.: MSQL: A query language for database mining. Data Mining and Knowledge Discovery 3, 373–408 (1999)
21. Jain, A.K., Murty, M.N., Flynn, P.J.: Data clustering: A review. ACM Comput. Surv. 31(3), 264–323 (Sep 1999), http://doi.acm.org/10.1145/331499.331504
22. Järvisalo, M.: Itemset mining as a challenge application for answer set enumeration. In: Logic Programming and Nonmonotonic Reasoning. LNCS, vol. 6645, pp. 304–310. Springer (2011)
23. Le Van, T., van Leeuwen, M., Nijssen, S., Fierro Gutiérrez, A.C.E., Marchal, K., De Raedt, L.: Ranked tiling. In: Calders, T., Esposito, F., Hüllermeier, E., Meo, R. (eds.) Proceedings of the European Conference on Machine Learning and Principles and Practice of Knowledge Discovery in Databases (ECML PKDD) 2014, The European Conference on Machine Learning and Principles and Practice of Knowledge

Discovery in Databases (ECMLPKDD), Nancy, France, 15-19 September 2014. pp. 98–113. Springer (2014), https://lirias.kuleuven.be/handle/123456789/457022
24. Mannila, H.: Inductive databases and condensed representations for data mining. In: ILPS. pp. 21–30 (1997)
25. Marriott, K., Nethercote, N., Rafeh, R., Stuckey, P.J., Garcia De La Banda, M., Wallace, M.: The design of the Zinc modelling language. Constraints 13(3), 229–267 (Sep 2008)
26. Meo, R., Psaila, G., Ceri, S.: A new SQL-like operator for mining association rules. In: VLDB. pp. 122–133 (1996)
27. Métivier, J.P., Boizumault, P., Crémilleux, B., Khiari, M., Loudni, S.: Constrained clustering using SAT. In: Advances in Intelligent Data Analysis XI, pp. 207–218. Springer (2012)
28. Métivier, J.P., Boizumault, P., Crémilleux, B., Khiari, M., Loudni, S.: A constraint language for declarative pattern discovery. pp. 119–125. SAC '12, ACM (2012), http://doi.acm.org/10.1145/2245276.2245302
29. Miettinen, P., Mielikäinen, T., Gionis, A., Das, G., Mannila, H.: The discrete basis problem. IEEE Transactions on Knowledge and Data Engineering 20(10), 1348–1362 (October 2008)
30. Mitchell, T.: Machine Learning. McGraw-Hill, first edn. (1997)
31. Negrevergne, B., Guns, T.: Constraint-based sequence mining using constraint programming. In: Integration of AI and OR Techniques in Constraint Programming - 12th International Conference, CPAIOR 2015, Barcelona, Spain, May 18-22, 2015, Proceedings. pp. 288–305 (2015)
32. Nethercote, N., Stuckey, P.J., Becket, R., Brand, S., Duck, G.J., Tack, G.: MiniZinc: Towards a standard CP modelling language. In: CP. LNCS, vol. 4741, pp. 529–543. Springer (2007)
33. Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., Vanderplas, J., Passos, A., Cournapeau, D., Brucher, M., Perrot, M., Duchesnay, E.: Scikit-learn: Machine learning in Python. Journal of Machine Learning Research 12, 2825–2830 (2011)
34. Stuckey, P., Tack, G.: MiniZinc with functions. In: Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems, LNCS, vol. 7874, pp. 268–283. Springer Berlin Heidelberg (2013)
35. Tan, P.N., Steinbach, M., Kumar, V.: Introduction to Data Mining. Addison-Wesley (2005)
36. Van Hentenryck, P.: The OPL optimization programming language. MIT Press (1999)
37. Van Hentenryck, P., Michel, L.: Constraint-Based Local Search. MIT Press (2005)
38. Zou, H., Hastie, T.: Regularization and variable selection via the elastic net. Journal of the Royal Statistical Society, Series B 67, 301–320 (2005)