From inductive databases to a modeling language for pattern mining, to a next-generation library for constraint solving

Tias Guns[1,*]

[1] KU Leuven, Belgium

Abstract

At the occasion of the 20th anniversary of the KDID workshop, I share the tale of how the *inductive databases* concept has inspired our work on MiningZinc, a constraint-based language for pattern mining; and how even today it influences how we are building a next-generation modeling library for constraint solving.

1. From inductive databases to MiningZinc

The inductive databases idea [1] called for systems to 1) support both traditional (SQL) database queries as well as knowledge discovery (KDD) queries, 2) that the output of both queries should be basic objects that can then be used in further queries (the closure principle), and 3) that in such nested cases, conditions from one query could flow into the processing of the next.

At the time, there were multiple query-language extensions proposed that aimed at putting KDD queries in database management systems. They were often thin layers that expose the parameters of a KDD system, an itemset mining or rule learning system, as attributes that could be constrained in the WHERE part of the KDD query.

We felt that it did not support concept 3) sufficiently, that it does not allow for more complex WHERE conditions which the system would be able to take into account.

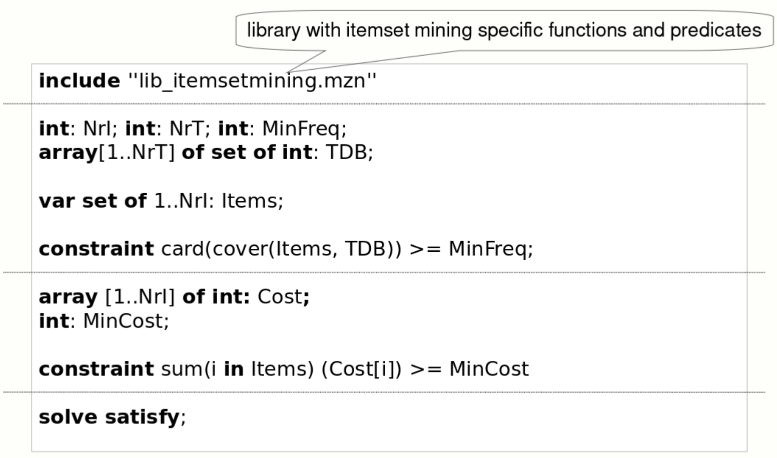
During my PhD I developed the use of Constraint Programming for solving Constraint-based Itemset Mining problems [2]. In the constraint programming literature, there is a rich study of *modeling languages*. A modeling language is a declarative algebraic language that allows to specify constraints over (integer or Boolean) decision variables, and to compute one solution, enumerate all or optimize an objective function.

MiningZinc To better support concept 3), instead of extending SQL with complex KDD queries, we decided to investigate the use of algebraic modeling languages to express complex

KDID 2022: 20th anniversary of KDID Workshop

[*] Research supported by the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (Grant No. 101002802, CHAT-Opt).

tias.guns@kuleuven.be (T. Guns)
https://people.cs.kuleuven.be/~tias.guns/ (T. Guns)
0000-0002-2156-2155 (T. Guns)



library with itemset mining specific functions and predicates

```
include "lib_itemsetmining.mzn"

int: NrI; int: NrT; int: MinFreq;
array[1..NrT] of set of int: TDB;

var set of 1..NrI: Items;

constraint card(cover(Items, TDB)) >= MinFreq;

array [1..NrI] of int: Cost;
int: MinCost;

constraint sum(i in Items) (Cost[i]) >= MinCost

solve satisfy;
```

Figure 1: MiningZinc model: frequent itemset mining with costs.

KDD queries. Examples of complex KDD queries include aggregation and weigted sum constraints over the items or transactions. Figure 1 shows an example model in this language, called Mini**ng**Zinc and based on the popular MiniZinc constraint solving language.

This clearly is a declarative language, though not a query language. Still, it allows to analyse the specified model and determine how it should be evaluated. In MiningZinc, our model analyzer generates multiple *execution plans*, just like a database management system, which can be grouped into 3 categories: a) directly translating the specified problem to a specialized itemset mining that supports all specified constraints; b) identifying that the core of the problem is supported by specialized itemset mining algorithms but that some constraints are not, where the system then uses the algorithm to enumerate patterns and filters that output using the additional constraints; and c) translating the specified model to a generic constraint solver. These execution plans were then ranked using a heuristic estimating the efficiency of the plan, and the top one was executed.

While it supported concept 3), nesting queries and their processing, really well, it left concept 1) and 2) open, namely: where does the data come from, and where does it go to.

The text-based language was great for query optimisations at the KDD query level (what the possible execution plans are, and which one is expected to be most efficient). However it did not satisfy the closure principle, as data loading and post-processing had to be written in a procedural programming language. Actually, looking back, most of our effort in developing the system was in developing glue code to connect different systems together. And even more glue code would have been needed to support generic data loading and post-processing.

2. From MiningZinc to a constraint modeling language

Today, the MiniZinc language that we had built upon is still the most popular modeling language for modeling Constraint Programming problems. However, the questions: where does the data come from (concept 1) and what to do with the results afterwards (concept 2), which were unanswered in our MiningZinc extension, are still unanswered in the MiniZinc language.

Meanwhile my research interests have shifted from using Constraint Programming (CP) for data mining/machine learning (ML) to the opposite direction: using ML to learn part of the CP problem specification, such as learning coefficients of an objective function, learning preferences of operators and generating (optimal) explanations of UNSAT and SAT problems.

For these use cases, integration with a) machine learning libraries, 2) multiple solver technologies 3) data loading libraries, and 4) visualisation libraries prompted us to develop a new Python-based library for CP.

Constraint solving systems from the viewpoint of inductive databases Our needs for this system fit exactly the concepts that were brought forward for inductive database systems: 1) support data manipulation (data queries) as well as computing the solutions to combinatorial problems (solve queries); 2) that the output of solve queries should be basic objects that can then be used in further queries (the closure principle); and 3) that in such nested cases, conditions from one type of operation flow into the processing of the next.

Notably, the CPMpy system [3][1] we are developing as part of my ERC Consolidator Grant is a *library* that builds symbolic expression trees that can be reasoned on, rather than a standalone language like SQL. We remark that in the machine learning field in general, the use of programmatic manipulation of data, numpy arrays more specifically, is much more prevalent then query languages. However, the functions offered by numpy and other scientific libraries like pandas are high level and often close the algebraic manipulations underlying SQL systems.

So a first import design decision in CPMpy is to adapt the numpy array as *basic object* that will enable the closure principle, meaning: numpy array's in and numpy array's out (note that they can be matrices, but also vectors or higher-order tensors; in contrast to tables all elements are typically of the same type though).

The key part of our modeling library is hence to have *decision variables* be numpy arrays, thereby allow all operations that you do on such arrays, to do them with the same syntax and meaning on decision variables. If the operations are done on data, the output is data; if the operations are done on decision variables, the output are expression trees of relations between variables, for which a solver can compute satisfying or optimal assignments. After a solver is called on them, the decision variables can be asked for their value with the `var.value()` function, which returns a numpy array with the values as data.

In this framework, the circle is hence round, and the three design goals are achieved.

Here are some examples of what this enables, in a single framework with a single syntax:

1. **Diverse solutions**[4] We read data from an excel file, use that to construct a packing problem, find the optimal solution; then feed that solution back into the packing problem ensuring that the next solution is sufficiently different, repeating this k times. (Equally applicable to diverse pattern mining with a constraint programming formulation of itemset mining for example.))
   Data is from an external source, used to create constraints, and solutions are used as data to create new constraints in turn.

[1] https://github.com/CPMpy/cpmpy

2. **Visual sudoku**[5] We pass an image through a pretrained neural network which retuns 81*10 probabilistic outputs: for each cell of the sudoku whether it contains number 1-9 or is empty; these (log) probabilities are used to create an optimisation problem that searches for the maximum likelihood assignment that satisfies the sudoku constraints. This solution is then projected back on the non-empty cells, for which another solver is called to verify whether the projected solution is unique or not, and if not it is added as a cutting plane to the original solver.
3. **Decision-focused learning**[6] The goal is to predict, for example, hourly energy prices, in a way that an energy-aware scheduling problem over the coming 24 hours minimizes expected energy costs. In decision-focused learning we feed a training example in the neural network to obtain predicted prices, perturb the prices in a statistically motived way, feed that into a constraint solver, and compute a subgradient based on the obtained solution, that is backpropagate through the neural network.
   (we even go as far as building a small database of solutions and query a solution from that as approximation for unseen predicition vectors)

There is no other constraint programming system that supports these operations in a single system with a single syntax, that is, in which data queries and solve queries are equal first-class citizens.

Who knows, perhaps the reason is that other constraint programming researchers had not read the seminal paper of Imielinksi and Mannila.

Acknowledgments I would like to wholeheartedly thank Siegfried Nijssen, Luc De Raedt, Anton Dries and all partners of the EU ICON project [7] for the many interesting exchanges we had that shaped these ideas.

References

[1] T. Imielinski, H. Mannila, A database perspective on knowledge discovery, Commun. ACM 39 (1996) 58–64.

[2] T. Guns, S. Nijssen, L. De Raedt, Itemset mining: A constraint programming perspective, Artificial Intelligence 175 (2011) 1951–1983.

[3] T. Guns, Increasing modeling language convenience with a universal n-dimensional array, cppy as python-embedded example., 2019.

[4] E. Hebrard, B. Hnich, B. O'Sullivan, T. Walsh, Finding diverse and similar solutions in constraint programming, in: AAAI, volume 5, 2005, pp. 372–377.

[5] M. Mulamba, J. Mandi, R. Canoy, T. Guns, Hybrid classification and reasoning for image-based constraint solving, in: International Conference on Integration of Constraint Programming, Artificial Intelligence, and Operations Research, Springer, 2020, pp. 364–380.

[6] M. Mulamba, J. Mandi, M. Diligenti, M. Lombardi, V. Bucarey, T. Guns, Contrastive losses and solution caching for predict-and-optimize, in: Z. Zhou (Ed.), Proceedings of the Thirtieth International Joint Conference on Artificial Intelligence, IJCAI 2021, Virtual Event / Montreal, Canada, 19-27 August 2021, ijcai.org, 2021, pp. 2833–2840.

[7] C. Bessiere, L. De Raedt, T. Guns, L. Kotthoff, M. Nanni, S. Nijssen, B. O'Sullivan, A. Paparrizou, D. Pedreschi, H. Simonis, The inductive constraint programming loop, IEEE Intelligent Systems 32 (2017) 44–52.