Relational Decomposition using Answer Set Programming

Luc De Raedt, Sergey Paramonov, and Matthijs van Leeuwen

KU Leuven, Department of Computer Science, Leuven, Belgium
{luc.deraedt,sergey.paramonov,matthijs.vanleeuwen}@cs.kuleuven.be

Abstract. Motivated by an analogy with matrix decomposition, we introduce the novel problem of relational decomposition. In matrix decomposition one is given a matrix and has to decompose it as a product of other matrices. In relational decomposition, one is given a relation r and one has to decompose it as a conjunctive query of a particular form $q :- q_1 \wedge ... \wedge q_n$. Furthermore, the decomposition has to satisfy certain constraints (e.g. that $r \approx q$ holds). Relational decomposition is thus the *inverse* problem of querying as one is given the result of the query and has to compute the relations constituting the query itself.

We show that relational decomposition generalizes several well-studied problems in data mining such as tiling, boolean matrix factorization, and discriminative pattern set mining. Furthermore, we provide an initial strategy for solving relational decomposition problems that is based on answer set programming. The resulting problem formalizations and corresponding solvers fit within the declarative modelling paradigm for data mining.

1 Introduction

Decomposing matrices is one of the most popular techniques in machine learning and data mining and many variants have been studied, e.g. non-negative, singular value and boolean matrix decomposition. The latter problem is illustrated in Figure 1. Given a boolean $n \times m$ matrix $\mathbf{A}$, the problem is to write it as the product of a $n \times k$ matrix $\mathbf{B}$ and $k \times m$ matrix $\mathbf{C}$, such that $\mathbf{A}_{i,j} = \sum_k \mathbf{B}_{i,k} \cdot \mathbf{C}_{k,j}$ in the boolean algebra (in which $1 + 1 = 1$). The columns of $\mathbf{B}$ and rows of $\mathbf{C}$ can be interpreted as patterns; cf. [1]. Depending on k, exact decompositions may not be possible, and one then resorts to an approximation, that is, one searches for a $\mathbf{B} \cdot \mathbf{C}$ that is close to $\mathbf{A}$. Usually $k << n, m$ so that the original matrix is compressed. Various uses exist for the resulting patterns.

This paper investigates this type of decomposition[1] using a relational algebra rather than a matrix algebra. It basically replaces the matrices by relations and the products by a conjunction or join; this is illustrated in Figure 2. The problem of relational decomposition can be formalized as follows.

Definition 1. *Let r be a relation with its extension, and let $q :- q_1, ..., q_n$ be a query*[2] *such that q and r share the same variables. Then, the problem is to find the extensions of the relations q_i such that $r \approx q$.*

[1] We synonymously use the word "factorization"

[2] In relational algebra, the query q is the projection on the attributes of r of the join of the q_i.

Fig. 1: Classic boolean matrix factorization

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

Fig. 2: Relational decomposition of a relation about cars

A

State	**Age**	**Fuel**	**Type**
Fair	Old	Gas	Sport
Good	New	Gas	Sport
Fair	Old	Electric	HEV
Good	New	Electric	HEV

=

B

C	**D**
code$_1$	*code*$_3$
code$_2$	*code*$_3$
code$_1$	*code*$_4$
code$_2$	*code*$_4$

⋈

C

C	**State**	**Age**
code$_1$	Fair	Old
code$_2$	Good	New

⋈

D

D	**Fuel**	**Type**
code$_3$	Gas	Sport
code$_4$	Electric	HEV

In many cases, further constraints on the problem will be given as well as a way to measure how close r and q are. The problem is thus a constrained optimization problem. Observe that the example in Figure 2 cannot be cast as a traditional matrix decomposition problem due to the symbolic nature of the values in the table. So, relational decomposition goes beyond matrix decomposition.

In this paper we mainly focus on the data mining tasks that we generalize, but there is also important related work in, e.g., inductive logic programming and statistical relational learning. For example, 'block models' are a relatively old application of matrix factorization for relations. 'Stochastic block models' build upon this by attaching a probabilistic model to the factorization [2]. Well-known recent examples are [3] and [4]. [5] recently described the use of Boolean Matrix Factorization for multi-relational learning and multi-label classification.

In the remainder of this paper, we will demonstrate that 1) relational decomposition provides a general framework that allows for abstraction of several well-studied problems in the data mining literature, such as boolean matrix factorization, tiling, and discriminative pattern set mining, and 2) a simple solver for some relational decomposition tasks using answer set programming (ASP) can be developed.

2 An example: tiling

Data mining has contributed numerous techniques for finding patterns in (boolean) matrices. One fundamental approach is that of *tiling* [6]. A tile is basically a rectangular area in a boolean matrix for which all values are 1, specified by a subset of rows and a subset of columns (or transactions and items). One is typically not interested in *any* tile, but in maximal tiles, i.e. tiles that cannot be extended. For instance, in matrix $\mathbf{A}$ in Figure 1, the tile defined by rows $\{1,2\}$ and columns $\{1,2\}$ is a maximal tile. Tiles characterize high density regions of interest and rather than searching for a single tile, one typically searches for a (small) set of tiles that together cover as much of the 1's in the matrix as possible. A second tile would be ($\{1,3\}$,$\{1,3\}$), and together the two tiles cover all 1's in the matrix.

Listing 1.1: Greedy tile formalization in answer set programming

```
%guess an extension of the code / at most one value per Attribute
0 { code(guess, Value, Attribute) : valid(Attribute, Value) } 1 :− col(Attribute).
%definition of over-coverage to encode the first constraint
over_covered(guess,T) :− not db(Value, Attribute, T), code(guess, Value, Attribute), transaction(T).
%check if the code intersects with the other codes to encode the second constraint
intersect(T) :− guess != Index, code(guess, Value, Attribute), code(Index, Value, Attribute), in(Index,T).
%check if a code can be applied
in(guess,Transct) :− transaction(Transct), not over_covered(guess, Transct), not intersect(Transct).
covered(Transct, Attribute) :− in(Index,Transct), code(Index, Value, Attribute).
#maximize[covered(Transct, Attribute)].
```

Let us now formalize tiling as a relational decomposition problem; we will then solve it using answer set programming. In doing so, we consider the full relational case, rather than restricting ourselves to boolean values as is traditionally done.

Given a relation $\texttt{db}(\mathit{Value}, \mathit{Attribute}, \mathit{Transct})$ (denoting that *Transct* has *Value* for *Attribute*), the task is to find a set of codes (combinations of attribute-values) that can be applied to the transactions to "summarize" the dataset $\texttt{db}$. Simply speaking, a coding set is a sequence of attribute-value pairs

$$c_1 = \{\texttt{code}(c1, a, 1), \texttt{code}(c1, b, 2), \texttt{code}(c1, c, 3)\},$$

where the first argument of each $\texttt{code}$ is the index of the code, the second is the value of this attribute, and the third argument is the index of an attribute. When code C is applied to a transaction T (i.e. it occurs in the transaction), this is denoted by $\texttt{in}(C, T)$.

Tiling now corresponds to finding an approximation $\texttt{adb}(\mathit{Value}, \mathit{Attribute}, \mathit{Transct})$ for $\texttt{db}(\mathit{Value}, \mathit{Attribute}, \mathit{Transct})$ by establishing a set of facts over $\texttt{code}$ and $\texttt{in}$ under the following constraints and query:

$$\begin{aligned} &\texttt{adb}(\mathit{Value}, \mathit{Attribute}, \mathit{Transct}) \text{ :- } \texttt{code}(\mathit{Index}, \mathit{Value}, \mathit{Attribute}), \texttt{in}(\mathit{Index}, \mathit{Transct}) \\ &\text{with the following clausal constraints} \\ &\texttt{code}(\mathit{Index}, \mathit{Value}, \mathit{Attribute}), \texttt{in}(\mathit{Index}, \mathit{Transct}) \rightarrow \texttt{db}(\mathit{Value}, \mathit{Attribute}, \mathit{Transct}) \\ &\texttt{in}(I_1, T), \texttt{in}(I_2, T), \texttt{code}(I_1, L, C_1), \texttt{code}(I_2, L, C_2) \rightarrow C_1 \neq C_2 \end{aligned} \tag{1}$$

The first constraint states that codes cover part of the database, the second one that two codes cannot occur in the same transaction if they contain the same attribute (i.e. tiles are not allowed to overlap). To find a maximal tiling, we need a notion of coverage.

$$\texttt{covered}(\mathit{Transct}, \mathit{Attribute}) \text{ :- } \texttt{in}(\mathit{Index}, \mathit{Transct}), \texttt{code}(\mathit{Index}, \mathit{Value}, \mathit{Attribute}).$$

A tiling is *maximal* iff it maximizes the number of covered attribute-transaction pairs.

The problem sketched above can be encoded in answer set programming as indicated in Listing 1.1. The code mimics a greedy algorithm for the maximal tiling problem with a fixed number of tiles k. It assumes we have already found an optimal tiling for $n - 1$ tiles, and indicates how to find the n-th tile to cover the largest area. The n-th tile is called *guess* in the listing. Furthermore, we have information about the names of the attributes and the possible values for a particular attribute (through the predicates $\texttt{attr}(\mathit{Attribute})$ and $\texttt{valid}(\mathit{Attribute}, \mathit{Value})$).

3 Generality of the framework

As mentioned, relational decomposition generalizes numerous data mining tasks. Key advantages of answer set programming are 1) the flexibility and 2) compactness of the problem formalizations. Indeed, using ASP it is very easy to specify several tasks, many of which have been studied in the literature; this is very much in the spirit of the declarative constraint programming paradigm [7]. We illustrate this with a few examples.

Overlapping Tiling Tiles in a tiling are usually not allowed to overlap; looking for overlapping tilings is generally a very hard problem. However, changing the assumption in our ASP implementation is straightforward. It involves replacing constraint 1 by e.g.

$$\#\{\texttt{in}(I_1,T),\texttt{in}(I_2,T),\texttt{code}(I_1,V,A_1),\texttt{code}(I_2,V,A_2),A_1=A_2\}\leq N, \quad (2)$$

which corresponds to assuming that in one transaction two codes can intersect only on N attributes. This can be encoded in ASP as indicated in Listing 1.2 (see Appendix).

Boolean Matrix Factorization It is well-known that tiling and *boolean matrix factorization* are closely related [8]. So, let us briefly show how BMF can be realized in our framework. It corresponds to the variant of the tiling problems where only binary values (true and false) are possible and a fixed number of codes k. It is straightforward to realize this by deleting all arguments in Listing 1.1 corresponding to Values and retaining only the transactions in db(*Value*, *Attribute*, *Transct*) with value true, that is, encoding only the true facts for db*(T,I)*. (See Listing 1.3 in the Appendix.)

Discriminative Pattern Set Mining A common supervised data mining task is that of discriminative pattern set mining [9]. Let db(*Value*, *Attribute*, *Transct*) be a categorical dataset, *positive(T)* (*negative(T)*) be the positive (negative) transactions, k be the number of codes. Then, the task is to find extensions of the relations code(*Index*,*Value*,*Attribute*) and in(*Index*,*Transct*) such that positive and negative transactions are discriminated. We can achieve this by maximizing the difference between the number of covered positive and negative transactions. (See Listing 1.4 in the Appendix.)

MDL-based pattern sets KRIMP [10] is a variation on tiling that derives a so-called code table to encode the data. It is based on the MDL principle and aims to identify a set of tiles (itemsets) that together compress the data well. It can be realized by replacing the optimization objective: the total compressed size of the data given the tiles should be minimized. (We do not present the formulation because of lack of space.)

4 Experiments

We evaluate the ASP problem formulations on a 64-bit Ubuntu machine with Intel Core i5-3570 CPU @ 3.40GHz x 4 and 8GB memory (except when indicated). The ASP engine is 64-bit clingo version 3.0.5. The experiments have been carried out on the following datasets: Congressional Voting Records, Solar Flare, Tic-Tac-Toe Endgame, Nursery, Mushrooms, Chess (King-Rook vs. King-Pawn) from [11] and the Animals with Attributes dataset from [12]. We present results obtained after performing a meta-experiment to determine the best parameters for the ASP solver.

Fig. 3: Greedy tiling – time

	Number of tiles				
Dataset	**5**	**10**	**15**	**20**	**25**
Animals	36s	64s	81s	92s	96s
Solar flare	6s	10s	13s	16s	18s
Tic-tac-toe	22s	31s	33s	34s	35s
Nursery	4m19s	6m32s	7m32s	7m56s	8m13s
Voting	52s	88s	102s	106s	109s
Chess	17h	22h	-	-	-
Mushrooms	13h	19h	-	-	-

Fig. 4: Greedy tiling – coverage

	Number of tiles					
Dataset	**1**	**5**	**10**	**15**	**20**	**25**
Animals	0.118	0.327	0.472	0.573	0.649	0.709
Solar flare	0.230	0.416	0.565	0.655	0.721	0.751
Tic-tac-toe	0.076	0.251	0.449	0.623	0.784	0.907
Nursery	0.074	0.269	0.454	0.634	0.773	0.905
Voting	0.134	0.399	0.553	0.662	0.749	0.810
Chess	0.254	0.483	0.618	-	-	-
Mushroom	0.168	0.476	0.586	-	-	-

Fig. 5: Overlapping tiling – time (s)

		Number of tiles				
Dataset	**Overlap**	**5**	**10**	**15**	**20**	**25**
Animals	1	70	158	226	264	287
	2	99	250	386	460	496
Solar flare	1	8	13	17	21	24
	2	8	15	20	25	29
Tic-tac-toe	1	24	41	49	52	53
	2	23	43	51	55	56
Nursery	1	300	519	610	648	672
	2	343	572	669	710	734
Voting	1	70	139	173	188	195
	2	99	214	275	309	333

Fig. 6: Overlapping tiling – coverage

		Number of tiles					
Dataset	**Overlap**	**1**	**5**	**10**	**15**	**20**	**25**
Animals	1	0.117	0.327	0.475	0.583	0.663	0.722
	2	0.117	0.332	0.482	0.592	0.675	0.742
Solar flare	1	0.230	0.433	0.595	0.684	0.734	0.756
	2	0.230	0.452	0.602	0.685	0.731	0.755
Tic-tac-toe	1	0.076	0.253	0.451	0.626	0.781	0.898
	2	0.076	0.253	0.451	0.626	0.781	0.898
Nursery	1	0.074	0.268	0.454	0.633	0.772	0.905
	2	0.074	0.268	0.454	0.633	0.772	0.905
Voting	1	0.134	0.403	0.558	0.675	0.765	0.828
	2	0.134	0.409	0.571	0.683	0.762	0.819

Greedy tiling Figures 3 and 4 present timing and coverage results obtained on all datasets. Due to high memory requirements of the ASP system experiments on the Chess and Mushroom have been performed on a 64-bit Ubuntu machine with 24 Intel Xeon CPU and 128GB of memory (but all experiments were run single-threaded).

In all cases the problem formalisation given in Listing 1.1 was used to mine 25 tiles. Since the problem becomes over-constrained as the number of tiles increases, running time and coverage changes for each new tile. We therefore report total running times and coverage for different total numbers of tiles. For Chess and Mushrooms only results on the first 10 tiles were computed due to long runtimes.

Overlapping tiling We apply the problem formalisation in Listing 1.2 (see Appendix) to five datasets, with two levels of overlap: tiles can intersect on at most one or two attribute(s). As the results in Figures 5 and 6 show, this can give a small increase in coverage, but runtimes increase due to the costly aggregate operation in Line 1.

Boolean Matrix Factorization We applied the formalisation of Listing 1.3 to the Animals dataset and measured coverage gain and required time per iteration, where the

Fig. 7: BMF on the Animals dataset

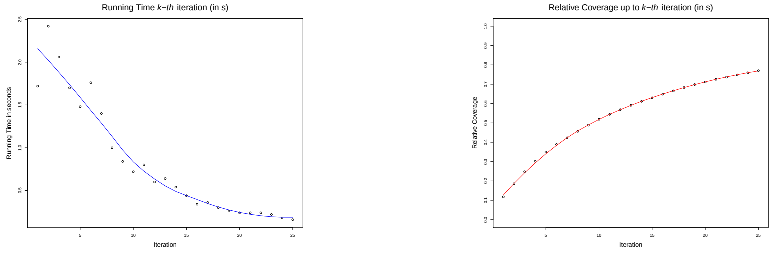
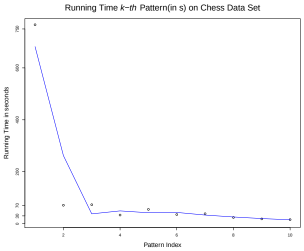


Fig. 8: Discriminative pattern set mining



	Chess (k = 10)	Tic-tac-toe (k = 5)
Covered negative transactions	160	92
Covered positive transactions	864	626
Difference positive/negative	704	534
Running time	18m48s	0.52s

decomposition rank k was incremented by one in each iteration. The results, summarized in Figure 7, show coverages similar to those obtained in [8]. However, running times are several times higher, which can be explained by the usage of a general solver.

Discriminative Pattern Set Mining Experiment Here we demonstrate the feasibility of our approach to discriminative k-pattern mining. Among the datasets we consider, there are two that can be directly used for discriminative pattern mining: Chess and Tic-tac-toe, each of which has two classes "won" and "no-win". We apply Listing 1.4 to both datasets and summarize the results in Figure 8. The experiments show that five patterns are enough to cover all positive examples in case of Tic-tac-toe; mining more than 5 patterns is useless. It is interesting to observe the running time for the Chess dataset. It seems that the problem gets significantly easier, from a computational point of view, once the initial tile is chosen, which confirms our intuition that the search space shrinks when the problem becomes more constrained (the number of answer sets and hence the "search space" becomes smaller with the addition of more constraints).

5 Conclusions and future work

We introduced the problem of relational decomposition, a form of inverse querying. From an inductive logic programming perspective, it could be related to predicate invention and a form of constrained abduction. From a data mining perspective, it provides a general, declarative framework for specifying a multitude of different pattern set mining problems. In the future we intend to further explore what tasks can be expressed within this framework, and to improve the expressibility and efficiency of the solver.

We have shown that – despite its simplicity – the preliminary ASP implementation can already solve reasonable decomposition problems, but complete decomposition of larger datasets like Chess and Mushrooms is currently out of reach. Nevertheless, we believe that the experiments provide evidence for the potential of the proposed approach.

References

1. Miettinen, P., Mielikäinen, T., Gionis, A., Das, G., Mannila, H.: The discrete basis problem. IEEE Trans. Knowl. Data Eng. **20**(10) (2008) 1348–1362
2. Holland, P.W., Laskey, K.B., Leinhardt, S.: Stochastic blockmodels: First steps. Social networks **5**(2) (1983) 109–137

3. Kemp, C., Tenenbaum, J.B., Griffiths, T.L., Yamada, T., Ueda, N.: Learning systems of concepts with an infinite relational model. In: Proc.nat.conf. on artificial intelligence. (2006)
4. Airoldi, E.M., Blei, D.M., Fienberg, S.E., Xing, E.P.: Mixed membership stochastic blockmodels. Journal of Machine Learning Research **9** (2008) 1981–2014
5. Wicker, J., Lehnerer, S., Kramer, S.: Multi-relational learning by boolean matrix factorization and multi-label classification
6. Geerts, F., Goethals, B., Mielikäinen, T.: Tiling databases. In: Discovery Science, Springer (2004) 278–289
7. Guns, T., Nijssen, S., Raedt, L.D.: Itemset mining: A constraint programming perspective (2011)
8. Miettinen, P.: Dynamic boolean matrix factorizations. In Zaki, M.J., Siebes, A., Yu, J.X., Goethals, B., Webb, G.I., Wu, X., eds.: ICDM, IEEE Computer Society (2012) 519–528
9. Knobbe, A.J., Ho, E.K.Y.: Pattern teams. In Fürnkranz, J., Scheffer, T., Spiliopoulou, M., eds.: PKDD. Volume 4213 of Lecture Notes in Computer Science., Springer (2006) 577–584
10. Siebes, A., Vreeken, J., van Leeuwen, M.: Item sets that compress. In: SDM. (2006)
11. Bache, K., Lichman, M.: Uci machine learning repository (2013)
12. Osherson, D., Stern, J., Wilkie, O., Stob, M., Smith, E.: Default probability. Cognitive Science **15**(2) (1991) 251–269

A Appendix

Listing 1.2: Overlapping tiling

```
intersect_N(T,Attr) :− guess != Indx, code(guess, V, Attr), code(Indx, V, Attr), in(Indx,T).
intersect(T) :− overlap_level #count{ intersect_N(T,Indx) : col(Indx) }, transaction(T).
```

Listing 1.3: Boolean Matrix Factorization

```
0 { code(guess,I) } 1 :− item(I).
over_cover(guess, T) :− not db(T,I), code(guess,I), transaction(T).
in(guess,T) :− not over_cover(guess,T), transaction(T).
covered(T,I) :− code(guess,I), in(guess,T), db(T,I).
#maximize[covered(T,I)].
```

Listing 1.4: Discriminative k-pattern set mining

```
in(guess,T) :− transaction(T), not over_covered(guess, T).
covered_plus(T) :− in(Indx,T), code(Indx, Value, Attribute), positive(T).
covered_minus(T) :− in(Indx,T), code(Indx, Value, Attribute), negative(T).
#maximize[covered_plus(T) = 1, covered_minus(T) = −1].
```