Machine Learning, 57, 305–333, 2004

Compact Representation of Knowledge Bases in Inductive Logic Programming

JAN STRUYF, JAN RAMON, MAURICE BRUYNOOGHE, SOFIE VERBAETEN
AND HENDRIK BLOCKEEL
Department of Computer Science, Katholieke Universiteit Leuven, Celestijnenlaan 200A, B-3001 Leuven, Belgium

Editor: Stan Matwin

Abstract. In many applications of Inductive Logic Programming (ILP), learning occurs from a knowledge base that contains a large number of examples. Storing such a knowledge base may consume a lot of memory. Often, there is a substantial overlap of information between different examples. To reduce memory consumption, we propose a method to represent a knowledge base more compactly. We achieve this by introducing a meta-theory able to build new theories out of other (smaller) theories. In this way, the information associated with an example can be built from the information associated with one or more other examples and redundant storage of shared information is avoided. We also discuss algorithms to construct the information associated with example theories and report on a number of experiments evaluating our method in different problem domains.

Keywords: Inductive Logic Programming, efficiency, scalability, knowledge bases, compact representation

1. Introduction

Machine learning in general is concerned with the induction of new knowledge (hypotheses) from a given set of examples, stored in a knowledge base. Knowledge can be stored and arranged in different ways, and the most obvious way is not always the most space-efficient one. There may be redundancy because some information is repeated across several examples (this may happen systematically, e.g., because of functional dependencies, or occasionally), or because certain information can easily be derived from other information.

Having a compact representation (i.e., one with less redundancy) is important for a number of reasons. The most obvious one is that storing a compact knowledge base requires less space, both on disk and in main memory. On the other hand, a more compact representation may render the processing of the data less time-efficient; this is a risk that needs to be avoided.

In this article we look at the problem of compact representations from a machine learning perspective, and more specifically that of inductive logic programming. The main contributions of this work are the introduction of a formalism that allows for a more compact representation without a significant computational penalty, and an algorithm that processes data thus represented in the most efficient way. In certain specific cases, the new formalism boils down to well-known techniques, but in general it is more widely applicable.

In Section 2 we provide some context and motivation for our work. The framework that we will introduce consists of two layers. In Section 3 we describe the first layer: the

knowledge representation layer. Examples will be defined compactly by meta-theories. In Sections 4 and 5 we describe the second layer, which consists of the algorithms that support the efficient querying of examples specified by a meta-theory. In Section 6 we present some experiments evaluating our method in different problem domains and in Section 7 we state the conclusions.

2. Context and motivation

In Inductive Logic Programming (ILP), an example is described by a number of relations, each relation formalizing a relevant property of the example. We here consider ILP systems that learn from interpretations (De Raedt and Džeroski, 1994), where an example is represented by a logic program (or theory) and its meaning is given by the interpretation that corresponds to the program's least Herbrand model. The logic program can be a trivial one, consisting of a set of ground facts, like base tables in a relational database, or can, in addition, also contain relations defined in terms of other ones, as in deductive databases, or views in relational databases.

Figure 1 compares different methods for storing examples in a knowledge base. Many ILP systems represent all examples by one monolithic logic program P (figure 1(a)). Typically, the size of P is linear in the number of examples. Indeed, each example adds a number of clauses to P. Some researchers have explored the use of a relational database management system (RDBMS) to store P (Blockeel and De Raedt, 1996; Morik and Brockhausen, 1997; Ito and Ohwada, 2001) and also deductive database systems (Das, 1992; Arni et al., 2003) could be used. However, storing P in a database causes a substantial slow-down in comparison with storing P as a compiled logic program in main memory, as is done by most ILP systems.

Storing P in main memory also has its disadvantages. When the number of examples becomes large, the main memory may be too small to store the whole of P. But even before the memory limits are exceeded, querying a single example can become expensive because the relevant clauses must be accessed through indexes.

A method that solves the problem posed by large numbers of examples is shown in figure 1(b). Each example is represented by a small logic program e_i (which we call the theory about the example). Querying a single example is more efficient because the size of e_i does not depend on the total number of examples. If the entire knowledge base does

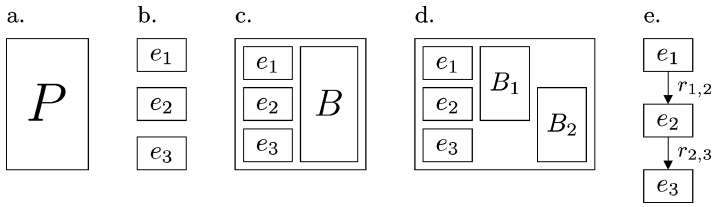

Figure 1. Different ways to represent a knowledge base in ILP.

not fit in main memory, then examples can be loaded, queried and removed one by one. This method is used by systems that implement the learning from interpretations setting (Blockeel et al., 1999), which is the focus of the work presented here. A system such as ilProlog (Blockeel et al., 2002) has special features for the efficient loading of examples, a functionality not available in relational and deductive database systems.

In many applications there exists information that is relevant to all examples (e.g., domain knowledge). Storing this information in each e_i introduces a lot of duplicated information. Many ILP systems therefore provide the possibility to add a background theory B to the set of examples (figure 1(c)). Every example is then the least Herbrand model of the program that is the union of an example specific logic program e_i and a fixed logic program B. In this way, the total size of the knowledge base is reduced, as duplicating the background knowledge in each example is avoided.[1]

However, the combination of example specific knowledge and background knowledge is often not completely satisfying. Consider a knowledge base where a large amount of information is relevant to several (but not all) examples. Repeating this information in all examples that need it would introduce a lot of duplication. On the other hand, if we store all information relevant to more than one example in B then we lose the "locality" of this information. The size of B increases and the process of querying examples slows down. Also, the background theory could become too big to fit in main memory. A possible solution is shown in (figure 1(d)) where the background theory is split in different parts. B_1 is contained in examples e_1 and e_2, but shared by them, rather than duplicated in each of them; similarly, e_2 and e_3 share B_2. Clearly, for this solution to be feasible, we need a way to structure the background knowledge into parts, and indicate which parts are relevant to which examples.

Finally, figure 1(e) sketches a situation where the most concise way to obtain the logic program describing an example e_j is by performing some actions, specified by a set of rules $r_{i,j}$ on the logic program of another example e_i. This is similar to the previous situation, but now the "relevant knowledge" includes not only background knowledge, but also another example.

In this work, we define a language that supports the structuring of a knowledge base into chunks of knowledge ("theories"), and the definition of such theories from other theories. Each example then corresponds to a theory that contains only knowledge relevant to that example, and different examples may share theories. Thus, the modularity and querying efficiency of figure 1(b) is combined with the representational efficiency of figure 1(c)–(e).

This may come at the cost of a more expensive example construction. E.g., in figure 1(e), example e_3 can only be constructed by first constructing e_1 and then applying rules $r_{1,2}$ and $r_{2,3}$. Therefore, in addition, we develop an example iterator that, given a set of examples, iterates over this set in some kind of optimal order. Generally, this optimal order minimizes the example construction cost. For instance, if examples e_1 and e_2 share many theories, say $e_1 = \{a, b, c\}$ and $e_2 = \{a, b, d\}$, then it is better to process e_2 immediately after e_1, when some of the theories relevant for it are already in main memory and we just have to remove theory c and add d, rather than let other examples come between them.

In the rest of this section we describe some concrete examples. Several of these will be treated in more detail later in the text.

Example 1. Consider a knowledge base that stores molecules, which contain several functional groups. Storing each molecule independently will introduce duplication because most functional groups will reoccur in different molecules; we would rather have the different molecules "share" such groups.

Example 2. Many knowledge bases store information in multiple dimensions. For example, one dimension can store structural information about drugs, another dimension can store clinical information about patients, and an example describes a specific patient treated with a particular drug. Representing examples independently will introduce a lot of duplication as both the same drug and the same patient can occur in several drug-patient combinations. Alternatively, storing all drug and patient information in the background knowledge has the drawback of introducing a large background theory and slowing down access to this information.

Example 3. Many learning tasks are concerned with the classification of an individual element that is part of a larger sequence and use the local context of the element. Examples are protein secondary structure prediction (Muggleton, King, and Sternberg, 1992), part of speech tagging (Cussens, 1997) and user modeling (Jacobs and Blockeel, 2001). Storing each example independently introduces duplication because the local context of consecutive examples overlaps. Again, the overlap is avoided by including all the sequence information in the background, which however destroys the locality of the information and is therefore detrimental to computational efficiency.

Example 4. The task in reinforcement learning (Džeroski et al., 2001) is to learn a relationship between the structural description of a state and the optimal action for that state. A typical knowledge base contains a number of episodes: sequences of states in which each state can be reached from the previous one by taking a certain action. Storing each state independently introduces redundancy. A more efficient alternative is to store only the initial state together with the sequence of actions. A similar situation is encountered when learning to play games like chess or Go, where the knowledge base consists of a number of played games (see Ramon, Francis, and Blockeel, 2000, for an example).

The above examples are quite different in nature, and for each one a different solution can be devised; but we are interested in finding a general framework that can handle all of them.

The framework that we introduce consists of two layers: a knowledge representation layer and an algorithmic layer. In the following section we describe the knowledge representation layer, which takes care of the structuring of the knowledge base into chunks that we call theories. Examples are defined in terms of these theories, and can be constructed from them on demand. In Sections 4 and 5 the algorithmic layer will be described, which contains algorithms able to iterate over a set of examples, processing each one of them consecutively, with minimal example construction cost.

3. The knowledge representation layer

In this section, we describe the knowledge representation layer of our framework.

First, the basic concepts of theory and meta-theory are defined and motivated with examples. Then, in Section 3.2, we present the precise meaning of these concepts by means of a translation to meta-programs. Some more elaborate examples are given in Section 3.3. Finally, in Section 3.4, the notion of schema is introduced that allows the user to concisely represent a set of similar (meta-)theories. All this is summarized again in Section 3.5.

3.1. Theories and meta-theories

First, we motivate and introduce the notion of theory as a basic unit of knowledge. Next, we develop meta-theories that allow the user to combine pieces of knowledge into larger ones.

We use the following standard terminology. A *term* is either a variable or a constant or of the form `f(t1,...,tn)` with `f` a functor symbol and `ti` ($n \geq 1$) terms. An *atom* is of the form `p` or `p(t1,...,tn)` with `p` a predicate symbol and `ti` ($n \geq 0$) terms. A *clause* is of the form `A :- B1,...,Bn.` with `A` and `Bi` atoms. A clause is called a *fact* and written as `A.` when $n = 0$.

A *substitution* θ is a finite set of the form $\{$`X1/t1,...,Xn/tn`$\}$, where the `Xi` are distinct variables and each `ti` is a term distinct from `Xi`. The application of a substitution θ to an expression `E` is written as `E`θ.

A substitution θ is called a *unifier* of two expressions `E1` and `E2` iff `E1`θ = `E2`θ. A unifier θ of `E1` and `E2` is called a *most general unifier*, or *mgu* for short, iff for each unifier σ of `E1` and `E2` and for each expression `E'`, there exists a substitution γ such that `E'`σ = `E'`$\theta\gamma$.

The purpose of our framework is to allow the user to group knowledge, expressed as clauses, into units and to provide a means to combine units of knowledge into larger ones. The basic unit is called *theory* and consists of a sequence of clauses.

Example 5. A theory representing the molecule H_2O can be defined as a sequence of facts:

```
begin_theory h2o
  atom(h1,h).       bond(h1,o1,1).
  atom(h2,h).       bond(h2,o1,1).
  atom(o1,o).
end_theory
```

The facts are grouped in a unit using the `begin_theory` and `end_theory` keywords; the knowledge unit is given the name `h2o`.

Example 6. The following is a simple background theory `bg` that uses a pair of clauses to define some properties of molecules.

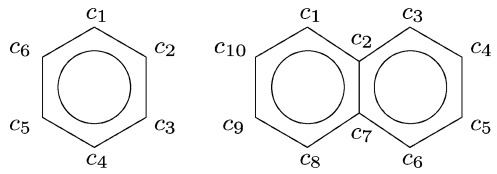

Figure 2. A benzene molecule and a naphthalene molecule.

```
begin_theory bg
  contains_double_bond :- bond(_,_,2).
  atom_count(N) :- findall(A,atom(A,_),L), length(L,N).
end_theory
```

Now, consider the molecules drawn in figure 2. On the left is a benzene molecule. One could define it by a number of `atom/2` and `bond/3` facts as we did for the H_2O molecule. However, this way, it cannot be used as a building block in defining the naphthalene molecule on the right that consists of two benzene rings. To facilitate the latter, the names of the *c*-atoms should be parameters that can be instantiated when using the benzene theory as a building block in a larger theory. This is done in the following example.

Example 7. In this theory, the names of the atoms are parameters (variables) that are included in the theory name.

```
begin_theory benzene_ring(C1,C2,C3,C4,C5,C6)
  atom(C1, c).       bond(C1, C2, aromatic).
  atom(C2, c).       bond(C2, C3, aromatic).
  atom(C3, c).       bond(C3, C4, aromatic).
  atom(C4, c).       bond(C4, C5, aromatic).
  atom(C5, c).       bond(C5, C6, aromatic).
  atom(C6, c).       bond(C6, C1, aromatic).
end_theory
```

It is necessary to make a distinction between theories that are complete examples (hence are intended to be queried by the ILP system) and other theories that are building blocks in constructing examples. We do so by using a distinct set of keywords to delineate the former. We use the pair `begin_example` and `end_example` to identify a theory as an example.

The above motivates the following definition of theory:

Definition 1 (Theory). A *theory* consists of a name and a sequence of clauses. A name is a term. The (possible) variables in this term are called the parameters of the theory. A theory is delineated by either the keywords `begin_example` and `end_example` or the keywords `begin_theory` and `end_theory`.

In the remainder of this text, when we refer to theories, this normally includes the special case of examples.

As will be detailed in Section 3.2, a theory defines a clause for one of the meta-predicates `example/2` and `theory/2` (depending on whether the `example` or `theory` keywords are used), which take as arguments the name of the theory and the list of clauses representing the theory. For example, calling `theory(benzene_ring(c1,c2,c3,c4,c5,c6),T)` will first unify the formal name `benzene_ring(C1,C2,C3,C4,C5,C6)` with the actual name `benzene_ring(c1,c2,c3,c4,c5,c6)` and then create an instance `T` of the benzene_ring clauses by applying the *mgu*.

We are now ready to describe how theories can be combined into larger ones. A basic primitive is the meta-predicate `add/1` to extend the current theory with the (instantiated) clauses of another theory. The meta-predicate is used inside a *meta-rule*, which is defined as follows:

Definition 2 (Meta-rule). A *meta-rule* is of the form:`- B.` with `B` an atom of a meta-predicate.

A meta-rule performs an action on the current theory. It has two hidden arguments: the current theory as defined by the preceding clauses and meta-rules, and the theory resulting from applying the operation as defined by the meta-atom `B`.

Example 8. A small theory about a benzene molecule can be constructed by combining an instance of the `benzene_ring` theory of Example 7 with the theory `bg` of Example 6:

```
begin_example benzene
  :- add(benzene_ring(c1,c2,c3,c4,c5,c6)).
  :- add(bg).
end_example
```

The first meta-rule adds the instantiated facts describing a benzene ring to the empty theory, the second meta-rule further extends that theory with the general properties of molecules as defined by the predicates `contains_double_bond/0` and `atom_count/1`. The resulting theory is as follows:

```
  atom(c1, c).  bond(c1, c2, aromatic).
  atom(c2, c).  bond(c2, c3, aromatic).
  ...
  atom(c6, c).  bond(c6, c1, aromatic).
  contains_double_bond :- bond(_,_,2).
  atom_count(N) :- findall(A,atom(A,_),L), length(L,N).
```

Example 9. The example theory about the naphthalene molecule (figure 2) can be constructed as follows:

```
begin_example naphthalene
  :- add(benzene_ring(c1,c2,c7,c8,c9,c10)).
  :- add(benzene_ring(c3,c4,c5,c6,c7,c2)).
```

```
  :- add(bg).
end_example
```

Note that c2 and c7 occur in both benzene rings of naphthalene. This ensures that the two rings are properly connected.

More formally, a meta-theory can be defined as follows:

Definition 3 (Meta-theory). A *meta-theory* is a *theory* where the sequence of clauses includes one or more meta-rules.

Similarly as for theories and depending on the chosen keywords, a meta-theory defines a meta-clause for one of the meta-predicates `example/2` and `theory/2` where the first argument is the name and the second argument is the list of clauses. This list of clauses is the *target theory* or *expanded logic program*, which is the result of executing the meta-rules in the *source theory* as formulated by the user.

3.2. Translation to meta-programs

In this section, we explain the precise meaning of a (meta-)theory. More specifically, we show how the source of a (meta-)theory can be translated into a meta-program (Barklund, 1995; Hill and Gallagher, 1998) that, given the name of a (meta-)theory, returns the expanded logic program as a list of clauses.

As already mentioned, there are two important meta-predicates: `theory/2` and `example/2`. Each theory gives rise to a so-called *defining clause* for one of these predicates.

Definition 4 (Defining clause). Let `name` be the name of a theory. The *defining clause* of `name` is the clause `theory(name,List):- Body` (`example(name,List) :- Body`, if the theory is an example) where `Body` is such that a call `theory(name`θ`,L)` (`example(name`θ`,L)`) binds L to the list of clauses of the expanded logic program that corresponds to `name`θ.

In the following we focus on the `theory/2` predicate; the `example/2` predicate is treated similarly.

The clauses of the expanded logic program, as returned by a call to the predicate `theory/2`, are represented as `clause(Head,Body)`, with `clause/2` the meta-predicate used for encoding clauses: `Head` is instantiated to the head of the corresponding clause and `Body` is instantiated to the body. In the following we will not distinguish between a clause and its associated `clause/2`-fact at the meta-level.

We now describe how a theory with name `name` is translated into a defining clause for `theory(name,Theory)`. When a theory contains no meta-rules, the translation is simple. The theory is initialised as an empty list and each clause is translated in a call to the `append/3` predicate to extend the current theory with the one element list holding the `clause/2`-term corresponding to a clause of the theory. For the `h2o` theory, we have[2]:

```
theory(h2o,Theory) :- T0 = [],
  append(T0,[clause(atom(h1,h),true)],T1),
  append(T1,[clause(atom(h2,h),true)],T2),
  append(T2,[clause(atom(o1,o),true)],T3),
  append(T3,[clause(bond(h1,o1,1),true)],T4),
  append(T4,[clause(bond(h2,o1,1),true)],Theory).
```

In the previous subsection we saw that parameters in a theory are represented as variables in the name of a theory. This is also the case in the meta-program obtained by translating a parametric theory. The meta-program for Example 7 is:

```
theory(benzene_ring(C1,C2,C3,C4,C5,C6),Theory) :- T0 = [],
  append(T0,[clause(atom(C1,c),true)],T1),
  append(T1,[clause(atom(C2,c),true)],T2),
  ...
  append(T10,[clause(bond(C5,C6,aromatic),true)],T11),
  append(T11,[clause(bond(C6,C1,aromatic),true)],Theory).
```

As can be seen from this translation, parameters in a theory are global to the whole theory. Since parameters are also represented as variables in the meta-program, constructing an instance of a parametric theory is simply a matter of unification.

Meta-rules in meta-theories are treated in a similar way as clauses. However, instead of using the `append/3` predicate to extend the current theory, they use the operation defined by the meta-predicate in the body of the rule to extend it. This is achieved by calling the meta-predicate with the current and new theory as extra parameters.

For example, the theory of Example 8 is translated into the meta-program

```
example(benzene,Example) :- T0 = [],
  add(T0,benzene_ring(c1,c2,c3,c4,c5,c6),T1),
  add(T1,bg,Example).
```

The `add/3` meta-predicate is predefined as

```
add(Tin,Name,Tout) :-
  (theory(Name,Theory);example(Name,Theory)),
  append(Tin,Theory,Tout).
```

Note that an *mgu* obtained by unification of the actual name `na` in a meta-call `add` and the formal name `nf` of a theory is not only applied on the theory defined by `nf`, but also on the meta-theory containing the meta-call (hence could instantiate that meta-theory if a variable occurring in `na` also occurs in other parts of the same meta-theory).

3.3. Some more elaborate examples

Tasks as described in Example 4, where a new theory is derived from an existing one by performing an action on a state, require a set of meta-predicates providing a full range of meta-programming facilities: the ability to query a theory, to select clauses from a theory, to delete clauses from a theory, *etc*. While one could provide a small library with useful predicates, complex applications will require that the user extends it with application-specific predicates. As an example, we sketch a Go application where one wants to formalize different states of a game as the examples to be used by the ILP system. Go is an abstract two-person complete-information deterministic board game like chess and draughts, popular in Asia (see Kim and Soo-hyun, 1997, for an introduction).

Example 10. The initial state (possibly already including some opening moves) can be described by a theory `state(s0)` that contains all relevant facts about the initial state of the game.

Assume the next example is the state resulting from playing a black stone on the fourth column of the third row. The following meta-theory defines it:

```
begin_example state(s1)
  :- add(state(s0)).
  :- update_state(stone(black,3,4)).
end_example
```

The example `state(s1)` is initialised with the logic program of the preceding state. Playing the move is much more involved than adding a fact that gives the position of the new move. One has to calculate and remove the stones that are captured as a result of the move. The position of the new stone is passed as argument of a new user defined meta-predicate `update_state/1`. The example is translated into:

```
example(state(s1),Example) :- T0 = [],
  add(T0,state(s0),T1),
  update_state(T1,stone(black,3,4),Example).
```

where `update_state/3` is a simple translation of the user-defined predicate `update_state/1`: two arguments are added for the input and output state (which are represented as lists of clauses). This predicate performs all the necessary calculations, starting with the preceding state `T1` and the position `stone(black,3,4)` of the new stone.

In some domains, there can be several ways to define a particular meta-theory. For example, in a game where the moves are reversible, one can also define:

```
begin_example state(s1)
  :- add(state(s2)).
  :- undo_move(stone(white,4,15)).
end_example
```

Alternative definitions for a theory `name` result in a number of different defining clauses for it. It is the user's responsibility to ensure the logical equivalence of the alternative definitions. The user must also ensure that there exists a partial order between theories that leads to a correct expansion of all theories (it must be possible to break circular dependencies by using an alternative definition).

When the ILP system needs the expanded program of a theory, the system can, based on which expanded programs are available, choose the alternative that offers the lowest cost.

3.4. Schemas

As a final extension, allowing the user to formulate more concisely a number of very similar (meta-)theories, we introduce the concept of schema.

Reconsider the game of Go as it was formalised in the previous section. All of the `state(si)` examples subsequent to the initial state are defined identically: each definition consists of a meta-rule for adding the preceding state and a meta-rule for executing a certain move.

Instead of defining an example `state(si)` for each state, it would be a lot more concise to be able to define a schema that creates all these examples at once.

We therefore define a theory `moves` that contains all the successive moves of the game. A move in the game adds a stone on a certain position.

```
begin_theory moves
  move(s0,s1,stone(black,3,4)).
  move(s1,s2,stone(white,4,15)).
  ....
end_theory
```

Using a meta-predicate `demo(T,Q)` for evaluating a query `Q` in the theory `T`, we can then define the example states by the following schema:

```
for_each (S,C,M) in :- demo(moves,move(S,C,M)).
begin_example state(C)
  :- add(state(S)).
  :- update_state(M).
end_example
```

This gives rise to examples such as

```
begin_example state(s1)
  :- add(state(s0)).
  :- update_state(stone(black,3,4)).
end_example
```

Note that the `for_each` construct generates bindings for three variables and that only one of these is used to construct the name of the example. The other bindings are used to instantiate the body of the schematic example.

More formally, a schema consists of

- a `for_each` construct:
  `for_each (X1,...,Xk) in :- B1,...,Bl.`
  with `X1,...,Xk` variables and `B1,...,Bl` (meta-)atoms,
- a (meta-)theory `T`; the name and the clauses/meta-rules in `T` may contain the variables `X1,...,Xk`.

A schema, defining a set of (meta-)theories, is dealt with in a pre-processing phase. Each instance of the variables `X1,...,Xk` that is an answer of the associated query `:- B1,...,Bl.` gives rise to a substitution that is applied to `T` (instantiating its name as well as its clauses/meta-rules). Each of these instances is then translated. The translations of the different instances are very similar, hence (see Section 4.1) it is advantageous to store the code of each instance as a set of bindings and a pointer to the generic code. Pre-processing includes the evaluation of the query in the `for_each` construct. If this query refers to (the expanded programs of) other theories (using `demo/2`) then those expanded programs need to be constructed. So if used unlimited, the pre-processing may have a substantial cost.

We conclude this section with another example of the use of a schema. Assume we have several theories about drugs `drug(di)` and patients `patient(pj)` (Example 2), and each example describes a specific patient treated with a particular drug. Instead of writing down these examples for each drug-patient combination, a schema allows us to define all these examples at once:

```
for_each (D,P) in :- theory(drug(D)), theory(patient(P)).
begin_example case(D,P)
  :- add(drug(D)).
  :- add(patient(P)).
end_example
```

In this way, an example `case(di,pj)` is defined for each pair of values `(di,pj)` that is an answer to the query `:- theory(drug(D)), theory(patient(P))`. The meta-predicate `theory/1`, used here to query theory names, can be defined as `theory(Name) :- theory(Name,_)`.

3.5. Summary

We have introduced a knowledge representation formalism where examples are described by theories. A theory is defined either explicitly, by listing the clauses defining a logical model for the example, or by means of a meta-theory that explains how to construct the theory from other theories using meta-rules. Meta-rules make use of meta-predicates such as the general `add` (merging theories), or application-specific `update_state`, `undo_move`, or

yet other meta-predicates that can be defined by the user. Finally, schemas provide a means of formulating concisely a set of similar meta-theories, using the `for_each` construct and parameterized theory names.

Looking at our framework from the ILP point of view, its contribution is as follows. Traditionally a distinction is made between examples and background knowledge. That distinction imposes a relatively simple structure on the knowledge base, as is shown in figure 1(c). Our framework allows the user to impose a more sophisticated structure on the knowledge base, which is then exploited by the learning system, rendering the latter more efficient.

4. A knowledge base graph

In this section and in Section 5 we discuss the algorithmic layer of our framework. Here we introduce the notion of a *knowledge base graph* (*KBG*). The *KBG* is a hypergraph that visualizes the dependencies between theories, more specifically, which theories can be constructed from which other theories. In Section 5, we will use the *KBG* to define the order in which the ILP system will process the examples.

4.1. Building a knowledge base graph

A *KBG* can be formalised as a directed hypergraph. In Gallo et al. (1993) the latter is defined as:

Definition 5 (Directed hypergraph). A directed hypergraph $\mathcal{H}$ is a pair $(N, \mathcal{E})$ where N is a set of nodes and $\mathcal{E}$ is a set of hyperedges. Each hyperedge ϵ is a tuple (S, n) from a source set $S = \{n_1, \ldots, n_k\} \subseteq N (k \geq 0)$ to a single target node $n \in N$. Given a hyperedge $\epsilon = (S, n)$, $source(\epsilon)$ is defined as S and $target(\epsilon)$ as n.

The knowledge base graph (*KBG*) of a knowledge base is a directed hypergraph. The nodes correspond to the theories stored in the knowledge base. A node is labeled with the name of the corresponding theory. For each theory definition T, with T^n the name of the theory and T^c its defining clause (cf. Definition 4), there is a hyperedge ϵ with $target(\epsilon)$ the node labeled T^n and $source(\epsilon)$ the nodes of the theories mentioned in the body of T^c. The hyperedge is labeled with T^c. This label is the recipe that has to be used to construct the expanded logic program of T. The recipe is executed by calling the predicate `theory/2` (or `example/2`). Executing the call in turn calls the recipes for the source nodes of the theory. If not available, they have to be constructed, hence the *KBG* is a structure that can be used to guide the construction of the examples stored in the knowledge base.

Theories without meta-rules do not refer to other theories. The corresponding hyperedges have empty sources. We call nodes that are the target of at least one such hyperedge *explicit* nodes; these can be constructed without accessing other theories. Other nodes are called *implicit* nodes. Formally:

Definition 6 (Explicit and implicit nodes). The set of explicit nodes $N_E \subseteq N$ is defined as $\{n \in N \mid \exists\, \epsilon \in \mathcal{E}, target(\epsilon) = n, source(\epsilon) = \emptyset\}$. The set of implicit nodes $N_I \subseteq N$ is defined as $N - N_E$.

Example 11. Consider again the benzene/naphthalene example. The corresponding *KBG* is shown in figure 3. There are two explicit nodes `benzene_ring` and `bg` and two implicit nodes `benzene` and `naphthalene`. The expanded logic programs of the explicit nodes `benzene_ring` and `bg` can be constructed by executing the defining clause associated with their incoming hyperedge. The defining clauses associated with the incoming hyperedges of the implicit nodes `benzene` and `naphthalene` are only executable when the expanded logic programs of their source nodes `benzene_ring` and `bg` are available.

Note that the code labeling a hyperedge encompasses calls to meta-predicates (system defined ones such as `add/1`, and user defined ones such as `update_state/1`), but not their definitions. The latter is a kind of meta-background-knowledge stored outside the knowledge base graph and available when constructing theories.

As already alluded to in Section 3.4, a single schema generates different hyperedges connecting different nodes, with code that differs only in the bindings for the variables in the `for_each` construct. It is therefore advantageous to store the meta-code of such

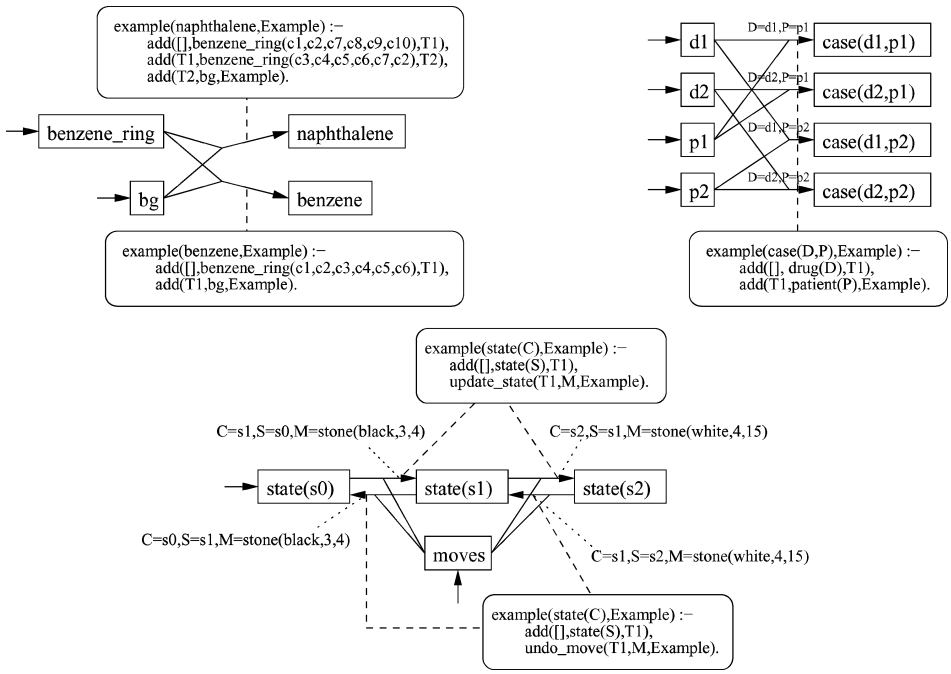

Figure 3. Labelled hypergraphs for benzene and naphthalene; drug-patient combinations; and game states. (Labels of edges defining explicit nodes are not shown.)

hyperedges as a pair containing the bindings for these variables and a call to the schema code, so the latter is shared by all instances of the schema defined by the `for_each` construct.

Example 12. The *KBG*s of the examples introduced in Section 3 (benzene and naphthalene, drug-patient combinations, consecutive game states) are shown in figure 3.

4.2. Some typical KBG structures

1. The setting from figure 1(b) (learning from interpretations without background knowledge) corresponds to a *KBG* where all nodes are explicit ($N_E = N$ and $N_I = \emptyset$). All edges have empty sources and all examples are explicit theories; there is no sharing.
2. The use of a common background theory as in figure 1(c) can be modeled by an explicit node with the background theory and an implicit node for each example. The code for an example consist of the facts about the example and a meta-rule for adding the background. All implicit nodes have the same source node.
3. When learning from episodes as in figure 1(e), we have one initial state for each episode i. Initial states $s_{i,0}$ are represented as explicit nodes. Consecutive states $s_{i,j}$ ($j > 0$) correspond to implicit nodes. The code in the hyperedges describes how each implicit state is computed from its predecessor state.
4. Sometimes a state can have several successors. For example, to represent moves in a game (sub)tree such as an alpha-beta search tree or to represent possible actions in reinforcement learning. In all these settings, the *KBG* is a set of trees. If different actions can lead to the same state, then the *KBG* becomes a directed acyclic graph. If an action sequence can reach the same state several times then the *KBG* becomes a directed graph (with cycles).
5. A set of trees is also obtained when examples share common substructures, as in the molecules example (Example 1).

5. Traversing the *KBG*

In order to be practically useful, a compactly represented knowledge base should satisfy two key properties. The first one is transparency: the interface for interacting with the knowledge base should be the same as when all examples are represented explicitly. Transparency implies that existing ILP systems can be adapted easily to use a *KBG*-based knowledge base. The second property is computational efficiency: some loss of efficiency when using the *KBG* may be unavoidable, but it should remain limited.

An ILP system queries examples to guide its search for the best hypothesis. Its overall execution time depends on two factors: the time to load an example in memory (if it is not yet in memory) and the time for querying the example. As discussed before, our method minimizes the query execution time by creating an environment in which only a small relevant part of the knowledge base is visible, and ensuring that this relevant part is in main memory. The price paid for this is the cost of constructing this environment (we will usually say "constructing the example"), using some defining clause, from the existing theories, which may or may not have to be loaded from disk. To minimize this cost, it is important

to avoid as much as possible that the same theories are loaded or constructed several times. This section discusses methods for achieving this goal.

5.1. *ILP algorithm interface*

We call the interface component between the ILP algorithm and the *KBG* the *example iterator*. The task of the example iterator is to traverse the *KBG* and to construct the expanded logic programs that the ILP algorithm needs to query.

Typically, the ILP algorithm has a set Q of one or more queries that it wants to run on a set of examples E. It provides this set E to the example iterator, which then constructs and provides to the ILP algorithm, one by one, all the examples in E. The ILP algorithm can then process these examples. When Q consists of multiple queries, one has the choice of iterating over E for each single query, or iterating once over E and running all queries on each example: this is the *queries outer loop* versus *examples outer loop* choice (Mehta, Agrawal, and Rissanen, 1996; Blockeel et al., 1999). Obviously, the more expensive example construction is, the more advantageous the examples outer loop version will be.

We assume without loss of generality that the order in which E is traversed is irrelevant to the ILP algorithm, so that the example iterator is free to choose a computationally optimal order. If the order is important, the ILP algorithm can present subsets of E, singletons if necessary, in the order that it wishes to process them. Generally, the larger the sets E that the ILP algorithm provides, the more efficient the examples can be traversed. So-called full scan algorithms, which scan the whole database a limited number of times (e.g., WARMR (Dehaspe and Toivonen, 1999), TILDE-*LDS* (Blockeel et al., 1999)), therefore have an advantage over algorithms that repeatedly query subsets of examples (e.g., PROGOL (Muggleton, 1995), TILDE (Blockeel and De Raedt, 1998)).

5.2. *A planning problem*

Given a set E of example names (a subset of the nodes in the *KBG*) required by an ILP algorithm, the task of the example iterator is to construct the expanded logic program for each example node $n \in E$ so that the ILP algorithm can query it. The example iterator must traverse the *KBG* in some optimal order to accomplish this with a minimal cost.

We say that a node n is *active* if the expanded logic program of the theory with name n is present in the main memory, ready for querying by the ILP algorithm (and for computing the expanded logic programs of other theories). The example iterator can *activate* a node n if there exists a hyperedge ϵ with n as target node and for which all nodes in the source set are active. If so, the expanded logic program of n can be computed by loading the defining clause that is the label of ϵ and calling `theory/2` (or `example/2`) with first argument n.

In what follows, activation of a node (i.e., computation of its expanded logic program) is modeled as $expand(\epsilon, n)$. The reverse action, removing the expanded logic program of theory n from main memory, is modeled as $remove(n)$. While the cost of removing a node can be ignored, expanding a node n has a certain cost: the time necessary for loading and executing one of the defining clauses available for n, and possibly compiling the expanded logic program (which may be beneficial if it will be queried many times).

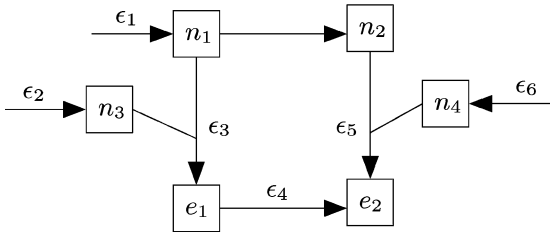

Figure 4. A planning problem.

The task of the example iterator is to minimize the total cost of constructing all the examples, while observing the constraint that the set of active nodes must fit in main memory at all times. This can be formalised as a planning problem. We describe the behavior of the example iterator as a sequence $a_1, a_2, \ldots, a_n$ of *expand* and *remove* actions. Using the set NA_i to denote the set of active nodes after performing action a_i (NA_0 denotes the empty set of initially active nodes), $cost(a_i)$ to denote the cost of action a_i and $mem(n)$ to denote the memory occupied by the expanded logic program of theory n, the planning problem is as follows:

Definition 7 (Planning problem). Given a set of examples E that the ILP algorithm needs to query, find a sequence of actions a_i such that:

1. (Objective) $\sum_{a_i} cost(a_i)$ is minimal.
2. (Memory constraint) $\forall i \sum_{n \in \mathrm{NA}_i} mem(n) \leq$ the maximal memory.
3. (ILP algorithm constraint) $\forall n \in E,\ \exists i$ such that $n \in \mathrm{NA}_i$.

Example 13. Consider the *KBG* shown in figure 4. Suppose that the ILP algorithm needs to query the examples $E = \{e_1, e_2\}$, that each theory occupies one memory unit and that three memory units are available. The sequence $\{expand(\epsilon_1, n_1), expand(\epsilon_2, n_3), expand(\epsilon_3, e_1), remove(n_3), expand(\epsilon_4, e_2)\}$ is a plan that satisfies both the memory and the ILP algorithm constraint. Whether or not this plan is optimal depends on the costs associated with the actions. Indeed, e_2 can also be constructed from n_2 and n_4 instead of e_1.

The above planning problem is in general NP-hard, and so is the related problem of determining the minimal amount of memory necessary to be able to traverse a *KBG*. This is shown in Appendix A . We proceed with proposing a heuristic planning algorithm that is fast and usually finds a good solution.

5.3. A heuristic example iterator

We need a heuristic algorithm that scales well in the number of nodes of the *KBG* and results in a plan with a near to optimal cost in frequently occurring cases. To achieve this,

we proceed in two steps. First, a so-called spanning forest is computed. Next, the spanning forest is used to construct a plan that is a solution for the planning problem of Definition 7.

Definition 8 (Spanning forest). A hypergraph is a *forest* if it is an acyclic hypergraph and if each node is the target of one hyperedge. With E a subset of the nodes of the *KBG*, a *spanning forest* for E is a forest $(N, \mathcal{E})$ with $\mathcal{E}$ a subset of the edges of the *KBG* and N a subset of its nodes such that $E \subseteq N$ and that $(N, \mathcal{E})$ is minimal, i.e., no $(N', \mathcal{E}') \neq (N, \mathcal{E})$ selected from *KBG* is a forest with $E \subseteq N'$, $N' \subseteq N$ and $\mathcal{E}' \subseteq \mathcal{E}$.

In a spanning forest, a node n is the target of exactly one hyperedge (S, n). We call S the set of *parents* of n, denoted *parents*(n). If S is empty, we call n a *root*, otherwise n is a *child* of each element of S. As a node can occur in the sources of several hyperedges, it can have several children; the set of children of a node n is denoted *children*(n). A node without any children is called a *leaf*. Note that the roots of a spanning forest are always explicit nodes of the *KBG*.

Consider a *KBG* $(N, \mathcal{E})$. Computing a spanning forest for a set $E \subseteq N$ can be done in time $\mathcal{O}(|N|)$. However, the spanning forest is not necessarily optimal. Given the costs of all hyperedges, an optimal spanning forest (the sum of the costs of the hyperedges is minimal) can be computed in time $\mathcal{O}(|\mathcal{E}| \log |N|)$, using an adapted version of Prim-Jarník's algorithm (Goodrich and Tamassia, 2002). This requires that enumerating all edges is feasible.

5.3.1. A generic approach. Given sufficient memory for storing the active nodes, there is a simple algorithm for computing an optimal plan that activates each node of a spanning forest exactly once. For each node one keeps a counter in that indicates how many of its parents have not yet been activated, and a counter out that indicates how many of its children still need to be activated. Any node with $in = 0$ can be activated, and any node with $out = 0$ is not necessary anymore and can be removed. One thus activates nodes in the forest from roots towards leaves.

The algorithm as described above is non-deterministic in the selection of the next node to be activated, and removes nodes only when they are not necessary anymore. Different selection strategies for the node to be expanded will result in different memory requirements. A depth first strategy is optimal for simple structures (e.g., with sequences, only 2 nodes will be active at the same time), but with more complex structures, more involved strategies are required. When memory is limited, it may be necessary to remove nodes for which $out > 0$ and re-activate them later on. A node removal strategy could take into account the memory size of candidate nodes, the cost of re-activating them, and the time point at which they will be needed again.

5.3.2. A depth-first algorithm. Figure 5 presents an example iterator algorithm with a depth-first node selection strategy. The Traverse_sub_tree procedure, when called for a given node, activates that node if it has never been activated before (ensuring that each node in the hypergraph is activated at least once), and then selects one of its children for activation. A node can only be activated if all its parents are active, so we select that child that currently has the fewest inactive parents. We have to activate these parents as well, and all

```
procedure Traverse_forest((N, ℰ))
    for each node n ∈ N:
        activated(n) := active(n) := visited(n) := false
        protected(n) := 0
    for each root node r ∈ N:
        Traverse_sub_tree((N, ℰ), r)

procedure Traverse_sub_tree((N, ℰ), n)
    if visited(n) = false then
        visited(n) := true
        if activated(n) = false then Activate_node((N, ℰ),n)
        C := children(n)
        while C ≠ ∅:
            select c ∈ C such that
                |{m|m ∈ parents(c) and active(m) = false}| is minimal
            Traverse_sub_tree((N, ℰ), c)
            C := C \ {c}
        active(n) := false
        remove(n)

procedure Activate_node((N, ℰ), n)
    Let ϵ = (P, n) be the hyperedge in ℰ with target n
    for each p ∈ P:
        if active(p) then protected(p) := protected(p) + 1
    for each p ∈ P:
        if not active(p) then
            Activate_node(p)
            protected(p) := protected(p) + 1
    while insufficient memory for expand(ϵ, n)
        select a node d with protected(d) = 0
        active(d) := false
        remove(d)
    expand(ϵ, n)
    activated(n) := active(n) := true
    for each p ∈ P:
        protected(p) := protected(p) − 1
```

Figure 5. An example iterator algorithm with a fixed node selection strategy that proceeds depth first, starting from each root node.

their currently inactive ancestors, before the child itself can be activated. The Activate_node procedure takes care of activating a node and (through recursion) all its inactive ancestors.

Our algorithm makes use of a number of flags and counters. The flag *visited* indicates that a node has already been visited by Traverse_sub_tree and need not be processed again (this flag is necessary because, even though in a spanning forest a node is the target of a single hyperedge, it can be reached several times through each of its parents). The flag *activated* indicates that a node has been activated at least once, the *active* flag that it is currently active (i.e., has not been removed since its last activation). Finally, the counter *protected* counts for an active node the number of reasons why it cannot be removed. Activate_node adds one

to the *protected* counters of all the parents of the node it is activating, ensuring to protect active parents before activating inactive parents (as the latter process might otherwise cause active parents to be removed).

When the available memory for expansion of a logic program is insufficient, unprotected nodes are selected and removed. These nodes may have to be reactivated at some later point. If there are insufficient unprotected nodes, the memory demand supersedes the available memory and the algorithm fails (the test is omitted for simplicity of presentation).

5.3.3. Exploiting specific structural properties of graphs. Of special interest are graphs where it can be guaranteed that each node is activated only once, with memory requirements linear in the height of the graph; all duplicate work with respect to example construction is then avoided. For graphs with a tree structure, where each example e_i can be constructed from its predecessor e_{i-1}, this is always possible. More generally, we can consider graphs where the nodes N can be partitioned into cheap (N_C) and expensive (N_E) nodes; a node is cheap if the total time to activate it (including the activation of its ancestors) is bounded by some small constant T_C. If each expensive node has at most one expensive parent, then it is possible to traverse the graph in such a way that all expensive nodes are activated only once. To achieve this, it suffices to take care that Traverse_sub_tree is never called for an expensive node that has an inactive expensive parent. Figure 6 presents an alternative version of Traverse_forest that guarantees this. This is the version that we use for our experiments.

5.3.4. Implementation issues. In this section we discuss a number of implementation choices that have been made in the algorithm that we use for our experiments.

Multiple passes. The Traverse_sub_tree function deactivates a node after all its children have been processed. However, an ILP system usually performs several passes over the examples, so a given node may be required again in a subsequent pass. Therefore, our implementation does not deactivate such nodes and relies on the node removal strategy called in the Activate_node function to free memory.

procedure Traverse_forest$(N_C, N_E, \mathcal{E})$
  for each node $n \in N_C \cup N_E$:
    $activated(n) := active(n) := visited(n) := false$
    $protected(n) := 0$
  for each $n \in N_E$ with $activated(n) = false$
    while n has parent p and $p \in N_E$: $n := p$
    Traverse_sub_tree$((N_C \cup N_E, \mathcal{E}), n)$
  for each $n \in N_C$ with $activated(n) = false$
    while n has parent p: $n := p$
    Traverse_sub_tree$((N_C \cup N_E, \mathcal{E}), n)$

Figure 6. An example iterator algorithm that guarantees that expensive nodes are activated only once and that requires memory linear in the height of the graph. (The algorithm uses the procedure Traverse_sub_tree$((N, \mathcal{E}), n)$ defined in figure 5.)

Before each pass, our implementation partitions E in the set of example nodes that are still active from a preceding pass E_a and the inactive nodes E_i. It first queries E_a and then uses the iterator algorithm to activate E_i.

Node removal. Our implementation uses a least recently used strategy (LRU) to select nodes for removal. It first partitions the unprotected active nodes into two sets: the nodes on the subtree that is being iterated of which not all children have been visited S_1 and the other nodes S_2. If $S_2 \neq \emptyset$, it selects a node from this set using LRU. If $S_2 = \emptyset$, it selects the LRU node from S_1.

Sharing. If a theory T adds a number of theories T_i, our implementation also shares code between the expanded logic program of T and those of the T_i, i.e., the expanded logic program of T is represented by a set of pointers to the expanded logic programs of the T_i.

Following these pointers during query execution may introduce overhead. If T and T_i are predicate-disjoint, then this overhead disappears: indexing on the predicate symbols will bring execution immediately to the expanded logic program that defines the predicate.

6. Experimental evaluation

6.1. Aims

Our experimental evaluation aims at gaining more insight in the performance of our framework for compactly representing knowledge bases in practical application domains.

In our experiments, we compare three representations of the knowledge base: a monolithic logic program (setting 1, figure 1(a)), the learning from interpretations setting (setting 2, figure 1(c)), and a representation that uses our new framework (setting 3).

We compare for each setting the size of the knowledge base on disk and the main memory usage and execution time of an ILP system (TILDE, see further).

6.2. Data sets and knowledge base structure

We consider three application domains where there is a significant difference between the three settings.

Cancer. The human tumor cell line screen database (DTP, 2003) has a two-dimensional structure: one dimension stores information about 60 human cancer cell lines and the other dimension the molecular structure of 53933 chemical compounds.

In setting 1, the knowledge base consists of facts that describe the cell lines (with key `C`), facts that encode the molecular structure of the compounds (with key `M`) and a number of `target(M,C,`GI_{50}`)` facts, with GI_{50} the target attribute for prediction (GI_{50} is the concentration of `M` that causes 50% growth inhibition for `C`).

In setting 2 there is one interpretation for each available GI_{50} value. It includes the facts describing the relevant cell line and molecule. As each cell line (molecule) is included in several interpretations, this representation will be redundant.

The compact version includes an auxiliary theory for each cell line and molecule, and an example theory for each GI_{50} value. The example theories are constructed by a schema and meta-rules are used to add the relevant cell line and molecule theory.

HIV. The AIDS anti-viral screen database (DTP, 2002) stores the molecular structure of 41768 compounds that were measured for their capability to protect human cells from HIV-1 infection.

In setting 1, the knowledge base consists of a number of facts that encode the molecular structure of the compounds.

Setting 2 is the same as setting 1, except that the facts describing each compound are grouped together in an interpretation.

Setting 3 includes an example theory for each compound. Frequently occurring structures, such as benzene rings, are stored separately as parametric theories and are included in the molecule definitions via meta-rules.[3] We have defined a number of these structures manually (Struyf et al., 2004).

Go. Go is an abstract two-person complete-information deterministic board game, popular in Asia. The Go data set (Ramon, Francis, and Blockeel, 2000) that we use here contains a log generated by an alpha-beta search algorithm. Starting from a number of initial states, the search algorithm explores the game state-tree and evaluates a number of moves for each state.

In setting 1, the states are described by a number of predicates that include a state key attribute. The moves are encoded with facts that include the state key and the position and value of the move.

In setting 2, each move corresponds to an interpretation that contains the move's position and value and also the description of the relevant state. As the same state may be included in several move interpretations, this representation is redundant.

Setting 3 includes one example theory for each move. Using a meta-rule, each move theory includes the relevant state theory. State theories are defined in terms of their predecessor state and a certain move. The definition of the meta-program that computes the states is available in Struyf et al. (2004).

6.3. Implementation

We have implemented our new formalism in the ILP system ACE 1.2.6[4] (Blockeel et al., 2002). Settings 1 and 2 were already available in ACE.

For each application, we run the first order decision tree induction system TILDE (Blockeel and De Raedt, 1998), which is included in ACE, for different sample sizes of the knowledge base. We measure the execution time and memory usage of TILDE in the *examples outer loop* setting (Section 5.1). Struyf et al. (2004) presents the same measurements for the more expensive *queries outer loop* setting.

Settings 2 and 3 use a fixed size block of main memory (the cache) for storing examples (interpretations/expanded logic programs). In each experiment, we set the cache size to 100 MB.

The experiments on the Cancer data were run on an Intel P4, 2 Ghz, 512 MB, the other experiments on an Intel P4, 1.8 Ghz, 512 MB.

Table 1. Size comparison (size on disk, in source format. N is the number of examples).

	N	S_1	S_2	S_3	S_1/S_3	S_2/S_3
Cancer	293176	44 MB	2710 MB	36 MB	1.2×	75×
HIV	41768	40 MB	30 MB	12 MB	3.5×	2.5×
Go	150000	207 MB	723 MB	13 MB	16×	55×

6.4. Results

6.4.1. Size of the knowledge base. Table 1 shows the knowledge base size S_i for each setting i, and the reduction factors S_1/S_3 and S_2/S_3.

The Cancer and Go data sets have two dimensions: cell lines and compounds, and states and moves. Such data sets can be represented compactly in setting 1 and 3. In setting 2, only a redundant representation is possible, which explains the huge S_2/S_3.

In the Go data set, most states are defined implicitly via a meta-theory in setting 3. Because the meta-theories are small compared to the explicit representation of the states, we obtain high reduction factors S_1/S_3 and S_2/S_3.

For HIV, the reduction S_2/S_3 relatively small. Apparently, the parametric structures cover only a small part of each molecule. S_1/S_3 is a bit larger because each fact in setting 1 includes an extra identifier.

6.4.2. Execution time and memory usage. Figures 7–9 show the execution time and memory usage of TILDE on each application. Each set of bars corresponds to a set of experiments for a different sample size N.

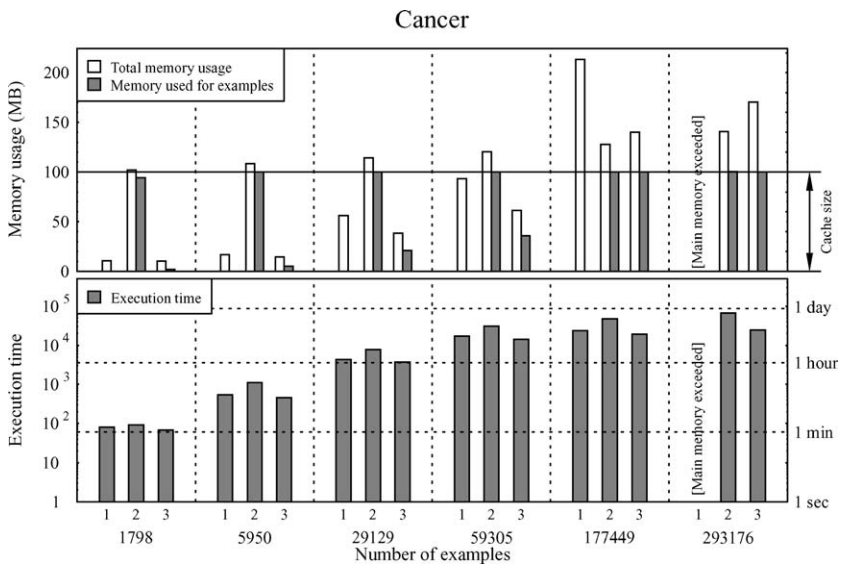

Figure 7. Memory usage and execution time for the Cancer knowledge base. For different sample sizes, results for setting 1, 2 and 3 are shown.

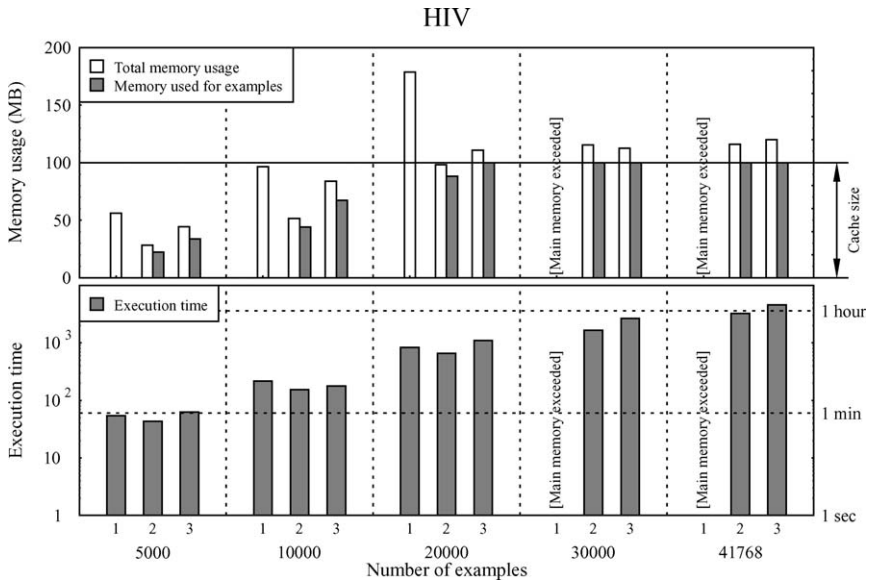

Figure 8. Memory usage and execution time for the HIV knowledge base.

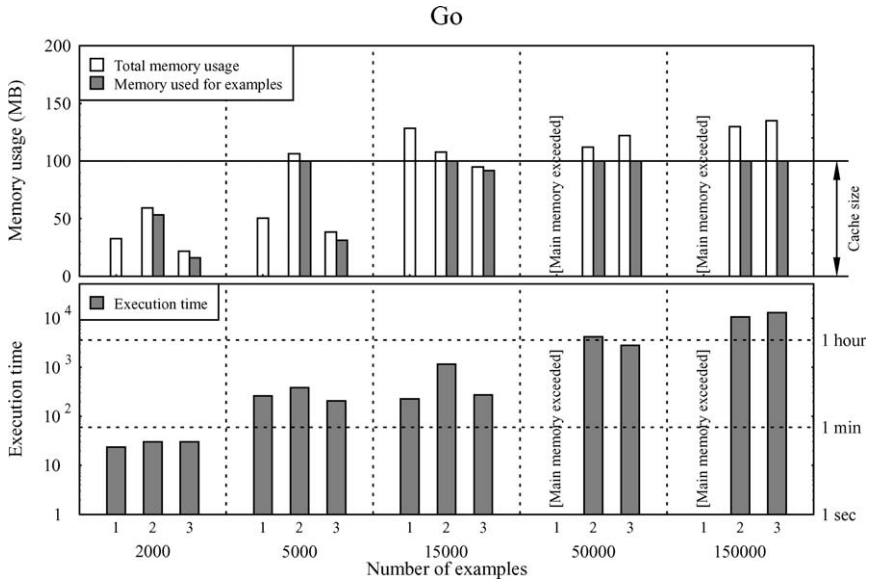

Figure 9. Memory usage and execution time for the Go knowledge base.

The total memory usage for setting 1 is linear in N. Above a given value of N, the logic program representing the examples does not fit in memory anymore and setting 1 is no longer feasible (the system starts swapping).[5] Settings 2 and 3 on the other hand can use an efficient caching mechanism (because of the modular representation) and scale well with N.

The execution time increases with N in all settings as more data must be processed. The difference in execution time between the settings is influenced by: (1) the trade-off between loading all data from disk and loading and constructing data with meta-programs, (2) the cache effect of having to load data several times if not all data fits in main memory, and (3) the extra indexing overhead in setting 1.

For Cancer, setting 3 is the most efficient setting. It is about a factor two faster than setting 2 because less data must be loaded and because the data must be loaded repeatedly in setting 2 if the cache is full. It is more efficient than setting 1 because query execution is faster (less indexing overhead). For HIV, setting 2 is the most efficient setting. Less data must be loaded in setting 3, but expanding the parametric structures takes more time than the gain obtained during loading. For Go, setting 3 is significantly faster than setting 2 for $5000 \leq N \leq 50000$ because of the cache effect. For $N = 150000$ and $N = 2000$, setting 2 is a more efficient as constructing examples is more expensive than loading them.

6.4.3. Summary. Our main findings are as follows. Setting 3 consistently yields the most compact representation of the data set. It is sometimes faster than the other settings, sometimes slower; this depends on how the repeated loading and construction of examples in setting 3, compares to the loading of larger examples in setting 2 and the more complex query evaluation in setting 1. For large data sets that do not fit in main memory, setting 1 becomes unusable.

In short, the experiments confirm that setting 3 combines the good scaling properties of setting 2 with a space efficiency better than that of setting 1, and time efficiency comparable to that of the other settings. This has yielded a performance improvement in two out of three data sets. For the HIV data set, setting 3 results in a small storage gain at the expense of a significant increase in processing time. Hence setting 2 is to be preferred here. Note that setting 2 can also be modeled with our approach.

7. Conclusions

The knowledge base from which an ILP system learns is usually treated as monolithic (a single logic program that is loaded into main memory), or structured into descriptions of examples (containing "local" information, relevant for this single example) and background knowledge (containing "global" information). The second approach has the advantage that querying a single example is more efficient, and that the whole knowledge base need not be loaded all at once in main memory (Blockeel et al., 1999), but this happens at the expense of duplicating information that is relevant for more than one example. Typical applications where this may be problematic, are the learning from episodes setting where consecutive states of a world have to be stored and applications that contain data about several dimensions.

In this article we have introduced a framework for reconciling the two goals of having information locally available and avoiding duplication of information. The framework makes it possible to structure the knowledge base in a more refined manner than by just distinguishing between knowledge relevant for one example, and background knowledge (relevant for all examples).

Our framework consists of two layers. The first layer is the knowledge representation layer. Using this layer, the user can define the knowledge base as a set of named (meta-)theories. The definition of a (meta-)theory can contain meta-rules and parameters. The meta-rules can be used to define how the examples can be constructed from other auxiliary theories.

The second layer is an algorithmic layer that supports efficient construction of the examples. If the meta-rules in a given meta-theory refer to a set of other (meta-)theories, then these (meta-)theories must be available before the new meta-theory can be constructed. This type of preconditions is expressed in our framework by the *knowledge base graph* (*KBG*).

Most ILP systems can be adapted easily to work with our framework. In order to accomplish this we have introduced an interface between the ILP system and the knowledge base: the *example iterator*. The example iterator has to construct the examples in an optimal order (based on the preconditions in the *KBG*). Finding this order corresponds in general to an NP-hard planning problem. We propose a heuristic algorithm that scales well in the number of nodes of the *KBG* and results in a plan with a near to optimal cost for *KBG* structures that occur often in practice.

We have evaluated our method on three example knowledge bases with quite different properties: the Cancer knowledge base has a two-dimensional structure, the HIV data set contains labeled molecular structures, and the Go data set contains states, moves and their evaluations for the board game Go. For each application we compared 3 representations of the knowledge base: one monolithic program, a theory for each example and a common background theory and our new framework. The experiments on the Go and Cancer knowledge bases show that our framework significantly reduces the storage requirements of the knowledge base. For the HIV application a smaller reduction was obtained. We also measured the execution time and main memory usage of the ILP algorithm TILDE for each representation. Because our framework supports on-demand loading of examples, it does not require much main memory. The obtained execution times vary: in some cases, our framework is faster, in other cases it is slower. The reason for these differences is that in our framework examples have to be constructed at runtime and in some cases the construction cost is high.

The reduction in size for the HIV application was smaller than for the other two applications. One possible way to improve on this is to try to find better patterns. Finding such patterns (with a high frequency, a long description and few parameters) is not trivial. Maybe this could be automated using a frequent pattern discovery (Dehaspe and Toivonen, 1999) or clustering approach.

Our method is formulated in the learning from interpretations setting, where examples are represented as logic programs (sets of clauses). Much work in ILP concerns the learning from entailment setting, where an example is represented as a clause with as head the instance of the predicate to be learned and as body the example specific facts. In addition,

there is a background theory shared by all examples. It is feasible to develop techniques similar to those we described for this setting. The background theory can be split in several theories and these theories can include meta-rules. The only extra thing that is required is a notation for linking theories to example clauses.

Expanding theories can be compared to saturation, which is used in some ILP systems. The representation as (meta-)theories is in principle sufficient to deduce all necessary information. However, ILP systems are typically not able to work with this information directly (and doing so would introduce much overhead). Saturation (Rouveirol, 1994) solves a special instance of this problem: some ILP algorithms cannot handle separate background knowledge. Therefore, they use saturation to expand the examples with the information that is implicit in the background knowledge. In our approach too, we have to expand examples in order to let the ILP system use them efficiently. While saturation can be seen as a special case, our approach is more general: saturation is essentially a process of specializing a logic program, expanding examples in our framework may involve other operations (deletions, updates, computations, etc.). Examples also need not be ground in our framework: they can include implicitly defined predicates. One can also be very specific about the operations that are performed, so that only what is necessary is computed, which contrasts with saturating a clause with all background knowledge.

Further research could try to improve the efficiency of constructing examples. Typical Prolog implementations do not offer primitives for efficiently manipulating compiled logic programs and repeated compilation introduces a large overhead. In this context, it could be interesting to look at incremental compilation of programs and at combining compiled programs at runtime. Another direction of research could look at knowledge bases that are lazily sampled from an implicit (infinite) example space.

Appendix A. Proof of NP-hardness of the planning problem

The planning problem of Definition 7 is in general NP-hard. This can be seen by considering a special instance that is equivalent with the Traveling Salesman Problem (TSP). Suppose that the *KBG* has one explicit node, a number of implicit nodes and for every node n_1 and every implicit node n_2 there is a hyperedge $(\{n_1\}, n_2)$. All nodes are examples for the ILP algorithm. If at most two active nodes fit in memory, the optimal plan corresponds to finding the shortest path through the *KBG* where the distances are given by the costs of the (hyper)edges.

Also, the time needed to determine (or "closely" approximate) the minimal amount of memory needed to traverse the *KBG* can not be bound by a polynomial in the number of nodes, in particular if there are many different alternative definitions for each theory (there exists a polynomial algorithm in the number of edges, but edges may be defined implicitly by a schema). This negative property can be seen as follows (figure 10). Consider the *KBG* $(N, \mathcal{E})$ and let $N = N_1 \cup N_2 \cup \{o\}$ with N_1 the set of explicit nodes and o the only node needed by the ILP algorithm. Moreover, let $\mathcal{E} = \mathcal{E}_1 \cup \mathcal{E}_2$ where $\mathcal{E}_1$ is a set of edges (S, d) with $d \in N_2$ and $S \subset N_1$ and $\mathcal{E}_2 = \{(S, o) | S \subset N_2 \wedge \forall n \in N_1, \exists e \in \mathcal{E}_1 : n \in source(e) \wedge target(e) \in S\}$. In other words, o can be constructed from any subset of N_2 that itself requires activation of all nodes in N_1. Assume that only nodes in N_2 take a substantial amount of memory when

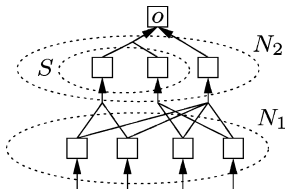

Figure 10. A set covering problem.

activated. Then, the task to minimize the amount of needed memory reduces to finding a minimal (weighted by memory usage) subset of N_2 that covers N_1. This "set covering" problem is known to be NP-complete (Paschos, 1997).

Acknowledgments

This work has been supported by the Fund for Scientific Research (FWO) of Flanders and by the GOA/2003/08(B0516), "Inductive Knowledge Bases". The authors thank Stefan Kramer for providing a pre-processed version of the HIV data.

Notes

1. We are discussing the problem in the context of learning from interpretations, but essentially the same options exist in the learning from entailment setting. In the latter, examples are clauses. The head of such a clause is an example literal of the predicate to be learned, the body contains facts relevant to this one example, and the background contains knowledge that may be relevant to any examples. The main difference is that in learning from entailment, the "locally relevant" information can only consist of facts, whereas in learning from interpretations it may consist of clauses. We return to this issue in Section 7.
2. We prefer clarity of presentation above efficiency; e.g., to avoid the overhead of append, one could use difference lists. Even better, one could specialise the code to a fact of the form theory(h2o,[clause(...),...,clause(...)]). Such optimisations are performed by our implementation.
3. A similar representation is possible for the compounds in the Cancer knowledge base. We choose not to do this as it would make interpretation of the results more complex.
4. `http://www.cs.kuleuven.ac.be/~dtai/ACE/`.
5. Setting 1 can be used with large N if the examples are stored in a RDBMS. However, Blockeel et al. (1999) shows that this introduces much overhead and that it is better to use a modular example representation.

References

Arni, F., Ong, K. Tsur, S. Wang, H., & Zaniolo, C. (2003). The deductive database system LDL++. *Theory and Practice of Logic Programming*, 3:1, 61–94.

Barklund, J. (1995). Metaprogramming in logic. *Encyclopedia of Computer Science and Technology*, 33, 205–227.

Blockeel, H., & De Raedt, L. (1996). Relational knowledge discovery in databases. In *Proceedings of the Sixth International Workshop on Inductive Logic Programming*, Vol. 1314 of *Lecture Notes in Artificial Intelligence* (pp. 199–212) Springer-Verlag.

Blockeel, H., & De Raedt, L. (1998). Top-down induction of first order logical decision trees. *Artificial Intelligence* 101:1-2, 285–297.

Blockeel, H., De Raedt, L., Jacobs, N., & Demoen, B., (1999). Scaling up inductive logic programming by learning from interpretations. *Data Mining and Knowledge Discovery* 3:1, 59–93.

Blockeel, H., Dehaspe, L., Demoen, B., Janssens, G., Ramon, J., & Vandecasteele, H. (2002). Improving the efficiency of inductive logic programming through the use of query packs. *Journal of Artificial Intelligence Research* 16, 135–166.

Cussens, J. (1997). Part-of-speech tagging using Progol. In *Proceedings of the Seventh International Workshop on Inductive Logic Programming* (pp. 93–108). Springer-Verlag.

Das, S. K. (1992). *Deductive databases and logic programming*. Addison-Wesley.

De Raedt, L., & Džeroski, S. (1994). First order *jk*-clausal theories are PAC-learnable. *Artificial Intelligence*, 70, 375–392.

Dehaspe, L., & Toivonen, H. (1999). Discovery of frequent Datalog patterns. *Data Mining and Knowledge Discovery* 3:1, 7–36.

DTP, The Developmental Therapeutics Program. U.S. Departement of Health and Human Services NIH, National Cancer Institute NCI. http://dtp.nci.nih.gov.

Džeroski, S., De Raedt, L., & Driessens, K. (2001). Relational reinforcement learning. *Machine Learning* 43, 7–52.

Gallo, G., Longo, G., Pallottino, S., & Nguyen, S. (1993). Directed hypergraphs and applications. *Discrete Applied Mathematics* 42, 177–201.

Goodrich, M. T., & Tamassia, R. (2002). *Algorithm design*. Wiley.

Hill, P. M., & Gallagher, J. (1998). Meta-programming in logic progamming. *Handbook of Logic in Artificial Intelligence and Logic Programming*, 5, 421–498.

Ito, M., & Ohwada, H. (2001). Efficient database access for implementing a scalable ILP engine. In *Work-In-Progress Report of the Eleventh International Conference on Inductive Logic Programming*.

Jacobs, N., & Blockeel, H. (2001). From shell logs to shell scripts. In *Proceedings of ILP2001—Eleventh International Workshop on Inductive Logic Programming*, Vol. 2157 of *Lecture Notes in Artificial Intelligence* (pp. 80–90).

Kim, J., & Soo-hyun, J. (1997). *Learn to play Go—A master's guide to the ultimate game*. Good Move Press.

Mehta, M., Agrawal, R., & Rissanen, J. (1996). SLIQ: A fast scalable classifier for data mining. In *Proceedings of the Fifth International Conference on Extending Database Technology*, Vol. 1057 of *Lecture Notes in Computer Science*. Springer-Verlag.

Morik, K., & Brockhausen, P. (1997). A multistrategy approach to relational discovery in databases. *Machine Learning*, 27:3, 287–312.

Muggleton, S., (1995). Inverse entailment and Progol. *New Generation Computing, Special issue on Inductive Logic Programming* 13:3/4, 245–286.

Muggleton, S., King, R., & Sternberg, M. (1992). Protein secondary structure prediction using logic-based machine learning. *Protein Engineering* 7, 647–657.

Paschos, V. (1997). A survey of approximately optimal solutions to some covering and packing problems. *ACM Computing Surveys* 29:2, 171–209.

Ramon, J., Francis, T., & Blockeel, H. (2000). Learning a Tsume-Go heuristic with Tilde. In *Proceedings of CG2000, the Second International Conference on Computers and Games*, Vol. 2063 of *Lecture Notes in Computer Science* (pp. 151–169). Springer-Verlag.

Rouveirol, C. (1994). Flattening and saturation: two representation changes for generalization. *Machine Learning*, 14, 219–232.

Struyf, J., Ramon, J. Verbaeten, S. Bruynooghe, M., & Blockeel, H. (2004). Compact representation of knowledge bases in inductive logic programming. Technical Report CW 377, Department of Computer Science, Katholieke Universiteit Leuven.

Received March 20, 2003
Revised January 19, 2004
Accepted June 14, 2004
Final manuscript June 17, 2004