Program extraction from quantified decision trees (Extended abstract)

Taisuke Sato
Tokyo Institute of Technology
sato@mi.cs.titech.ac.jp

1 Introduction

There are two approaches in the top-down construction of first order decision trees that use definite or normal clauses. The covering approach [5, 3] builds one clause at a time that covers part of positive examples, and repeats this process until no positive examples remain uncovered. The divide-and-conquer approach [1, 2] on the other hand divides the given examples into smaller subsets recursively to get ones explaining positive examples.

While the divide-and-conquer approach is conceptually simple, and dividing examples by a literal is a direct extension of the propositional case, there does not seem to be much work on such approach in ILP. Quinlan [5] employed division by a literal but within the covering approach. Boström and Idestam-Almquist [2] adopted division by unfolding, not by a literal. Blockeel and De Raedt [1] used division by a literal but without tuple extension and their synthesized clauses have a ground head to represent a class name.

In this paper, we propose a new divide-and-conquer approach to ILP to synthesize logic programs. It is based on the construction of *quantified decision trees* explained in the next section. Our approach is thought to be a logical refinement of FOIL [5] in which existentially quantified negative literals are used as well as existentially quantified positive literals. The synthesized clauses may include universally quantified formulas in their body but can further be compiled into definite clauses with disequality constraints by First Order Compiler [6].

2 Quantified decision trees

The basic idea of *quantified decision trees* (*QDT*s) is the same as that of FOIL. The difference lies in the fact that we allow for existentially quantified negative literals when tuples are divided and extended. A *quantified decision tree* for a predicate $p(z)$ is a tree in which the root node is a most general atom[1] representing a clause head, and is connected via the *top edge* to a binary subtree below it, representing the clause body. In the subtree, each node is labeled with a literal, or labeled “Positive”, “Negative” or “Mixed” when they are leaves. A set of tuples representing positive and negative examples is attached to each edge. A tuple t

[1]For an n-ary predicate p, a most general atom is one that has the form $p(z) = p(X_1, \ldots, X_n)$ where X_is are distinct variables.

is labeled + if it is a positive example or - otherwise. By construction, t is always an extension of some tuple t' in the initial tuple set attached to the top edge. When such relationship holds, we call t' the *founder* of t. Founders represent original positive and negative examples, and our purpose is to generate logical descriptions of the founders (particularly of the positive ones). An example of quantified decision tree is shown in Figure 1. It illustrates a process of dividing and extending positive and negative examples of `subset(A,B)` representing the subset relationship (`A` is a subset of `B`, using lists as sets). `mem(C,B)` says `C` is a member of `Prolog` list `B`. We will extract a runnable program from the tree in Section 3.

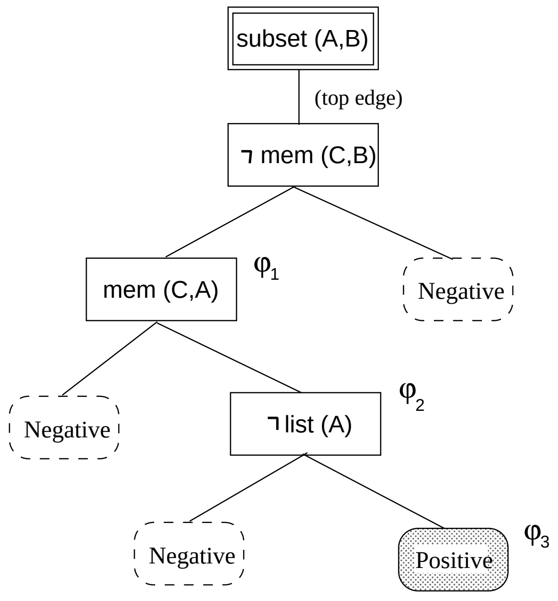

Figure 1: An example of a quantified decision tree

We now describe an algorithm for constructing quantified decision trees. Suppose positive and negative examples of a predicate $p(z)$ are given as a set of tuples T_0 in such a way that a tuple $t \in T_0$ labeled + (resp. -) represents a positive example (resp. a negative example) of $p(z)$. A quantified decision tree QDT for $p(z)$ is constructed from T_0 as follows. In the sequel, T^+ (resp. T^-) denotes the set of tuples labeled + (resp. -) in a set of tuples T. We assume a list S of literals usable for tuple division is given together with their positive and negative ground instances as background knowledge BK. We also assume some stopping condition is specified beforehand to avoid wasting time.

QDT construction algorithm

Initialization:

Create a root node and label it with a most general atom $p(z)$;

For every tuple t in T_0, define itself as its founder;

Create an outgoing edge (top edge) from the root as the current edge;

Attach T_0 to the current edge;

$T = T_0$;

Recursion on edge:

Let T be a set of tuples attached to the current edge;

Put $T = T^+ \cup T^-$;

If $T^+ = \emptyset$, create a leaf labeled "Negative" and return;

If $T^- = \emptyset$, create a leaf labeled "Positive" and return;

If neither T^+ nor T^- is empty, and the stopping condition is satisfied, create a leaf node labeled

"Mixed" and return; Otherwise

Literal selection:

By using BK, and based on some criterion for literal selection[2], select a best literal $L[y]$ from S where y is a set of new variables. We call y *new variables*.

Node creation:

Create a node at the end of the current edge, and label it with $L[y]$.

Tuple division and extension:

Divide T by $\exists y\ L[y]$ into T^l and T^r;

where T^l (resp. T^r) is a set of tuples satisfying (resp. failing to satisfy) $\exists y\ L[y]$;

Extend tuples in T^l with the value of y taking from BK while inheriting their founders;

Tuple deletion:

If tuples $t \in T^l$ and $t' \in T^r$ have a common founder, keep t and delete t';

Edge creation:

Create an outgoing left edge and attach T^l;

Create an outgoing right edge and attach T^r;

Recurse on both edges;

At **Tuple deletion** step, some tuples are deleted. This may look strange but necessary for the reason below. When new variables y are introduced at a node N by a literal $L[y]$, they are assumed implicitly existentially quantified, and we associate with the left child node a *tuple descriptor*, a formula describing tuples, of the form $\exists x, y\ (\phi[x] \wedge L[x, y])$ where x are new variables introduced until N. It reflects the fact that tuples attached to the edge to the left child give variable bindings to x and those bindings are extended to y. The tuple descriptor of the right child node on the other hand will be of the form $\forall x, y\ (\phi[x] \rightarrow \neg L[x, y])$ which says that no tuple can give variable bindings to x that can be extended to y so that $L[x, y]$ is satisfied.

Suppose there are two extended tuples, one being t satisfying $L[x, y]$ and the other t' having the same founder as t but no extension at N. In such case, since the descriptor associated with the right child node is refuted by the very existence of t, and will never be a correct description of the founder of t', there is no need to keep t'.

[2]We use a criterion used in ID3 [4].

3 Program extraction

Let QDT be a quantified decision tree constructed for $p(z)$. We extract a logic program from it. First of all, we label binary edges with literals, and then by gathering those literals on a path from the root to a leaf labeled "Positive," we synthesize the clause body describing positive examples of the top node (head atom in the clause).

Program extraction algorithm

Labeling edges:

For every internal node N in QDT, do as follows.

Let $L[y]$ be a selected literal used to make child nodes at N. Label the edge to the left child (resp. right child) with $L[y]$ (resp. $\neg L[y]$) and call it a *left literal* (resp. *right literal*).

Tuple descriptor:

Let N_0 be the root node of the binary subtree of QDT. Associate `true` with N_0 as a *tuple descriptor.* For every non-leaf node N below N_0, do as follows. Let $l_1, \ldots, l_m$ be left literals labeling the edges from N_0 down to N, and $L[y]$ and $\neg L[y]$ respectively be the left literal and the right literal labeling outgoing edges from N, where y is a set of new variables introduced at N. Let x be a set of all new variables introduced from N_0 down to N. Put

$$\begin{aligned}\varphi_l &= \exists x, y\, (\bigwedge_{k=1}^{m} l_k \wedge L[y]) \\ \varphi_r &= \forall x, y\, (\bigwedge_{k=1}^{m} l_k \rightarrow \neg L[y])\end{aligned}$$

and associate φ_l (resp. φ_r) with the left child (resp. right child) as tuple descriptors.

Synthesis:

Let N_0 be as above. For every path π from N_0 to a leaf node N_P labeled "Positive," do as follows. Let φ_P be the tuple descriptor associated with N_P, $\varphi_{r_1}, \ldots, \varphi_{r_h}$ those associated with the nodes on π that are a right child node. Put

$$\Psi[z] \quad = \quad \varphi_P \wedge \varphi_{r_1} \wedge \ldots \wedge \varphi_{r_h}$$

Construct such $\Psi[z]$ for each "Positive" node and let them be $\Psi_1[z], \ldots, \Psi_K[z]$. Put

$$p(z) \quad \Leftrightarrow \quad \Psi_1[z] \wedge \ldots \wedge \Psi_K[z]$$

In the case of the `subset(A,B)` example, descriptors in Figure 1 become

$$\begin{aligned}\varphi_1 &= \exists \mathtt{C} \neg \mathtt{mem(C,B)} \\ \varphi_2 &= \forall \mathtt{C}(\neg \mathtt{mem(C,B)} \rightarrow \neg \mathtt{mem(C,A)}) \\ \varphi_3 &= \forall \mathtt{C}(\neg \mathtt{mem(C,B)} \rightarrow \neg\neg \mathtt{list(A)})\end{aligned}$$

We therefore get the following clause:

$$\mathtt{subset(A,B)} \quad \Leftrightarrow \quad \varphi_1 \wedge \varphi_2 \wedge \varphi_3$$

Simplification assuming there always exists an element that does not belong to a given list (φ_1) and `A` is a list (φ_3) yields

$$\mathtt{subset(A,B)} \quad \Leftrightarrow \quad \forall \mathtt{C}(\mathtt{mem(C,A)} \rightarrow \mathtt{mem(C,B)})$$

which exactly describes the subset relationship expressed by (`Prolog`) lists.

4 First Order Compiler

Since the clause obtained in Section 3 contains a universally quantified goal, and hence not executable directly. It is however an example of first order clauses [6], and compilable, automatically by the First Order Compiler [6], into a `Prolog` program with disequality constraints. The First Order Compiler is a program transformation system designed for the synthesis of runnable logic programs from first order clauses whose body can be an arbitrary first order formula. The compilation is based on unfolding/folding transformation applied to universal continuation forms representing continuation passing style computation of logic programs.

Running the First Order Compiler for the following first order clauses

```
subset(A,B):- all([X],(mem(X,A) -> mem(X,B))).
mem(X,[X|Y]).
mem(X,[H|Y]):-mem(X,Y).
```

gives,

```
subset(A,B):- closure_mem0(A,f0(B)).
closure_mem0(A,B):-
    (\+ A=[C|D] ; A=[C|D],cont(C,B)),
    (\+ A=[E|F] ; A=[E|F],closure_mem0(F,B)).
cont(A,f0(B)):- mem(A,B).
```

Here `\+ A=[C|D]` is a `Prolog` substitute for for $\forall$`C,D` (`A` $\neq$ `[C|D]`) which works correctly when `A` is ground. If we know `A` is a list, the above program reduces to a more familiar `subset` program:

```
subset(A,B):- closure_mem0(A,f0(B)).
closure_mem0(A,B):-
    ( A=[]
    ; A=[C|D],cont(C,B),closure_mem0(D,B) ).
cont(A,f0(B)):- mem(A,B).
```

5 Discussion

We have implemented a quantified decision tree generator on `Prolog`. The implementation is very naive, and in the early stage of development. The example in Figure 1 was generated from 412 positive examples and 4212 automatically generated negative examples of `subset(A,B)`. We have tried some other synthesizing examples such as `max(M,L)` predicate (an integer `M` is the maximum element in the list `L`) and `sort(L,M)` (`M` is the sorted list of `L`), in which tuple division (and extension) by existentially quantified negative literals were effective and lead to the success of program synthesis. It is apparent however that various improvements including the optimization of program extraction in Section 3 are necessary and the feasibility of this approach should be tested on further examples.

Acknowledgments We greatly thank Mr.Kusama and Mr.Miyagawa for the implementation and improvement of the first working QDT generator.

References

[1] Blockeel,H. and De Raedt,L., Top-down induction of first-order logical decision trees, *Artificial Intelligence*, 101, pp.285-297, 1998.

[2] Boström,H. and Idestam-Almquist,P., Induction of Logic Programs by Example-Guided Unfolding, *J. Logic Comput.*, 40(2-3), pp.159-183, 1999.

[3] Muggleton, S., Inverse entailment and PROGOL, *New Generation Computing* 13, pp.245-286, 1995.

[4] Quinlan,J.R., Induction of Decision Trees, *Machine Learning*, 1, pp.81-106, 1986.

[5] Quinlan,J.R., Learning Logical Definitions from Relations, *Machine Learning*, 5, pp.239-266, 1990.

[6] Sato,T., First Order Compiler: A Determinstic Logic Program Synthesis Algorithm, *J. of Symbolic Computation*, 8, pp.605-627, 1989.