Can Decision Procedures be Learnt Automatically?

Mateja Jamnik [1,2]

University of Cambridge Computer Laboratory
J.J. Thomson Avenue, Cambridge, CB3 0FD, England, UK
www.cl.cam.ac.uk/~mj201

Predrag Janičić [3]

Faculty of Mathematics, University of Belgrade, Studentski trg 16
11000 Belgrade, Serbia and Montenegro
www.matf.bg.ac.yu/~janicic

Abstract

In this paper we present an investigation into whether and how can decision procedures be learnt automatically. Our approach consists of two stages. First, a refined brute-force search procedure applies exhaustively a set of given elementary methods to try to solve a corpus of conjectures generated by a stochastic context-free grammar. The successful proof traces are saved. In the second stage, a learning algorithm (by Jamnik et al.) tries to extract a required supermethod (i.e., decision procedure) from the given traces. In the paper, this technique is applied to elementary methods that encode the operations of the Fourier-Motzkin's decision procedure for Presburger arithmetic on rational numbers. The results of our experiment are encouraging.

1 Introduction

Learning proof methods and programs is a challenging task. Jamnik and colleagues [7] devised a framework for proof planning [4] systems where new

[1] The first author was supported by the EPSRC Advanced Research Fellowship, and the second author was supported by EPSRC grant GR/R52954/01 and by the Serbian Ministry of Science research grant 1379.

[2] Email: mateja.jamnik@cl.cam.ac.uk

[3] Email: janicic@matf.bg.ac.yu

proof methods can be learnt automatically (the implementation of this framework is called LearnΩmatic [8]). In this approach, a proof planning system is used to construct examples of proofs that use similar reasoning patterns. These proofs consist of low level inference steps or proof methods that are available to the system initially. The goal is to learn a procedure which uses these methods in some structured and efficient way. In order to learn such a procedure, a series of example proofs is generated automatically. The traces of example proofs are then fed into the learning mechanism which learns the so-called *method outline*, which captures the pattern common to all of the example proofs. Finally, the representation of a learnt method outline is enriched into a fully fleshed proof method so that it can be used by a specific proof planning system of choice. Such a learnt proof method is then used in subsequent proof planning attempts for other conjectures.

In this paper, we discuss how the learning approach in LearnΩmatic (for background, see §2.1) can be extended and used for a wider range of domains and procedures. In particular, we apply LearnΩmatic to developing decision procedures (for background, see §2.2). This is a challenging task as the learnt method should be terminating, sound and complete. Learning decision procedures automatically would be beneficial for a reasoning system, especially for user defined theories or when for some theory a decision procedure is not available. So, our main motivation is a mechanisation of learning and discovery of new decision procedures (while learning existing decision procedures serves as an illustration of an important step towards the final goal). Learning new decision procedures automatically can reduce the time required for developing them, it can prevent human implementation flaws, and presents a generic approach (that is independent of the theory) to generating decision procedures. We propose the programme and demonstrate how it can yield one specific procedure — Fourier-Motzkin's decision procedure [12] (the proposed framework can, of course, be used for other proof methods as well).

While our larger aim is to discover new procedures, we start by learning an existing procedure. This is a difficult task, since even if the idea of the required procedure is know and all the building blocks are available, it is still very challenging to combine them correctly into the require decision procedure. Our framework does not provide full automation (or guarantees formal properties, such as termination), however, it can be used as a very useful mechanised assistant. The user needs to provide the necessary building blocks and also some guidance to refine the brute force search according to the specific theory, in order to construct examples for automatic learning which generates the decision procedure.

In the research presented in this paper, we used the system LearnΩmatic [8], while all other discussed/used algorithms and modules were newly developed (and serve as an extension to LearnΩmatic).

Our programme (which also reflects the structure of this paper) consists of the following steps (we illustrate our approach with the example of linear arithmetic and the Fourier-Motzkin's procedure):

- the methods that can make up a decision procedure are provided (§3);
- the examples of proofs using the given methods are constructed (§4); this requires:
  - a number of conjectures is generated randomly (4.1);
  - implementing a simple PROLOG deduction system (which essentially carries out a brute force search) that applies the given methods (4.2);
  - grouping and ordering of methods to direct the brute force search and to prevent non-termination in the process of generating proof examples (4.3);
  - all example proofs are divided into groups according to a number of variables; from each group the most illustrative proofs are taken; all these selected proofs make the learning set (4.4).
- the selected example proofs are input into the learning mechanism which learns a procedure that captures the pattern of reasoning employed in all of the example proofs (§5);
- on the basis of the learnt pattern, a PROLOG mechanism automatically generates a corresponding supermethod (also in PROLOG), which is our required decision procedure (§6);
- the learnt procedure is tested on the original set of examples (§7).

We finish the paper with a brief discussion of related work in §8, and conclusions and future directions in §9.

2 Background

2.1 Automatic learning

Jamnik et al [7] devised a framework within which a proof planning [4] system can learn frequently occurring patterns of reasoning automatically from a number of typical examples, and then use them in proving new theorems [9]. The availability of such patterns, captured as proof methods in a proof planning system, reduces search and proof length. Jamnik et al implemented this learning framework for the proof planner ΩMEGA [2] – they call the system LEARNΩMATIC. The entire process of learning and using new proof methods in LEARNΩMATIC consists of the following steps:

(i) The user chooses informative examples and gives them to ΩMEGA to be automatically proved. Traces of these proofs are stored.

(ii) Proof traces of typical examples are given to the learning mechanism which automatically learns so-called *method outlines*.

(iii) Method outlines are automatically enriched by adding to them additional information and performing search for information that cannot be reconstructed in order to get fully fleshed proof methods that ΩMEGA can use in proofs of new theorems.

The methods LEARNΩMATIC aims to learn are complex and are beyond the complexity that can typically be tackled in the field of machine learning. Therefore, LEARNΩMATIC learns *method outlines*, which are expressed in the following language L, where P is a set of known identifiers of primitive methods used in a method that is being learnt:

- for any $p \in P$, let $p \in L$,
- for any $l_1, l_2 \in L$, let $[l_1, l_2] \in L$,
- for any $l_1, l_2 \in L$, let $[l_1|l_2] \in L$,
- for any $l \in L$, let $l^* \in L$,
- for any $l \in L$ and $n \in \mathbf{N}$, let $l^n \in L$,
- for any $list$ such that all $l_i \in list$ are also $l_i \in L$, let $T(list) \in L$.

"[" and "]" are auxiliary symbols used to separate subexpressions, "," denotes a *sequence*, "|" denotes a *disjunction*, "$*$" denotes a *repetition* of a subexpression any number of times (including 0), n a fixed number of times, and T is a constructor for a branching point ($list$ is a list of branches), i.e., for proofs which are not sequences but branch into a tree. For more information on the expressiveness of this language, the reader is referred to [9].

Our learning technique considers some typically small number of positive examples which are represented in terms of sequences of identifiers for primitive methods, and generalises them so that the learnt pattern is in language L. The pattern is of *smallest size* with respect to a defined heuristic measure of *size* [9], which essentially counts the number of primitives in an expression. The pattern is also *most specific* (or equivalently, least general) with respect to the definition of specificity *spec*. *spec* is measured in terms of the number of nestings for each part of the generalisation [9]. Again, this is a heuristic measure.

The algorithm is based on the generalisation of the simultaneous compression of well-chosen examples. Here is just an abstract description of the learning algorithm, but the detailed steps with examples of how they are applied can be found in [9]:

(i) Split every example trace into sublists of all possible lengths.

(ii) If there is any branching in the examples, then recursively repeat this algorithm on every element of the list of branches.

(iii) For each sublist in each example find consecutive repetitions, i.e., patterns, and compress them using exponent representation.

(iv) Find compressed patterns that match in all examples.

(v) If there are no matches in the previous step, then generalise the examples

by joining them disjunctively.

(vi) For every match, generalise different exponents to a Kleene star, and the same exponents to a constant.

(vii) For every matching pattern in all examples, repeat the algorithm on both sides of the pattern.

(viii) Choose the generalisations with the smallest size and largest specificity.

The learning algorithm is implemented in SML of NJ v.110. Its inputs are the sequences of methods extracted from proofs. Its output are method outlines.

2.2 Decision procedures

A theory $\mathcal{T}$ is *decidable* if there is an algorithm (which we call a *decision procedure*) such that for an input $\mathcal{T}$-sentence F, it returns *true* if and only if F is valid in $\mathcal{T}$ (i.e., $\mathcal{T} \models F$) and returns *false* otherwise. The role of decision procedures is often very important in theorem proving (e.g., see [10]). Decision procedures can reduce the search space of heuristic components of a prover and increase its abilities. Decision procedures can usually be much more efficient than some other proving strategies (e.g., induction). There are many decision procedures in standard use, including decision procedures for fragments of arithmetics, theories of lists, theory of equality etc. Due to its importance in hardware and software verification, decision procedures for fragments of arithmetic (like PRA — Presburger Rational Arithmetic) are of particular interest.

Instead of using basic inference rules, decision procedures are usually built from some higher-level building blocks. We start with methods in the spirit of Bundy's proof plans for normalisation [5].

We look at the ideas from Fourier-Motzkin's decision procedure [12] (which is essentially the same as the well known implementation of Hodes' decision procedure for Presburger arithmetic [6]). Fourier-Motzkin's algorithm is a decision procedure for rational numbers, but it is also often used (because of its better efficiency) as sound (but incomplete) procedure for the universal fragment of PIA – Presburger Integer Arithmetic (see, for instance, [3]).

3 Building blocks

We use a simple stand-alone PROLOG implementation of a deduction system based on the proof–planning paradigm, but it is simplified as it does not require preconditions and postconditions of methods.

Decision procedures can be implemented as compact, optimised procedures or they can be built from separate methods (some of which can be general-purpose methods, i.e., methods used also within other procedures). The lat-

ter approach often leads to additional overhead processing and is thus less efficient. However, it is much more flexible and gives easily understandable algorithms, and hence we use it in our programme.

We use the following sorts of normalisation methods (in the spirit of Bundy's proof plans for normalisations [5]):

Remove is a normalisation method used to eliminate a certain function symbol, predicate symbol or a quantifier from a formula. For instance, we can eliminate a connection $\Rightarrow$ by exhaustive application of the following rewrite rule: $f_1 \Rightarrow f_2 \longrightarrow \neg f_1 \vee f_2$.

Stratify is a normalisation method used to stratify a class of formulae into two (or more) syntactical layers containing just *some* specific predicate symbols, function symbols or connectives. For instance, *stratify* puts a formula into prenex normal form, moves negations inside disjunctions and conjunctions, moves conjunctions inside disjunctions etc.

Thin is a normalisation method that exhaustively applies thinning rewrite rules, such as elimination of multiple negations: $\neg\neg f \longrightarrow f$ or elimination of multiple unary minus symbols: $--t \longrightarrow t$.

Reduce is a method that reduces the number of occurrences (to at most one) of a certain function symbol, predicate symbol or a connective in a formula. For instance, it reduces the number of symbols $\top$ and $\bot$ in a formula being proved.

Left Association is one of the normalisation methods for reorganisation within a class. If a syntactical class contains only one function symbol and if that function symbol is both binary and associative, then members of this class can be put into left associative form. For instance, we can use this method for left association of addition and multiplication (given the needed rewrite rules).

Poly-form is a method which we will use for putting a formula into polynomial normal form. It uses rewrite rules such as: $i_1 \cdot i_2 \longrightarrow i_3$ where i_1, i_2, i_3 represent numbers and $i_1 \cdot i_2 = i_3$.

Reorder is one of the methods for reorganisation within one syntactical class. If a class contains only one function and if that function is commutative and associative, this method is used to reorder arguments within a term (which is supposed to be in left associative form). We can use it to reorder arguments in a term which is in polynomial normal form or in a formula in disjunctive normal form. This transformation requires an ordering on variables as an additional device.

Collect is a method which we will use to reduce multiple occurrences of some variable in a term.

Isolate is a method which we use to isolate a specific variable in an atomic formula.

The methods described above are general ones. Clearly, some theories may

require more specific methods.[4] However, even if all the necessary methods (general or theory-specific) are available, it may still be very challenging to combine them correctly into a required decision procedure.

4 Generating solved examples

We generated a set of solved examples in several stages: we generated a corpus, grouped and ordered the methods, ran brute force search for proofs and chose solved examples.

4.1 Generating corpus

We generated 1000 Presburger arithmetic conjectures by using the stochastic context-free grammar[5] given in Table 1. The probabilities used were chosen *ad-hoc* (a similar stochastic grammar was used in [11]). We believe that choosing different probabilities would give similar final results to the ones we got in this study. For simplicity, we generated only quantifier-free formulae,[6] and then took their universal closure.

4.2 Search for proofs

We implemented (in PROLOG) a simple mechanism for brute-force search for proofs of the given conjectures. The mechanism works as follows:

- if the current formula is equal to $\top$ or $\bot$, then stop the search;
- if the current list of applied methods exceeds the given limit, then stop the search;
- try to apply one of the available methods to the current formula; if the method changes the current formula, add that method to the list of applied methods and try to prove the obtained (now new current) formula.

If a current formula is transformed to $\top$ or $\bot$, we consider it solved and we call a sequence of applied methods *a proof trace*. We put the limit (100) for the number of applied methods in order to prevent infinite loops in this search. Some of the generated formulae were huge (one of them had 409 functions

[4] For example, in order to learn the Fourier-Motzkin's procedure, we need a method which performs *cross-multiply and add* step [12] (see also §4.3).

[5] A stochastic context–free grammar is a context–free grammar with a stochastic component which attaches a probability to each of the production rules and controls its use.

[6] Note that closed formulae without redundant quantifiers cannot be generated by a context-free grammar. However, this restriction is not critical. Namely, most quantifier elimination procedures (including the Fourier-Motzkin's procedure) eliminate universal quantifiers by reducing them to existential quantifiers. So, the learning process would be the same if we considered full Presburger arithmetic. Moreover, the learnt procedure (presented in §5) is a decision procedure for full Presburger arithmetic.

#	*Rule*	*Probability*
1.	$\langle \text{formula} \rangle := \langle \text{atomic formula} \rangle$	0.5
2.	$\langle \text{formula} \rangle := (\neg\ \langle \text{formula} \rangle)$	0.125
3.	$\langle \text{formula} \rangle := (\langle \text{formula} \rangle \vee \langle \text{formula} \rangle)$	0.125
4.	$\langle \text{formula} \rangle := (\langle \text{formula} \rangle \wedge \langle \text{formula} \rangle)$	0.125
5.	$\langle \text{formula} \rangle := (\langle \text{formula} \rangle \Rightarrow \langle \text{formula} \rangle)$	0.125
6.	$\langle \text{atomic formula} \rangle := (\langle \text{term} \rangle = \langle \text{term} \rangle)$	0.20
7.	$\langle \text{atomic formula} \rangle := (\langle \text{term} \rangle < \langle \text{term} \rangle)$	0.20
8.	$\langle \text{atomic formula} \rangle := (\langle \text{term} \rangle \leq \langle \text{term} \rangle)$	0.20
9.	$\langle \text{atomic formula} \rangle := (\langle \text{term} \rangle > \langle \text{term} \rangle)$	0.20
10.	$\langle \text{atomic formula} \rangle := (\langle \text{term} \rangle \geq \langle \text{term} \rangle)$	0.20
11.	$\langle \text{term} \rangle := (\langle \text{term} \rangle + \langle \text{term} \rangle)$	0.20
12.	$\langle \text{term} \rangle := 1$	0.20
13.	$\langle \text{term} \rangle := 0$	0.20
14.	$\langle \text{term} \rangle := \mathit{var}$	0.40
15.	$\langle \text{var} \rangle := x$	0.30
16.	$\langle \text{var} \rangle := y$	0.25
17.	$\langle \text{var} \rangle := z$	0.20
18.	$\langle \text{var} \rangle := u$	0.15
19.	$\langle \text{var} \rangle := v$	0.10

Table 1
A stochastic grammar for the quantifier-free fragment of Presburger arithmetic.

symbols, predicate symbols and connectives) so we also put a time limit for solving each conjecture. We used the time limit of 1 minute.[7]

4.3 Grouping methods and ordering of methods

On the basis of the generic normalisation methods discussed in §3, we implemented (in PROLOG) a set of arithmetic-specific methods. We also added the method for elimination of an existentially quantified (and isolated) variable based on Fourier-Motzkin's *cross-multiply and add* step [12]. For the sake of simplicity, we grouped some of these methods (in a natural, expected way), yielding the following set of 9 methods (some of them compound):

`M1`: remove $\Rightarrow$

[7] All modules were implemented in SWI Prolog; experiments were ran on a 64Mb PC 466Mhz. All source files are available upon request from the authors.

M2: remove $\neq, >, <, \geq$

M3: adjust the innermost quantifier (transforms $\forall x F$ to $\neg\exists x \neg F$)

M4: stratify $\neg$s beneath $\vee$s and $\wedge$s; thin $\neg$, remove $\neg$

M5: delete the innermost redundant quantifier (*cross-multiply and add* step)

M6: isolate the innermost variable (provided it is isolated in each atomic formula)

M7: stratify $\cdot$ beneath $+$, left-assoc $\cdot$, left-assoc $+$, poly-form

M8: stratify $\wedge$s beneath $\vee$s and eliminate the innermost variable

M9: reduce $\top$ and $\bot$

Despite having only 9 methods after grouping, a simple depth first search over them does not always produce proofs, because 9 methods still give a large search space [8] and, more importantly, some rules cancel each other out, which can lead to non-termination. Namely, most of the available methods consist of sets of rewrite rules. Even though each set of these sets of rewrite rules is terminating (but not always confluent), the union of sets is not necessarily terminating. Therefore, our set of methods is not terminating. Hence, in order to simplify and direct search, we also had to change the ordering of methods.

The two strategies just described, i.e., grouping and ordering, involve some human knowledge based on experiments in this context, and present a control information for search for proofs.

Methods are tried on given goals in the following order: M1, M2, M3, M4, M5, M6, M7, M8, M9. This ordering is *ad-hoc* and in our experiments we tried several orderings. We chose this as the most appropriate one. Notice that the ordering and grouping phase is not expected to provide the termination argument for the learnt procedure. It can be viewed as a heuristic which directs and improves the brute force search. Moreover, ordering and grouping can be helpful when considering the properties (such as termination and completeness) of the generated procedure (see §5).

4.4 Running brute force search and choosing examples

We ran the described search engine on the set of 1000 generated conjectures/ examples. 76.8% of conjectures were solved (proved or disproved) by this engine; results are given in Table 2. Table 2 also shows how the percentage of solved examples decreases as the number of variables increases. This is reasonable as the search space is rather big and the brute-force search is practically lost on very complex conjectures.

Having 768 solved examples, we needed to choose the subset of examples which would be used in the learning process (well-chosen examples are essential

[8] The situation is even worse if we consider low level inference rules, rather than higher level methods (since the proofs would be much longer, and the search space would be much larger).

# of variables	0	1	2	3	4	5	total
total	121	340	249	118	77	95	1000
solved	121	301	189	77	45	35	768
% solved	100	88.5	75.9	65.2	58.4	36.8	76.8
longest trace	5	10	15	18	23	30	N/A
# of examples with longest traces	6	8	10	4	5	2	35

Table 2
Results of the brute force method

for this phase of the programme). Good examples are demonstrative examples, i.e., the ones that involve as many methods as possible that should be in the decision procedure that we are learning. But these methods should be used in a concise way in good examples. The search for a proof (given our set and ordering of methods) stops as soon it reaches $\top$ or $\bot$. Thus, the available proofs are the shortest ones that the brute force engine can find. Amongst such proofs of different conjectures, we select as the most illustrative and descriptive proofs the longest ones. Namely, in some cases some methods (that form some parts of the procedure we are learning) leave certain formulae under consideration unchanged, but in other cases they transform (rewrite) them. So, such methods must be considered in order for the system to learn a (general) decision procedure. To learn such pieces of our sought procedure it was sensible to choose examples that use as many of the relevant methods as possible (i.e., examples that are the most difficult and demanding, and not trivial or easy ones). In other words, in a sense we choose the longest amongst the shortest proofs.

Since the number of variables has a critical role in proving Presburger arithmetic conjectures (the same holds for almost all theories), we separated all solved examples into groups according to the number of variables. We considered formulae with 0, 1, 2, 3, 4 and 5 variables. From each group we selected the longest proof traces (see Table 2).

Within the groups of formulae with 0, 1, and 2 variables all conjectures with the longest proof traces had the same traces (respectively):

```
[M1, M2, M4, M7, M9]
[M1, M2, M3, M4, M6, M8, M5, M4, M7, M9]
[M1, M2, M3, M4, M6, M8, M5, M3, M4, M6, M8, M5, M4, M7, M9]
```

Within the groups of formulae with 3 and 4 variables there were 4 and 5 conjectures with the longest proof traces, but these traces were not equal (within each respective group). Since it is not clear which amongst these are the most descriptive ones, we did not use them for learning.[9] Within the

[9] Namely, considering a possibly very complex procedure, it is not likely that within 1000

group of formulae with 5 variables there were 2 conjectures with the (same) longest proof trace. Finally, we took the longest traces for formulae with 0, 1 and 2 variables and put them into the learning mechanism.

5 Learning and generating supermethods

From the given sequences, the learning mechanism (described in §2.1) learnt the following general pattern:[10]

$$[M1, M2, [M3, M4, M6, M8, M5]^*, M4, M7, M9].$$

We notice that in each run of the loop ($[M3, M4, M6, M8, M5]^*$), one quantifier is eliminated. Since their number is finite in any conjecture, this process eventually terminates. Provided that all the used primitive methods are sound, the generated supermethod is also sound. Provided the methods are complete, then each conjecture is transformed by the above supermethod to $\bot$ or $\top$, and hence, the learnt procedure is a decision procedure for PRA. Although our proposed programme does not provide a guarantee about the properties of a learnt procedure (such as termination, soundness and completeness), often these properties can be easily proved (as we can see in the above informal discussion).

6 Automatic programming for learnt methods

We implemented (in PROLOG) a system for automatic generation of PROLOG predicates on the basis of sequences provided from the learning mechanism. The system supports all constructions that the LEARNΩMATIC system can make (see §2.1), and can generate corresponding PROLOG code. Given the sequence $[M1, M2, [M3, M4, M6, M8, M5]^*, M4, M7, M9]$, our system generated the following PROLOG code (which we finally applied to the original set of conjectures):

```
pa(Fa,FF):-
method('M1',Fa,Fb),
method('M2',Fb,Fc),
pb(Fc,Fd),
method('M4',Fd,Fe),
method('M7',Fe,Ff),
method('M9',Ff,FF).

pb(Fa,FF) :-
```

formulae we will have conjectures with 3, 4, 5,... variables whose proofs contain all the needed steps of the procedure in all iterations. Larger corpus would perhaps contain such conjectures (but then we may want to consider more variables, so the problem remains).

[10] As expected, it turns out that if examples with 5 variables were used for learning as well, then this learnt pattern would still be the same.

```
method('M3',Fa,Fb),
method('M4',Fb,Fc),
method('M6',Fc,Fd),
method('M8',Fd,Fe),
method('M5',Fe,Ff),
pb(Ff,FF),!.
pb(F,F).
```

7 Evaluation

Given the learnt method and the generated PROLOG program, we ran it on the original set of 1000 generated conjectures. While the brute force method solved 768 conjectures (within the given time limit), the learnt decision procedure solved 991 conjectures (see Table 3). Nine unsolved examples had hundreds of symbols and the method had not failed to solve them, but exceeded the time limit. For each conjecture solved by the brute force search, we measured the speed-up when using the newly generated procedure (see Table 3). The overall speed-up average was 1.0619. However, the main gain from the learnt procedure is in 223 conjectures that were not solved at all by the brute force method. We can see in Table 3 that the speed-up increases as the number of variables increases. The speed-up for 5-variable case would probably be higher if we used a higher time limit.

# of variables	0	1	2	3	4	5	total
total	121	340	249	118	77	95	1000
solved	121	340	249	118	77	86	991
% solved	100	100	100	100	100	90.5	99.1
speed-up	1	1.0001	1.0287	1.0990	1.4394	1.4181	1.0619

Table 3
Results of the learnt method

8 Related work

The work presented in this paper uses the learning mechanism of LEARNΩMATIC, which is related to the least general generalisation, and to some more recent work on learning regular expressions, grammar inference and sequence learning [13]. For details, see [9].

Our work is related to ideas from [5]. In Bundy's programme a decision procedure should be synthesised given all needed rewrite rules and several general patterns for normalising formulae. Considering automatic derivation

of decision procedures our work is also related to work presented in [1] which is aimed at deriving decision procedures using superposition.

9 Conclusions and future work

Our conclusion is that learning decision procedures is not an easy task (even when all the needed primitive methods are given), but it is possible. It is difficult to have the process of learning a complex decision procedure fully automated, so at some stages human interaction and human help is needed. We presented a methodology consisting of a number of steps, techniques and ideas (including a mechanism for generating a corpus of conjectures, a controlled brute force search, strategies for choosing examples, learning mechanism, and the system for automatic programming based on the learnt sequences). Automation in this field is important as it can prevent human flaws in analysing decision procedures or in implementing them. We believe that this methodology (and learning decision procedures in general) can be useful, especially for new or user defined theories. Here are some of the main lessons we learnt during the development of the proposed programme:

- Despite the fact that the implementation of decision procedures based on autonomous, independent methods is less efficient, we find that this approach is flexible and suitable for both analysing and synthesising decision procedures.
- Given a set of methods sufficient to solve any conjecture of a given theory, it is still not a trivial task to build a decision procedure for that theory. The brute force search can solve a number of conjectures, but it is difficult to make a brute force search complete, efficient and terminating (even when all the building blocks are terminating).
- Even if the idea of the required procedure is known and all the necessary building blocks are available, it may still be a non-trivial task to correctly implement the procedure. Automatic assistance in this can be very important.
- In order to make a brute force search more efficient, it is useful to provide some sort of control information. We used grouping and ordering of methods (where it was sensible to do so). This task requires human assistance.
- Having a number of solved examples, it is essential to make a good selection of examples to be used in the learning process. Our strategy was the following: we selected the longest proofs among the shortest proofs found by the brute force search. The rationale is that the most demanding conjectures are the most illustrative ones for learning.
- Provided that we have good examples and a choice of good methods, the learning mechanism can learn a decision procedure from just a few example

proofs.

- A system can be made which for a given learnt proof sequence generates a corresponding implementation.
- The learnt method outperforms the brute force search both in the number of conjectures solved and in the CPU time spent.
- We believe that the methodology presented in this study is very well suited to the proof planning paradigm (or its simplified version, as described here), and can be applied to other environments as well.

It is difficult to provide a characterisation of theories for which the proposed approach is successful, since some very deep theory-specific knowledge may be required. However, we can give a characterisation of decision procedures which cannot be learnt: the proposed framework cannot learn procedures which cannot be expressed with the language used in LEARNΩMATIC. All other procedures can potentially be learnt. At the moment, LEARNΩMATIC covers a wide range of languages, while further extensions are under consideration. Learning procedures expressed in another language would require that we replace in our framework LEARNΩMATIC's learning mechanism with another one that uses the desired language, but the other modules of our framework (e.g., generating examples, automatic generation of code from the learnt pattern) can remain unchanged. We also plan to extend the learning approach and the realm of covered languages so that the mechanism could learn recursive methods, which would enable automatic learning of a new range of decision procedures.

Another limitation of our proposed programme is that it may require non-trivial human assistance (e.g., in ordering and grouping). We plan to further develop our methodology and to try to automate (at least to some extent) the steps which now need human interaction.

A comparison between a direct implementation of the decision procedure and a learnt decision procedure would be interesting for further work. But this is out of the scope of the present paper, as we are interested in a larger picture of discovering new decision procedures, rather than in efficient implementations of the existing ones. Mechanised learning of existing decision procedures is an important step towards mechanised learning and discovery of decision procedures. In this sense, the work presented in this paper is an encouraging preliminary step towards discovery. Our hope is that such a framework will be used as a useful assistant in such a process, and moreover, it will lead to automatic discovery of new decision procedures.

References

[1] A. Armando, S. Ranise, and M. Rusinowitch. Uniform Derivation of Decision Procedures by Superposition. CSL 15, LNCS 2142. Springer, 2001.

[2] C. Benzmüller et al. ΩMEGA: Towards a mathematical assistant. CADE 14, LNCS 1249, Springer, 1997.

[3] R. S. Boyer and J S. Moore. Integrating Decision Procedures into Heuristic Theorem Provers: A Case Study of Linear Arithmetic. *Machine Intelligence 11*, 1988.

[4] A. Bundy. The use of explicit plans to guide inductive proofs. CADE 9, LNCS 310, Springer.

[5] A. Bundy. The use of proof plans for normalization. In *Essays in Honor of Woody Bledsoe*, Kluwer, 1991.

[6] L. Hodes. Solving problems by formula manipulation in logic and linear inequalities. IJCAI 2, William Kaufmann, 1971.

[7] M. Jamnik, M. Kerber, and M. Pollet. Automatic learning in proof planning. ECAI 15, 2002.

[8] M. Jamnik, M. Kerber, and M. Pollet. LEARNΩMATIC: System description. CADE 18, LNCS 2392, Springer, 2002.

[9] M. Jamnik, M. Kerber, M. Pollet, and C. Benzmüller. Automatic learning of proof methods in proof planning. Technical Report CSRP-02-5, School of Computer Science, University of Birmingham, 2002. Submitted to Journal of AI.

[10] Predrag Janičić and Alan Bundy. A general setting for the flexible combining and augmenting decision procedures. *Journal of Automated Reasoning*, 28(3), 2002.

[11] Predrag Janičić, Ian Green, and Alan Bundy. A comparison of decision procedures in Presburger arithmetic. LIRA '97, Univ. of Novi Sad, 1997.

[12] J.-L. Lassez and M.J. Maher. On Fourier's algorithm for linear arithmetic constraints. *Journal of Automated Reasoning*, 9(3), 1992.

[13] Sun, R., Giles, L., eds.: Sequence Learning: Paradigms, Algorithms, and Applications. LNAI 1828, Springer, 2000.