Inductive Rule Learning on the Knowledge Level

Ute Schmid and Emanuel Kitzelmann

Faculty Information Systems and Applied Computer Science
University of Bamberg, 96045 Bamberg, Germany
{ute.schmid, emanuel.kitzelmann}@uni-bamberg.de

Abstract

We present an approach to learning sets of recursive rules based on analytical inductive programming. We propose that our approach can be used within cognitive architectures to model regularity detection and generalization over experience. Induced rule sets represent the knowledge underlying systematic behavior in complex situations. Such rule sets can explain systematicity and productivity of behavior by applying a learned rule set in structural similar situations. The proposed analytical approach to rule induction provides a generalization mechanism which is based on the given experience. That is, this approach is complementary to evolutionary and other generate-and-test based approaches to rule learning where large spaces of hypothetical rule sets are searched for such a set of rules which covers the regularities in the examples. After presenting our approach we will give a variety of example applications from different problem solving domains and show that the same approach can be applied to rule acquisition for reasoning and natural language processing.

1 Introduction

The human ability to master complex demands is to a large extend based on the ability to exploit previous experiences. Based on our experience, we are able to predict characteristics or reactions of (natural or man-made, inanimate or animate) objects, we can reason about possible outcomes of actions, and we

can apply previously successful routines and strategies to new tasks and problems. In philosophy, psychology and artificial intelligence, researchers proposed that the core process to expand knowledge, that is, construct hypotheses, in such a way that we can transfer knowledge from previous experience to new situations is \textit{inductive} inference (Goodman, 1965; Klahr & Wallace, 1976; Holland, Holyoak, Nisbett, & Thagard, 1986).

From the perspective of machine learning, inductive inference is characterized as a mechanism for generalizing over observed regularities in examples. To allow for generalization, every learning algorithm must make some \textit{a priori} assumptions. This so called inductive bias gives the rational basis to allow transfer of the learned hypothesis to new situations (Mitchell, 1997). The \textit{restriction} or language bias characterizes the language in which induced hypotheses are represented. The \textit{preference} or search bias characterizes the mechanism for selecting hypothesis. Every machine learning algorithm, be it statistical, symbolical or neuronal, makes such (maybe implicit) assumptions. This basic proposition of machine learning certainly also holds for human induction. That is, human learning is very powerful but nevertheless learning is restricted by the way knowledge can be represented and by some mechanism for preferring some generalizations over others.

In this paper we focus on learning of complex structures and routines on the knowledge level. More precisely, we want to demonstrate that an approach to inductive programming can be applied to model the acquisition of generalized rules from example experience. We specifically are interested in rules which are productive in such a sense that they can be applied in situations of various complexity, for example, rules which can characterize the ability of humans to grasp the transitivity of some concepts such as \textit{ancestor}. Typically, we get acquainted with such a concept by some, necessarily small and finite examples, such as \textit{my father is my ancestor}, \textit{my grandfather is my ancestor}, and \textit{my grand-grand-father} is also my ancestor and are able to induce the infinite regularity of the concept which allows us to apply it to ancestor relations of various complexities. Such productive rules also govern the generation and application of regular action sequences. For example, if humans have learned how to solve Tower of Hanoi problems with three and four discs, at least some of them are able to generalize the underlying strategy for solving problems with an arbitrary number of discs (Kotovsky, Hayes, & Simon, 1985; Anderson & Douglass, 2001; Altmann & Trafton, 2002; Welsh & Huizinga, 2005).

Such learned concepts and strategic rules typically can be communicated, that is, they must be available as declarative structures on the knowledge level (Newell, 1982). For that reason, we are interested in inductive programming as a \textit{symbolic} approach to learning. While statistical and neuronal approaches present powerful mechanisms for inductive learning, the acquired classifiers typically can not or not easily be transformed into symbolic representations.
In the following we shortly introduce our inductive programming system *IGOR*. Then we present the general setting for incorporating *IGOR* in a cognitive architecture. Afterwards we will give various examples of cognitive rule acquisition in the domain of problem solving. We will shortly demonstrate that the same mechanisms can be applied to learning in the context of reasoning and language understanding and we will conclude with a discussion and further work to be done.

2 Analytical Inductive Programming with *IGOR*

We give a short introduction to inductive programming research and *IGOR* as our contribution to this field. More extensive descriptions of inductive programming and the *IGOR* system can be found in the references.

Inductive programming research addresses the problem of learning computer programs from incomplete specifications, typically samples of the desired input/output behavior and possibly additional constraints (Biermann, Guiho, & Kodratoff, 1984; Flener & Schmid, 2009). Induced programs are typically in a declarative – functional or logic – programming language. As a special application of machine learning (Mitchell, 1997), inductive programming creates program *hypotheses*, that is, generalized, typically recursive, programs. In contrast to classification learning, program hypotheses must cover all given examples correctly since for programs it is expected that a desired input/output relation holds for all possible inputs.

There are two general approaches to inductive programming: analytical and search-based methods. Search-based methods enumerate syntactically correct programs and test these programs against the given examples guided by some search strategy. For example, the inductive logic programming (ILP) system *FOIL* constructs sets of Horn clauses by sequential covering, that is by a first construct then test approach. This approach was also applied to learning recursive rule sets (Quinlan & Cameron-Jones, 1995). The most successful search-based system is *ADATE* (Olsson, 1995) which constructs ML programs using evolutionary principles. The system *MAGICHASKELLER* (Katayama, 2005) enumerates HASKELL programs with higher-order functions.

Clearly, it is not plausible to assume that a cognitive system learns rules by a generate-and-test approach. In contrast, in analytical methods learning is guided by the structure underlying the given examples. Beginning of research on inductive programming in the nineteen-seventies was concerned with such analytical strategies for learning *LISP* programs from small sets of positive
input/output examples (Biermann et al., 1984). The most influential of the early systems is Thesys (Summers, 1977). It realized a two step approach to synthesize programs: In a first step, input/output examples were rewritten into traces, in a second step recurrent patterns were searched-for in the traces and the found regularities were generalized to a recursive function.

For Thesys and all later analytical approaches it is enough to present a small set of only positive input/output examples. These examples must be the first representants of the underlying data-type of the input parameter. In contrast, in search-based approaches, an arbitrary set of positive examples can be presented. In addition negative examples can be used to eliminate unsuitable hypotheses.

IGOR2 (Kitzelmann, 2008) was developed as a successor to the classical Thesys system and its generalization IGOR1 (Kitzelmann & Schmid, 2006). To our knowledge, IGOR2 is currently the most powerful system for analytical inductive programming. Its scope of inducible programs and the time efficiency of the induction algorithm compares favorably with inductive logic programming and other approaches to inductive programming (Hofmann, Kitzelmann, & Schmid, 2008). The system is realized in the constructor term rewriting system Maude. Therefore, all constructors specified for the data types used in the given examples are available for program construction. Since IGOR2 is primarily designed as an assistant system for program induction, it relies on small sets of noise-free positive input/output examples and it cannot deal with uncertainty. Furthermore, the examples have to be the first inputs with respect to the complexity of the underlying data type. Given these restrictions, IGOR2 can guarantee that the induced program covers all examples correctly and provides a minimal generalization over them. Classification learning for noise-free examples such as PlayTennis (Mitchell, 1997) can be performed as a special case (Kitzelmann, 2008).

IGOR2 specifications consist of a set of examples as described above together with a specification of the input data type. Background knowledge for additional functions can (but needs not) be provided. IGOR2 can induce several dependent target functions (i.e., mutual recursion) in one run. Auxiliary functions are invented if needed. In general, a set of rules is constructed by generalization of the input data by introducing patterns and predicates to partition the given examples and synthesis of expressions computing the specified outputs. Partitioning and search for expressions is done systematically and completely which is tractable even for relative complex examples because construction of hypotheses is data-driven. IGOR2’s restriction bias is the set of all functional recursive programs where the outermost function must be either non-recursive or provided as background knowledge.

IGOR2’s built-in preference bias is to prefer fewer case distinctions, most spe-
specific patterns and fewer recursive calls. Thus, the initial hypothesis is a single rule per target function which is the least general generalization of the example equations. If a rule contains unbound variables on its right-hand side, successor hypotheses are computed using the following operations: (i) Partitioning of the inputs by replacing one pattern by a set of disjoint more specific patterns or by introducing a predicate to the right-hand side of the rule; (ii) replacing the right-hand side of a rule by a (recursive) call to a defined function where finding the argument of the function call is treated as a new induction problem, that is, an auxiliary function is invented; (iii) replacing sub-terms in the right-hand side of a rule which contain unbound variables by a call to new subprograms.

Instead of presenting the algorithmic details, in the following sections we will illustrate the IGOR approach with various examples.\(^2\)

### 3 IGOR as Cognitive Rule Acquisition Device

In the context of a cognitive architecture IGOR can be seen as a module which observes the content of the working memory and greedily tries to generalize over its content. In many cases, IGOR will not detect regularities. This can be because for the current cognitive task there exists no underlying regularity or because the current problem representation is structured in such a way that the regularities cannot be detected. In the second case, one can hope that over several problem solving trials in the domain eventually the problem is represented such that regularity detection becomes possible.

While we are aware that finding the “right” representation is a crucial problem (Kaplan & Simon, 1990), in this paper we focus on IGOR as a powerful and fast mechanism for constructing minimal generalizations for complex domains. In the domain of problem solving, we explored some strategies for rewriting action sequences into suitable representations for IGOR (Schmid & Wysotzki, 2000; Kitzelmann, 2003; Schmid, 2003).

If IGOR produces a generalization over examples, the found rules are available as symbolic representations in the working memory. This can be interpreted as some realization of the “aha”—experience – the sudden impression that one has understood a problem (Bühler, 1907; Ohlsson, 1984; Knoblich, Ohlsson, Haider, & Rhenius, 1999). The experience of having a sudden insight is well-known to every human problem solver. We all remember episodes where we suddenly “saw” the solution procedure for some problem and “knew” that this

---

\(^2\) The complete data sets and results can be found on www.cogsys.wiai.uni-bamberg.de/effalip/download.html.
is the right one. More systematic empirical evidence of this phenomenon is for example available for Tower of Hanoi where it could be shown that successful problem solvers can verbalize the components of the recursive solution strategy (Welsh & Huizinga, 2005).

There are some aspects of IGOR which might be considered as making this approach implausible as a cognitive mechanism. First, one might argue that humans notoriously have difficulties in understanding recursion (Kahney, 1989) and therefore that representing expertise by recursive rule sets is inadequate. We argue that such recursive rule sets are just a means to represent the competence of producing systematic and productive behavior in a given domain. Nobody would argue that neural nets are inadequate for representing human skills because humans are not able to multiply large matrices of real numbers on the fly. Furthermore, all approaches to classification learning which do not take into account recursion – be it symbolic approaches or sub-symbolic or statistical approaches – will always be bound by some fixed size of the input to produce a meaningful output. For example, such approaches might correctly learn an ancestor-relation up to the tenth grandfather but be unable to produce a useful output for the eleventh one; they would produce correct action sequences for solving Tower of Hanoi problems up to five discs but not for more discs, and so on.

Second, one might argue that such a brittle approach as analytical inductive programming does not account for human errors since, once the correct rule set is induced, the system will always generate correct solutions. Here we argue that IGOR models learning on the competence level. We do not dispute the fact that there are many factors which influence performance. There are many reasons why a person who has acquired a correct set of rules might produce errors when solving a given problem in a given context. Typical sources of error are working memory restrictions or motivational or emotional states which influence information processing (Bach, 2009; Simon, 1958).

Third, one might argue that a mechanism which can only induce correct rules if the first examples with respect to some underlying data type are presented and that this examples are error-free is not suitable for modeling learning in complex domains. Here we have a trade-off between analytical approaches where rule construction is guided by the examples and generate-and-test approaches. Generate-and-test approaches are robust with respect to variance in complexity of the given examples and with respect to erroneous examples. However, there is a price to pay: in such approaches, huge spaces of possible rule sets are generated and tested against the given examples. It is not plausible to assume that such kind of search processes take place in working memory. Furthermore, these approaches typically are much slower than IGOR2 which typically can induce a rule set in milliseconds (Hofmann et al., 2008). In general, there will be a trade-off between the amount of search (and
search space) necessary to produce generalized rule sets and the requirements which constrain the examples. Fortunately, in many domains, obtaining the $k$ first positive examples comes quite naturally. As we will show in the following section, in problem solving typically the examples are derived from the problem solving traces which can be ordered by the number of actions which were performed.

4 Rule Induction in Problem Solving

Often, in cognitive psychology, speed-up effects in problem solving are modelled as composition of primitive rules as a result of their co-occurrence during problem solving, e.g., knowledge compilation in ACT (Anderson & Lebière, 1998) or operator chunking in SOAR (Rosenbloom & Newell, 1986). Similarly, in AI planning macro learning was modelled as composition of primitive operators to more complex ones (Minton, 1985; Korf, 1985). We will not dispute that performance improvements in problem solving can often be explained by such mechanisms for a more efficient organization of the rule base. But, there is empirical evidence that humans are able to acquire general problem solving strategies from problem solving experiences, that is, that generalized strategies are learned from sample solutions. For example, after solving Tower of Hanoi problems, at least some people have acquired the recursive solution strategy (Anzai & Simon, 1979; Welsh & Huizinga, 2005). Typically, experts are found to have superior strategic knowledge in contrast to novices in a domain (Meyer, 1992).

There were some proposals to the learning of domain specific control knowledge in AI planning (Shell & Carbonell, 1989; Shavlik, 1990; Martín & Geffner, 2000). All these approaches proposed to learn cyclic/recursive control rules which reduce search. Learning recursive control rules, however, will eliminate search completely. With enough problem solving experience, some generalized strategy, represented by a set of rules (equivalent to a problem solving scheme) should be induced which allows a domain expert to solve this problem via application of his/her strategic knowledge. We already tried out this idea using IGOR1 (Schmid & Wysotzki, 2000). However, since IGOR1 was a two-step approach where examples had to be first rewritten into traces and afterwards recurrence detection was performed in these traces, this approach was restricted in its applicability. With IGOR2 we can reproduce the results of IGOR1 on the problems clearblock and rocket faster and without specific assumptions to preprocessing and furthermore can tackle more complex problem domains such as building a tower in the blocks-world domain.

The general idea of learning domain specific problem solving strategies is that first some small sample problems are solved by means of some planning or
problem solving algorithm and that then a set of generalized rules are learned from this sample experience. This set of rules represents the competence to solve arbitrary problems in this domain.

4.1 How to Clear a Block

We illustrate the idea of our approach with the simple clearblock problem (see Figure 1). How to infer the underlying recursive strategy was also researched in the context of deductive program synthesis (Manna & Waldinger, 1987). A problem consists of a set of blocks which are stacked in some arbitrary order. The problem solving goal is that one specific block – in our case A – should be cleared such that no block is standing above it. We use predicates clear(x), on(x, y), and ontable(x) to represent problem states and goals. The only available operator is puttable: A block x can be put on the table if it is clear (no block is standing on it) and if it is not already on the table but on another block. Application of puttable(x) has the effect that block x is on the table and the side-effect that block y gets cleared if on(x, y) held before operator application. The negative effect is that x is no longer on y after application of puttable.

**Problem domain:**
puttable(x)
PRE: clear(x), on(x, y)
EFFECT: ontable(x), clear(y), not on(x, y)

**Problem Descriptions:**
: init-1 clear(A), ontable(A)
: init-2 clear(A), on(A, B), ontable(B)
: init-3 on(B, A), clear(B), ontable(A)
: init-4 on(C, B), on(B, A), clear(C), ontable(A)
: goal clear(a)

**Problem Solving Traces/Input to Igor2**
```plaintext
fmod CLEARBLOCK is

*** data types, constructors
sorts Block Tower State .
op table : -> Tower [ctor] .
op __ : Block Tower -> Tower [ctor] .
op puttable : Block State -> State [ctor] .

*** target function declaration
op ClearBlock : Block Tower State -> State [metadata "induce"] .

*** variable declaration
vars A B C : Block .
var S : State .

*** examples
eq ClearBlock(A, A table, S) = S .
eq ClearBlock(A, A B table, S) = S .
eq ClearBlock(A, B A table, S) = puttable(B, S) .
eq ClearBlock(A, C B A table, S) = puttable(B, puttable(C, S)) .
endfm
```

**Induced Clearblock Strategy** (4 examples, 0.036 sec)
ClearBlock(A, (B T), S) = S if A == B
ClearBlock(A, (B T), S) = ClearBlock(A, T, puttable(B, S)) if A != B

Fig. 1. Initial experience with the clearblock problem with three blocks and solution
We use a PDDL-like\textsuperscript{3} notation for the problem domain and the problem descriptions. We defined four different problems of small size each with the same problem solving goal (\textit{clear}(A)) but with different initial states: The most simple problem is the case where A is already clear. This problem is presented in two variants – A is on the table and A is on another block – to allow the induction of a \textit{clearblock} rule for a block which is positioned in an arbitrary place in a stack. The third initial state is that A is covered by one block, the fourth that A is covered by two blocks. A planner might be presented with the problem domain – the \textit{puttable} operator – and problem descriptions given in Figure 1.

The resulting action sequences can be obtained by any PDDL planner (Ghallab, Nau, & Traverso, 2004) and rewritten to IGOR\textsuperscript{2} (i.e. MAUDE) syntax. When rewriting plans to MAUDE equations (see Figure 1) we give the goal, that is, the name of the block which is to be cleared, as first argument. The second argument represents the initial state, that is, the stack as list of blocks and \textit{table} as bottom block. The third argument is a situation variable (McCarthy, 1963; Manna & Waldinger, 1987; Schmid & Wysotzki, 2000) representing the current state. Thereby plans can be interpreted as nested function applications and plan execution can be performed on the content of the situation variable. The right-hand sides of the example equations correspond to the action sequences which were constructed by a planner, rewritten as nested terms with situation variable \textit{S} as second argument of the \textit{puttable} operator. Currently, the transformation of plans to examples for IGOR\textsuperscript{2} is done “by hand”. For a fully automated interface from planning to inductive programming, a set of rewrite rules must be defined.

Given the action sequences for clearing a block up to three blocks deep in a stack as initial experience, IGOR\textsuperscript{2} generalizes a simple tail recursive rule system which represents the competence to clear a block which is situated in arbitrary depth in a stack (see Figure 1). That is, from now on, it is no longer necessary to search for a suitable action sequence to reach the \textit{clearblock} goal. Instead, the generalized knowledge can be applied to produce the correct action sequence directly. Note, that IGOR\textsuperscript{2} automatically introduced the equal predicate to discern cases where A is on top of the stack from cases where A is situated farther below since these cases could not be discriminated by disjoint patterns on the left-hand sides of the rules.

4.2 Rocket Transport

A more complex problem domain is \textit{rocket} (Veloso & Carbonell, 1993). This domain was originally proposed to demonstrate the need of interleaving goals.

\textsuperscript{3} see \url{http://ls5-www.cs.tu-dortmund.de/~edelkamp/ipc-4/pddl.html}

9
Fig. 2. Learned rules for the rocket domain

The problem is to transport a number of objects from earth to moon where the rocket can only fly in one direction. That is, the problem cannot be solved by first solving the goal at(o1, moon) by loading it, moving it to the moon and then unloading it. Because with this strategy there is no possibility to transport further objects from earth to moon. The correct procedure is first to load all objects, then to fly to the moon and finally to unload the objects. IGOR2 learned this strategy from examples for zero to two objects (see Figure 2).

4.3 Building Towers

A most challenging problem domain which is still used as a benchmark for planning algorithms is blocks-world. A typical blocks-world problem is to build a tower of some blocks in some prespecified order. With evolutionary programming, an iterative solution procedure to this problem was found from 166 examples (Koza, 1992). The found strategy was to first put all blocks on the table and than build the tower. This strategy is clearly not efficient and cognitively not very plausible. If, for example, the goal is a tower on(A, B), on(B, C) and the current state is on(C, B), on(B,A), even a young child will first put C on the table and then directly put B on C and not put B on the table first. Another proposal to tackle this problem is to learn decision rules which at least in some situations can guide a planner to select the most suitable action (Martín & Geffner, 2000). With the learned rules, 95.5% of 1000 test problems were solved for 5-block problems and 72.2% of 500 test problems were solved for 20-block problems. The generated plans, however, are about two steps longer than the optimal plans. In Figure 3 we present the rules IGOR2 generated from only nine example solutions. This rule system will always produce the optimal action sequence.

To illustrate how examples were presented to IGOR2 we show one example in Figure 4. The goal is to construct a tower for some predefined ordering of blocks. To represent this ordering, blocks are represented constructively as

Tower domain

(additionally: 10 corresponding examples for Clear and IsTower predicate as background knowledge)

Tower(O, S) = S if IsTower(O, S)
Tower(O, S) =
    put(O, Sub1(O, S),
        Clear(O, Clear(Sub1(O, S),
            Tower(Sub1(O, S), S)))) if not(IsTower(O, S))
Sub1(x, O, S) = O .

Fig. 3. Induced optimal strategy for solving arbitrary tower problems
\begin{verbatim}
eq Tower(s s table,  
   ((s s s s table) (s table) table | ,  
   (s s s table) (s s table) table | , nil)) =  
put(s s table, s table,  
   put(s s s s table, table,  
       put(s s s s s table, table,  
           ((s s s s s s table) (s s table) table | ,  
           (s s table) (s s s table) table | , nil)))) .
\end{verbatim}

Fig. 4. One of the nine example equations for \textit{tower} “successors” to the table with respect to the goal state (\(|\) representing the empty tower). Therefore the top object of the to be constructed tower is given as first argument of the \textit{tower} function. If the top object is \(s s s s s s s s \text{table}\), the goal is to construct a tower with three blocks with \textit{s table} on the table, \(s s \text{table} \text{on} s \text{table} \text{on} s \text{table}\) and \(s s s \text{table} \text{on} s s \text{table}\). The second argument again is a situation variable which initially holds the initial state. In the example in Figure 4 \(s s \text{table}\) (we may call it block 2) shall be the top object and the initial state consists of two towers, namely block 4 on block 1 and block 3 on block 2. That is, the desired output is the plan to get the tower block 2 on block 1. Therefore blocks 1 and 2 have to be cleared, these are the both innermost puts, and finally block 2 has to be stacked on block 1 (block 1 lies on the table already), this is the out-most put.

In addition to the \textit{tower} example, IGOR2 was given an auxiliary function \textit{IsTower} as background knowledge. This predicate is true if the list of blocks presented to it are already in the desired order. Furthermore, we did not learn the \textit{Clear} function used in \textit{tower} but presented some examples as background knowledge.

\subsection{Car Park Problems}

In 2008, the biennial \textit{international planning competition (IPC)} for the first time included a special \textit{learning track\footnote{Homepage of the IPC: http://www.icaps-conference.org/index.php/Main/Competitions, homepage of the 2008 learning track: http://eecs.oregonstate.edu/ipc-learn/}}. The learning track contains two phases: A learning phase, where the planners solve example problems in the competition domains; and an evaluation phase, where they solve further problems in the same domains and where they may use domain-specific control-knowledge—e.g., domain-specific heuristics, macro operators, or policies—that they learned based on solving the problems in the learning phase.

One of the used domains was the \textit{parking domain}. Several cars are to be parked into \(N\) curb locations, where each curb location can carry up to two cars. If two cars are “stacked” into one location, only the back-most car can move. If
there are $N$ curbs, then there are $2 \times (N - 1)$ distinguished cars; two parking positions are always free. A problem consists of re-parking all cars from a particular initial configuration to a particular target configuration.

The domain is isomorphic to a variant of the blocksworld: Instead of building towers anywhere at the table, there are $N$ possible tower-locations and each tower can maximally contain two blocks. Furthermore, there are exactly $2 \times (N - 1)$ involved blocks.

We consider a generalized version of this car parking domain—namely a generalization to $M$ (instead of 2) cars that can be stacked into one curb location. If there are $N$ locations, then there are $M \times (N - 1)$ cars involved.

Coming up with an (optimal) solution strategy for this problem class is even harder than for tower building in the ordinary blocksworld, because there are more constrains regarding applicability of operators. A general strategy could be to fill one parking position after the other with the correct car (according to the goal configuration) where the parking positions are ordered from innermost to outermost in each curb location. (In the equivalent constraint blocksworld domain this would correspond to get the correct blocks position by position where positions in each tower are ordered from bottom to top.) However, filling one particular position is itself not a trivial task. Obviously, it comprises (i) clearing the correct car such that no other car is stacked behind it and (ii) clearing the particular parking position such that the curb location including this position is only filled up to the preceding position. These two sufficient preconditions for moving the correct car to the chosen position imply an even stronger precondition. Consider the particular task of moving car $A$ to the innermost position in some curb location. That means, the target curb location must be completely free, no car may already be parked in it. Yet since there are $N$ locations and $M \times (N - 1)$ cars and each location only can take $M$ cars, this implies that all other curb location are completely filled. Hence, our car $A$, in order to be movable, must be located at the outermost position of any curb location.
That is, one relevant subproblem of the complete parking problem class is to fill one particular curb location completely with a particular car at the outermost position. We have considered this particular subproblem. The function to be learned was \texttt{PutLast}. It takes five parameters: The selected car, the selected curb location, a list of further (possibly all remaining) cars that can be moved according to their order in the list, a number indicating the remaining number of free positions in the selected curb location, and a state. We assume, that the selected car to be moved to the out-most position is already cleared, hence movable. Figure 5 shows the four examples we provided to IGOR and the learned rules. The learned strategy is to take cars from the list of cars and move them into the selected curb location until only one free position remains. Then move the selected car into this outermost position.

4.5 Tower of Hanoi

Finally, the recursive solution to the Tower of Hanoi problem was generated by IGOR2 from three examples (see Figure 6).

For the discussed typical problem solving domains IGOR2 could infer the recursive generalizations very fast and from small example sets. The learned recursive rule systems represent the strategic knowledge to solve all problems of the respective domains with a minimal number of actions. As it was to be expected, Tower of Hanoi can be learned faster and with less examples than tower. This is contrary to what we know from human cognitive development: Tower of Hanoi is the by far more intellectually complex problem. While already small children can build towers of blocks in some order, they usually are

\begin{verbatim}
Input to IGOR2
eq Hanoi(0, Src, Aux, Dst, S) = move(0, Src, Dst, S).

eq Hanoi(s 0, Src, Aux, Dst, S) =
  move(0, Aux, Dst,
  move(s 0, Src, Dst,
  move(0, Src, Aux, S))).

eq Hanoi(s s 0, Src, Aux, Dst, S) =
  move(0, Src, Dst,
  move(s 0, Aux, Dst,
  move(s 0, Src, Dst,
  move(0, Dst, Aux,
  move(s 0, Src, Aux, S))))).

Induced Tower of Hanoi Rules (3 examples, 0.076 sec)
Hanoi(0, Src, Aux, Dst, S) = move(0, Src, Dst, S)
Hanoi(s D, Src, Aux, Dst, S) =
  Hanoi(D, Aux, Src, Dst,
  move(s D, Src, Dst,
  Hanoi(D, Src, Dst, Aux, S))).

Fig. 6. Posing the Tower of Hanoi problem for IGOR2 and induced recursive rule set
\end{verbatim}
not able to produce the optimal action sequence for Tower of Hanoi. This difference between human intelligence and artificial intelligence is also reflected in planning which is faster for Tower of Hanoi than for tower problems due to the fact that the the problem space of Tower of Hanoi is smaller, highly symmetric and operator application is more constrained.

5 Rule Induction in Reasoning and Language

A classic work in the domain of reasoning is how humans induce rules in concept learning tasks (Bruner, Goodnow, & Austin, 1956). Indeed, this work has inspired the first decision tree algorithms (Hunt, Marin, & Stone, 1966). This work addressed simple conjunctive or more difficult to acquire disjunctive concepts. However, people are also able to acquire and correctly apply recursive concepts such as ancestor, prime number, member of a list and so on.

The acquisition of recursive and even mutually recursive concepts (such as odd/even) is explicitly addressed in the inductive logic programming system ATRE (Malerba, 2003) and also covered by IGOR2.

In the following, we will focus on the concept of ancestor which is often used as standard example in inductive logic programming (Lavrač & Džeroski, 1994). The competence underlying the correct application of the ancestor concept, that is, correctly classifying a person as ancestor of some other person, in our opinion is the correct application of the transitivity relation in some partial ordering. We believe that if a person has grasped the concept of transitivity in one domain, such as ancestor, this person will also be able to correctly apply it in other, previously unknown domains. For example, such a person should be able to correctly infer is-a relations in some ontology. We plan to conduct a psychological experiment with children to strengthen this claim.

**Ancestor** (9 examples, 10.1 sec)

(and corresponding 4 examples for IsIn and Or)

Ancestor(X, Y, nil) = nilp .
Ancestor(X, Y, node(Z, L, R)) =
IsIn(Y, node(Z, L, R)) if X == Z .
Ancestor(X, Y, node(Z, L, R)) =
Ancestor(X, Y, L) Or Ancestor(X, Y, R) if X /= Z .

Corresponding to:
ancestord(x,y) = parent(x,y).
ancestord(x,y) = parent(x,z), ancestor(z,y).

isa(x,y) = directlink(x,y).
isa(x,y) = directlink(x,z), isa(z,y).

Fig. 7. Learned Transitivity Rules
For simplicity of modeling, we used binary trees as domain model. For trees with arbitrary branching factor, the number of examples would have to be increased significantly. The transitivity rule learned by IGOR2 is given in Figure 7.

Finally, we demonstrate the ability of IGOR2 to learn a phrase-structure grammar. This problem is also addressed in grammar inference research (Sakakibara, 1997). We avoided the problem of learning word-category associations and provided examples abstracted from concrete words (see Figure 8). This, in our opinion, is legitimate since word categories are learned before complex grammatical structures are acquired. There is empirical evidence that children first learn rather simple Pivot grammars where the basic word categories are systematically positioned before they are able to produce more complex grammatical structures (Braine, 1963; Marcus, 2001).

The abstract sentence structures correspond to sentences as (Covington, 1994):

1: The dog chased the cat.
2: The girl thought the dog chased the cat.
3: The butler said the girl thought the dog chased the cat.

The recursive rules can generate sentences for an arbitrary depth which is given as parameter. IGOR2 can also learn more complex rules, for example allowing for conjunctions of noun phrases or verb phrases. In this case, a nested numerical parameter can be used to specify at which position conjunctions in which depth can be introduced. Alternatively, a parser could be learned. Note that the learned rules are simpler than the original grammar but fulfill the same functionality.

original grammar (in the very original grammar, d n v are non-terminals D N V which go to concrete words)
S → NP VP
NP → d n
VP → v NP | v S
examples
fmod GENERATOR is
  *** types
  sorts Cat CList Depth .
  ops d n v : → Cat [ctor] .
  op 1 : → CList [ctor] .
  op __ : Cat CList -> CList [ctor] .
  op s_ : Depth -> Depth [ctor] .
  *** target fun declaration
  op Sentence : Depth -> CList [metadata "induce"] .
  *** examples
  eq Sentence(1) = (d n v d n !) .
  eq Sentence(s 1) = (d n v d n v d n !) .
  eq Sentence(s s 1) = (d n v d n v d n v d n !) .
learned grammar rules (3 examples, 0.072 sec)
Sentence(1) = (d n v d n !)
Sentence(s N) = (d n v Sentence(N))

Fig. 8. Learning a Phrase-Structure Grammar
We presented the inductive programming system IGOR as a mechanism for inducing generalized rules from example experience. Our focus was to demonstrate IGOR’s capability to induce generalized strategies for various domains in the area of problem solving. We could show that IGOR learns rule sets for generating correct and complete action sequences from a small amount of only positive examples in very short time. This corresponds with claims in cognitive science about the human ability of generalization learning (Marcus, 2001; Klahr & Wallace, 1976).

Furthermore, we could demonstrate that IGOR can not only be applied to rule learning in problem solving but also to learn generalized rules for reasoning and natural language processing. Since the induction mechanism underlying IGOR is driven by analyzing structural regularities and thereby domain independent, IGOR can be considered as one proposal for a general cognitive rule acquisition device realizing the claims Chomsky made about a language acquisition device (LAD) which allows to obtain a grammar for a language from linguistic experience (Chomsky, 1959, 1965). Unfortunately, this idea became quite unpopular (Levelt, 1976): One reason is, that only performance and not competence is empirically testable and therefore the idea was only of limited interest to psycho-linguists. Second, Chomsky (1959) argued that there “is little point in speculating about the process of acquisition without much better understanding of what is acquired” and therefore linguistic research focussed on search for a universal grammar. Third, the LAD is concerned with learning and learning research was predominantly associated with Skinner’s reinforcement learning approach which clearly is unsuitable as a language acquisition device since it explains language acquisition as selective reinforcement of imitation.

Since the time of the original proposal of the LAD there was considerable progress in the domain of machine learning (Mitchell, 1997) and we propose that it might be worthwhile to give this plausible assumption of Chomsky a new chance. Obviously, typical approaches to classification learning from rather large sets of positive and negative examples which are based on the assumption of PAC (probably approximately correct) learnability are unsuitable to model a general rule acquisition device. To model the acquisition of a general cognitive competence, an approach is needed which learns from positive examples only, covers all examples correctly and learns a set of rules which capture the regularities observed in these examples. A niche of machine learning research where algorithmic approaches fulfilling these presuppositions are developed and investigated is inductive programming. Inductive programming is based on the notion of language identification in the limit (Gold, 1967) and can be viewed as more general approach to rule learning than grammar infer-
ence (Sakakibara, 1997) since it is concerned with learning recursive programs from examples. As argued above, especially analytical approaches to inductive programming might be helpful instruments to provide an algorithmic foundation for a general rule acquisition device.

Furthermore, the conception of inductive biases (Mitchell, 1997) introduced in machine learning, namely restriction (i.e. language) and preference (i.e. search) bias might be an alternative approach to the search of a universal grammar: Instead of providing a general grammatical framework from which each specific grammar – be it for a natural language or for some other problem domain – can be derived, it might be more fruitful to provide a set of constraints (biases) which characterize what kinds of rule systems are learnable by humans.

There are several aspects of IGOR which we want to explore in future research. One line of work will be concerned with different possibilities to include background knowledge. Currently, background knowledge can – but need not – be presented to IGOR in form of additional examples. We used this possibility, for example, when generating the rule set for tower by presupposing knowledge about how to check whether a block is clear and how to check whether blocks in a partial tower are already in correct sequence isTower. Assuming that humans exploit already acquired knowledge to solve more complex domains, we want to provide a possibility to present background knowledge which has already the form of recursive rule sets. Furthermore, we currently work on an extension of IGOR2 where examples are checked for their subsumption under higher-order program schemes (Hofmann & Kitzelmann, 2010). We could already demonstrate that by this technique, problems which previously could only dealt with if background knowledge was given to IGOR2 now can be learned without recurring to background knowledge. To investigate mechanisms for constructing examples suitable for IGOR we currently are working on embedding IGOR in a robot scenario where the robot learns recursive rule sets for behavior which will generate positive reinforcements. Here we are experimenting with color patterns with underlying grammar structures such as \((\text{red})^n(\text{green})(\text{red})^m\). Only if the robot accesses patterns belonging to this grammar it will be rewarded. From some examples, it should generalize the grammar rules which allow it to select rewarding patterns of arbitrary complexity.

References


E. Soloway & J. C. Spohrer (Eds.), *Studying the novice programmer* (p. 209-228). Lawrence Erlbaum.


