Applying Explanation-Based Learning to Natural-Language Processing (part 2)
by
Manny Rayner and Christer Samuelsson
Applying Explanation-Based Learning to Natural-Language Processing, part 2

Manny Rayner and Christer Samuelsson

Swedish Institute of Computer Science
Box 1263, S-164 28 KISTA
Sweden

manny@sics.se, sam@sics.se

Abstract

Explanation-based learning is a technique which attempts to optimize performance of a rule-based system by adding new rules constructed from generalizations of successfully-solved examples. The paper summarizes previous work showing how this idea can be used in natural language processing, and describes experiments in which the EBL method was applied to the CHAT-80 system of Pereira and Warren. In particular, we address the problem of assuring the utility of learning a rule, since the benefit of a learned rule may not outweigh the increased search time incurred in checking its applicability. We show that this problem can be overcome in the NL domain by indexing acquired rules by their lexical constraints, which in general vastly reduces the number of potentially applicable rules. Such an indexing method was implemented and timing studies were made comparing its access speed to that of normal linear search. The indexing scheme required an average access time of 35 - 40 ms independent of the number of learned rules. The results suggest that the overhead of the indexing scheme is small.
$iv(Agr,X^\{IV,X\}) \rightarrow [IV], \{lex(IV,iv(Agr))\}$.  

\begin{align*}
\text{lex}(\text{john}, \text{pn}) . \\
\text{lex}(\text{a}, \text{det}) . \\
\text{lex}(\text{cat}, n(3-s)) . \\
\text{lex}(\text{loves}, tv(3-s)) . \\
\text{lex}(\text{sleeps}, iv(3-s)) . \\
\end{align*}

This grammar can parse a few trivial sentences like "John sleeps" or "John loves a cat", and associate with each a corresponding expression in first-order logic; we will now show what happens when the second sentence is generalized with respect to the lexicon.

We first construct the normal derivation tree for the original sentence, or to be more exact for the proposition

$$s(\text{quant}(a,A,[\text{cat},A],[\text{loves},\text{john},A]), [\text{john,loves,a,cat}],[\{}]).$$

(The first argument - the logical form - is to be read as "An A such that [cat,A] is also such that [loves,john,A].") The derivation tree will be as shown in figure 1, where each node has been marked with the number of the clause resolved on at that point.

![Diagram](image)

Figure 1

Now we want to generalize away the lexical information present. To do this, we perform the same series of resolution steps, but this time omitting all resolutions with unit clauses of the type \text{lex}(_-,-_); this will yield us the conditional derivation tree in figure 2, where the assumptions have been written in bold-face. Since the new tree represents a valid derivation for any values of the (meta-)variables A, B, C, D and Agr, it thus constitutes a proof of the derived rule at the bottom of
as pointed out in the previous paper, this kind of transformation makes precise the idea of lexical generalization originally suggested in [Ramsay 85].

\textit{Generalized derivation tree}

\begin{align*}
\text{lex}(A, \text{pn}) & \quad \text{lex}(B, \text{tv}(3-s)) & \quad \text{lex}(C, \text{det}) & \quad \text{lex}(D, n(\text{Agr})) \\
(6) & \quad (9) & \quad (7) & \quad (8) \\
\text{pn}(A^\text{VP})^\text{VP}, & \quad \text{tv}(3-s), & \quad \text{det}(X^\text{S1})^\text{S}^\text{S2}^\text{S}, & \quad n(\text{Agr}, Y^\text{D}, Y^\text{D}, Y^\text{D}, Y^\text{D}) \\
[A|R], R, & \quad X^Y[B,X,Y], & \quad [C|R], R, & \quad [D|R], R \\
(2) & \quad (1) & \quad (3) & \\
\text{np}(3-s, (A^\text{VP})^\text{VP}, & \quad \text{vp}(3-s, X^\text{quant}(C,Y,[D,Y],[B,X,Y]), & \quad \text{np}(A^\text{gr}, Y^\text{S}^\text{quant}(C,Y,[D,Y],[B,X,Y]), & \\
[A|R], R) & \quad [B,C,D|R], R) & \quad [C,D|R], R) & \quad [C,D|R], R \\
(4) & \quad (1) & \quad (3) & \\
\text{s}(\text{quant}(C,Y,[D,Y],[B,A,Y]), & \quad \text{s}(\text{quant}(C,Y,[D,Y],[B,A,Y]), & \quad \text{s}(\text{quant}(C,Y,[D,Y],[B,A,Y]), & \\
[A,B,C,D], [1]) & \quad [A,B,C,D], [1]) & \quad [A,B,C,D], [1]) & \\
(1) & \quad (1) & \quad (1) & \\
\text{Generalized derived rule} & \quad \text{Generalized derived rule} & \quad \text{Generalized derived rule} & \\
\text{s}(\text{quant}(C,Y,[D,Y],[B,A,Y]), [A,B,C,D], [1]) & : - & \text{s}(\text{quant}(C,Y,[D,Y],[B,A,Y]), [A,B,C,D], [1]) & : - \\
\text{lex}(A, \text{pn}), & \quad \text{lex}(B, \text{tv}(3-s)), & \quad \text{lex}(C, \text{det}), & \quad \text{lex}(D, n(\text{Agr})).
\end{align*}

\textit{Figure 2}

The example we have just seen should hopefully have given the reader a reasonable intuitive picture of what EBL is capable of doing in the context of natural language processing. The previous paper demonstrated the feasibility of using the method on logic grammars which, though small, were not trivial in size; none the less, it was fairly clear from the referees' and other readers' comments that something more substantial would be required if people were to be convinced of the general utility of the scheme. In particular, there seemed to be three main questions to be answered:

A) EBL as described is only applicable to pure Horn-clause programs. Can NLP systems large enough to be taken seriously be "cleaned up" (i.e. converted to pure Horn-clause form),

---

1 Some readers may feel, as we do, that this argument is a little clumsy. It in fact appears that it can be formulated rather more elegantly in the language of partial inductive definitions [Hallnäs 87]; although of considerable theoretical interest, further discussion of this point would unfortunately take us too far from the main theme of the paper.

2 The largest was the logic grammar for Swedish comparatives from [Rayner & Banks 88]. This contains about 80 DCG rules and an additional 130 Prolog clauses.
without either seriously lowering performance or requiring the rewriting of large sections of code?

B) Even if A above can be answered affirmatively, can the extraction of learned operators be performed with a reasonable expenditure of time and space for a large system?

C) As S. Minton pointed out last year in a much-remarked-on paper from AAAI [Minton 88], it is very important with EBL applications to establish that the derived operators are actually worth the trouble of acquiring. If the average amount of time lost in unsuccessful look-ups of an operator exceeds the average saving when the operator can be applied, the operator is not worth acquiring. What is the situation here with respect to NLP?

So far the past: now the present. The work reported here can essentially be read as an attempt to answer the questions just posed from a practical experimental viewpoint; our general conclusions are positive, and seem to suggest fairly strongly that NLP, even on a realistically large scale, is indeed an unusually suitable application domain for EBL. Most of our argumentation will be based on a detailed case study using the well-known CHAT-80 system of Pereira and Warren [Pereira 83].

The rest of the paper is organized as follows. Section 2 describes the EBL interpreter, which has been extended in several ways compared to that described in the previous paper; in section 3 we consider the problems arising from non-Horn-clause constructs in CHAT, and describe the changes needed to rewrite the system as a pure Prolog program. Section 4 then addresses the issue of efficiently accessing learned rules, and describes a simple solution which indexes them using the lexical information required by each rule from its input string. In section 5 we present results from the CHAT experiments, which support our two main claims: that EBL can practically be applied to virtually the whole of a large-scale natural-language system, and that lexical indexing of learned rules represents a considerable improvement on naive sequential search. In the last section, we summarize our conclusions. Examples of output from the CHAT experiments are given in appendix 1, and the full program code for the EBG interpreter and the rule-indexing compiler in appendix 2.

2. An EBL interpreter in Prolog

This section is to be read primarily as an extended set of comments to the code in appendix 2. To begin with, we describe the system's input and output. The user supplies the following:

- A Prolog program, which corresponds to the background theory Γ in the theoretical picture presented above.
- A set of clauses for the predicate operational_goal/1: these define the operationality criterion Λ.
- A set of clauses for the predicates built_in_goal/1 and insufficiently_instantiated/1. These (as the names would suggest) are for dealing with built-in Prolog goals: the first predicate holds of a goal G iff G is "built-in" (i.e. has no accessible clauses); the second predicate holds if G is both built-in and insufficiently instantiated in its present state to be capable of being called without producing an error condition.

With this information available, a call

:- generalize(+Goal, ?Rule, ?Sub_rules)
will instantiate `Rule` to a rule, whose head is a literal which subsumes `Goal`, and whose body is a list of operational literals. `Sub_rules` will be instantiated to a list of rules, one for each literal in the body of `Rule`, constructed by recursively applying the interpreter to each literal in turn and temporarily suppressing the operability criterion for that literal during the call.

Given the above, it is hopefully fairly easy to understand the interpreter's construction. The top predicate `generalize/3` calls `generalize/4`, which in turn calls the two predicates `generalize1/7` and `simplify_conditions/2`. The first of these is the core of the generalizer, and closely resembles the program from [van Harmelen & Bundy 88]; the only obscure point is perhaps the third argument (Status), which when instantiated to `force_non_operational` locally suppresses the operability criterion for the called goal. As can be seen, the last four arguments together form two difference lists, one for the operational conditions on the main rule, and one for the learned sub-rules.

The output conditions from `generalize1/7` are then fed into `simplify_conditions/2`. As has been pointed out by Minton and others, rules learned by EBL methods are usually clumsy and verbose, and need to be subjected to further processing (typically some kind of partial evaluation): in the present case, a learned rule produced by `generalize1/7` normally contains about ten to twenty calls to "built-in" predicates like `=/2`, `dif/2`, and `functor/3`, most of which are sufficiently instantiated to be evaluable at compile-time. (In particular, a call to `=/2` can always be removed, and a call to `dif/2` can be removed if its arguments cannot be unified with each other).

The condition-simplifier uses a very simple uniform method: any condition which has exactly one solution is called at compile-time and is then removed from the list. Since this can lead to variables becoming instantiated, the process is allowed to continue until no new reduction is possible. As can be seen from the code, the only complicated part of the condition simplifier is in fact the definition of the "single-solution" predicate, which occupies most of the last page.

Incidentally, examination of the examples in appendix 1 shows that the simplification technique sometimes has unexpected results. A particularly striking case is sentence 5, "how large is the smallest American country?"; the generalizer, set to treat lexical items as operational, generalizes `smallest` to something equivalent with "superlative form of an adjective with negative scale". The simplifier, however, discovers that `smallest` is in fact the only word in CHAT-80's lexicon which meets these specifications, and makes it specific again. Although this is a fairly extreme example, it illustrates a common effect: words which are functionally unique in the lexicon will normally be made specific in the learned rule. In general, the fact that mechanical application of these simple criteria usually results in intuitively acceptable generalizations seems to us a good advertisement for the method as a whole.

3. Dealing with non-Horn-clause programs

We now turn to the problem of how to cope with "dirty" Prolog programming techniques. As already indicated, a potentially serious problem with the EBL interpreter is that it is at present only capable of handling pure Horn-clause programs; it is natural to wonder whether this makes it unsuitable when dealing with realistically large systems.

It is of course impossible to give any definite positive answer to this question; there may always exist some system, or some way of writing systems, which is inherently incapable of being reduced to pure Horn-clauses. However, our initial experiments with CHAT-80 do not point to this conclusion. It has in fact (with some reservations described below) proved surprisingly easy to remove all extra-logical constructs from the code. In all, only about 20 changes needed to be
made, nearly all of them involving the removal of cuts\(^1\); these were replaced by appropriate calls to \textit{dif}. One example should be enough to show how this can in general be done.

The following small piece of code is taken from the part of CHAT-80 that handles quantifier scoping:

\begin{verbatim}
compare_dets(Det0, Q, [quant(Det,X,P,Y) | Above], Above, Below, Below) :-
    open_quant(Q, Det1, X, P, Y),
    governs(Det1, Det0), !,
    bubble(Det0, Det1, Det).

compare_dets(Det0, Q0, Above, Above, [Q | Below], Below) :-
    lower(Det0, Q0, Q).
\end{verbatim}

We also need the definitions of the auxiliary predicates \textit{open_quant}/5 and \textit{governs}/2:

\begin{verbatim}
open_quant(qquant(Det,X,P,Y), Det, X, P, Y).

governs(Det, set(J)) :-
    index_det(Det, I),
    I \(\subseteq\) J.

governs(Det0, Det) :-
    index_det(Det0, _),
    ( index_det(Det, _);
    Det=det(_);
    Det=quant(_, _)
    )

govern(_, void).

govern(_, lambda).

govern(_, id).

govern(det(each), question([_ | _]))

govern(det(each), det(each)).

govern(det(any), not).

govern(det(Strong), Det) :-
    strong(Strong),
    weak(Det).
\end{verbatim}

We want to eliminate the cut in the third line of \textit{compare_dets}/6. To do this, we observe that the effect of the cut is to allow the second clause to be called exactly when the conjunction of the first two literals in the first clause fails. This can evidently occur in two possible ways: either the first literal fails, or the first literal succeeds and the second literal then fails irrespective of how the first one succeeds. Since \textit{open_quant}/5 is in fact defined by a single unit clause, this second case can be simplified to: the first literal succeeds, and the second literal fails. Assuming for the moment the existence of "negative" predicates \textit{not_open_quant}/5 and \textit{doesnt_govern}/2, which succeed exactly when their "positive" counterparts would fail, we can rewrite the definition of \textit{compare_dets}/6 as:

\(^1\)There were also a few calls to \(=\subseteq\); these could all be replaced by corresponding calls to \textit{dif} in an unproblematic fashion.
compare_dets(Det0, Q, [quant(Det,X,P,Y)|Above], Above, Below, Below) :-
  open_quant(Q, Det1, X, P, Y),
  governs(Det1, Det0),
  bubble(Det0, Det1, Det).

compare_dets(Det0, Q0, Above, Above, [Q|Below], Below) :-
  (not_open_quant(Q0, _, _, _, _);
   open_quant(Q, Det1, _, _, _),
   doesnt_govern(Det1, Det0)),
  lower(Det0, Q0, Q).

We now consider how to define the predicates not_open_quant/5 and
doesnt_govern/4. Unfortunately this is not entirely trivial, and
demands some global analysis of the program. Looking first at
open_quant, we discover that it is always called with the first
argument instantiated and the others uninstantiated. It follows from
the predicate's definition that it will fail exactly when the first
argument is not a term of the form quant(_, _, _, _, _); not_open_quant
can thus be defined by the clause

not_open_quant(Q, Det, X, P, Y) :-
  not_functor(Q, quant, 5).

where not_functor/2 is defined as

not_functor(T, F, N) :-
  functor(T, F1, N1),
  dif([F, N], [F1, N1]).

We still have to define the predicate doesnt_govern/2. The definition of
governs/2 consists of
nine separate clauses, with no cuts; consequently, we want
doesnt_govern to succeed if and only
if each of these clauses will separately fail. Using a construction like
that we have just seen above,
we produce the following code, which in its turn will require definitions of the predicates
not_index_det, not_strong0, and not_weak; these are generated similarly, and have been
omitted in the interests of brevity.

doesnt_govern(Det0, Det) :-
  (not_functor(Det, set, 1);
   Det = set(J),
   (not_index_det(Det, _);
    index_det(det, I),
    I == J)),
  (not_index_det(Det0, _);
   not_index_det(Det, _),
   not_functor(Det, det, 1),
   not_functor(Det, quant, 2)),
  dif(Det, void),
  dif(Det, lambda),
  dif(Det, id),
  (dif(Det0, det (each));
   not_functor(Det, question, 1)),
  dif([Det0, Det], [det (each), det (each)]),
  dif([Det0, Det], [det (any), not]),
  (not_functor(Det0, det, 1);
\( \text{Det0} = \text{det}(S), \)
\[ \text{not\_strong0}(S); \]
\[ \text{not\_weak}(\text{Det})) \].

Although rather tediously long, it can be seen that the transformation effected above is of a general form, and constitutes a simple way of removing cuts from Prolog programs. The only part of the system which could not be reduced to pure Horn-clause form turned out to be the query simplifier; this is not surprising, since it is an essentially meta-level routine which manipulates Prolog programs, and which uses the normal "dirty" tricks to represent program variables. Even if it were rewritten, however, there would be little likely gain, since query simplification is normally dependent on the exact appearance of the output query and is not capable of being generalized.

In summary, it was a fairly simple business to make CHAT-80 into a clean logic program. However, it is only fair to sound a note of caution here: purity of logic programming style is often as much as anything else an indication of the programmer's skill, which in this case was far higher than one might normally expect; in other words, it might be as well to attempt to repeat the experiment with some more sloppily-coded system. We leave this as a topic for further study.

4. Indexing methods for learned rules

4.1 Why indexing methods are needed.

We will now consider the problem of accessing learned rules efficiently; the content of this and the following section constitute the main experimental results of the paper. We begin by recapping Minton's basic argument: quite simply, there is in general no reason why learning a new rule should improve efficiency. As Minton puts it:

... EBL is not guaranteed to improve problem solving performance. Indeed, in many cases performance may even degrade. The problem is that control knowledge has a hidden cost that can defeat its purpose - the cost of testing whether the knowledge is applicable as the search is carried out. To improve efficiency, an EBL program must generate knowledge that is efficient - its benefits must outweigh its costs ... In practice, it is much more difficult to improve performance over a population of examples than it is to improve performance on isolated examples. [Minton 88, p. 564]

Most of the rest of Minton's paper is concerned with methods for deciding whether learned operators are worth keeping; however, it is assumed throughout that they will be retrieved by linear search. The obvious possibility of using some kind of indexing method is considered briefly, but dismissed in the following terms:

It is sometimes claimed that the utility problem will be solved by the development of highly parallel hardware and/or efficient indexing schemes. This opinion is based on the belief that either of these schemes would make matching (and by extension, memory search) extremely inexpensive. However, this is unlikely to be true, for the following reasons. First, the learned descriptions produced by EBL are neither bounded in number nor in size. Secondly, matching even a single conjunctive description containing variables is NP-complete ... In the worst case, the behaviour of the system may be very poor as the descriptions grow in number of in size. [Ibid, p. 565]

Fortunately for us, it seems however that things are by no means as bad as the passage above would suggest. Minton is naturally correct in pointing out that the general case is intractable; none the less, this does not rule out the possibility of success in good special cases, and there are reasonable a priori grounds for expecting NL to be just such a case. As we have already seen above, learned rules normally consist wholly or in large part of lexical constraints on the words in
the input string, and it is therefore possible to base an indexing scheme on the lexical categorizations demanded by each rule. By using the lexical constraints as a filter, the number of potentially applicable rules can be vastly reduced; in practice, it will often be possible either to reduce the search space at once to a single potentially useful rule, or to conclude that no such rule exists. There is no obvious corresponding solution for the planning and scheduling problems considered by Minton, which presumably explains in part his expressed lack of interest in ideas of this kind.

In the next subsection we go on to describe the implementation in Prolog of a simple way of compiling the lexical information in the input string into a "key", which can then be used as an index into the learned rules; as shown by the experimental results in section 5, this results in an access method considerably more efficient than naive sequential search.

4.2 Description of the "key" method

We will now describe a straight-forward way of constructing an indexing scheme which directly makes use of the "first functor" clause-indexing mechanism provided by most Prolog implementations. The idea is essentially to add an atom, whose print-name summarizes the lexical constraints, as an extra first argument to each learned rule; this atom will thus be the "key" for the rule. In mathematical terms, we are mapping the word category structure of sentences into a space of atomic keys.

When the system tries to find a learned rule applicable to a given input string, the process now works as follows. Each individual item in the string is first looked up in the lexicon to determine its lexical category; items are sorted into the eleven basic classes of nouns, verbs, adjectives, prepositions, conjunctions, pronouns, adverbs, proper names, determiners, numerals and terminators. The list of lexical categories then determines the key: if any of the words are lexically ambiguous, there will be several keys. (In fact, the number of keys will be the product of the number of possibilities for each individual lexical ambiguity). In view of this, it is evidently desirable not to have too large a number of distinct, potentially confusable lexical categories. We have, for example, considered it unnecessary to distinguish between verbs with different subcategorization restrictions, since many verbs can play both an intransitive and a transitive part, depending on context.

Once a category list has been determined, it is then collapsed into an atomic key. This could be done in any one of several ways; the one we have chosen is to associate each category name with a distinct letter, and to construct the atom whose print-name consists of the sequence of letters corresponding to the list of categories. In SICStus, this operation turns out to be cheap, though its efficiency is in general implementation-dependent.

The operation of finding a rule predicate with a particular key as a first argument is very fast. This observation, be it simple, is the fundamental idea of this particular implementation of the indexing method. Once the learned rule associated with the key is accessed, it is applied to the input string. If the rule turns out not to be applicable to the input string, the hunt is still on, and the next rule predicate whose first argument is the key in question is consulted and its associated rule applied. Thus continues a linear search confined, however, to rule predicates bearing the same distinguishing key. Should the search terminate fruitlessly, then the correct rule is not after all present in the data base and one can resort to analyzing the input string by other means.

Let us look at an example. The sentence

\[ \text{Is Canada a large country?} \]

will be associated with the key \text{vndast}, since
"is" is a verb (v),
"Canada" is a proper name (n),
"a" is a determiner (d),
"large" is an adjective (a),
"country" is a proper noun (s, n being already in use) and
"?" is a terminator (t).

We expect to find a corresponding predicate:

```plaintext
rule(vndast, "rule1").
```

where "rule1" applies to this sentence. The sentence

_Is London an old city?_

also maps to vndast and the same rule is most probably applicable to both this sentence and to _Is Canada a large country?_

On the other hand, both

_Is Stockholm the capital of Norway?_

and

_Was Stockholm the capital of Norway?_

map to vndsplt (p for preposition), but the same rule may not be applicable to both due to the difference in tense and there will be two different rules in the data base with identical keys:

```plaintext
rule(vndsplt, "rule2").
rule(vndsplt, "rule3").
```

Thus when different rules correspond to the same key, key indexed search (KIS) results in linear search among these rules.

Words belonging to several lexical categories complicate matters: The sentence

_Do a German swallow a German swallow?_

can be mapped to sixteen different keys since "German" can be regarded as either a noun (s) or an adjective (a) and "swallow" may refer to the bird (s) as well as to the process of devouring it (v); only one of these (corresponding to the key vdsvdas) will however result in a grammatical interpretation. Ambiguities in lexical category result in time for KIS exponential in the number of ambiguities, since the number of keys a sentence can be mapped to increases combinatorially.

---

1 The reader familiar with CHAT-80 should note that a number of new entries have been added to the lexicon, primarily in order to experiment with the introduction of lexical ambiguities. For most of these entries, there is only syntactic information.

2 Actually there are thirty-two distinct keys, taking into account the existence of female deer...
5. Practical experiments

5.1 General description of the experiments

The experiments carried out can be divided into two groups. In the first group, the intent was simply to ascertain that the different variants on the EBL method outlined above could be successfully applied to a non-trivial NL system. These experiments covered both syntactic and semantic processing, first using a simple lexical operationality criterion and then a more complex "two-level" version in which relative clauses were treated as operational. The second group of experiments concentrated on timing studies and used only the syntactic part of the system, with the lexical operationality criterion. We now describe both groups of experiments in more detail.

5.1.1 Applying EBL to CHAT-80

In the first group of experiments, we aimed simply to test the feasibility of applying the EBL interpreter to as large a portion as possible of the CHAT-80 system. A group of 16 typical sentences was selected; most of these were taken from a test file which accompanied our version of the system. CHAT was "cleaned up" as described in section 3 above, to make it a pure Horn-clause program, and the interpreter was applied to the modified system, using the test sentences as examples.

Two different definitions of the operationality criterion were used. The simpler version essentially made "operationality" synonymous with "lexicality": this involved defining as "operational" the 19 predicates which together comprised the lexicon, and the 14 which defined the "templates" for the slot-filling module. In the second version, we introduced extra clauses which had the effect of making relative clauses operational as well; this was in principle also fairly easy to implement, but in practice involved acquiring a rather deeper understanding of the details of CHAT's implementation. Examples of tests carried out using both versions can be found in appendix 2.

The general conclusions we can draw from the first group of experiments are the following:

- EBL can be applied to virtually the whole of a reasonably large NL query system; the only part which was not amenable to generalization was the module which dealt with query simplification. The reasons for this are discussed briefly in section 3 above.

- It was possible to treat relative clauses as generalizable units, and still perform generalization on the whole process. However, it must be stressed that the way in which this was done heavily exploited the special nature of this particular construct, since relative clauses in CHAT are at all stages in the process represented as indivisible entities: it was thus possible to define as "operational" the predicates that handled the representation of the relative clause at all three levels in the system, namely syntax, slot-filling, and logical-form building. With constituents like noun-phrases, whose contribution can get "spread out" through the representation, this would not be feasible, although it is still quite possible to generalize over them at the syntactic level.

We now move on to the second group of experiments, which highlighted performance aspects.

5.1.2 Timing studies

The main purpose of the timing studies was to perform a systematic comparison between key indexing search (KIS) and linear search (LS), with regard to access times. In the case of success, access time was taken as the sum of the time spent searching for a suitable rule and the time spent applying it once it had been found. (Distinguishing between these two processes is of course only meaningful in the KIS case). When no applicable rule existed, the relevant time was simply that
required to return with a failure. Measuring access time in this way corresponds to measuring the
time taken to return from the get_rule call in the code.

The experiments were set up primarily to test the following specific hypotheses:

- Access time for LS should be linear in the size of the data base, whereas access time for KIS
  should be independent of it.

- Access time for KIS should contain a term proportional to the length of the sentence.

- Access time for KIS should be exponential in the number of lexical ambiguities.

- The overhead in KIS pertaining to the construction and manipulation of keys should be small.

Experiment 1

To examine the relationship between access time and data base size, the system was set to learning-
mode and given 50 test sentences to generalize, producing 50 different rules. The learned rules
where then compiled, the system switched to using-mode and reconfronted with the test sentences,
which were successfully matched with the rules. The KIS immediately struck upon the correct rule
whereas the LS systematically tried rule after rule until the correct one was encountered.

Experiment 2

In order to confirm our suspicions of the influence of sentence length on the efficiency of KIS, we
repeated the previous experiment on a set of 65 test sentences whose lengths varied from 3 to 23
words. Access times were then plotted against sentence lengths.

Experiment 3

A database containing 65 compiled learned rules was fed a series of sentences, each of which
contained between zero and seven ambiguities; each ambiguous word belonged to exactly two
lexical categories. The sentences had all been selected so as not to match any rule in the database,
simulating worst-case behaviour; LS thus searched sequentially through all 65 rules, while KIS
generated $2^n$ keys, $n$ being the number of ambiguities in the sentence. The logarithm of the failure
time for each indexing method was then plotted against the number of ambiguities.

Experiment 4

This experiment was performed in order to verify that KIS behaves as a linear search among rules
with identical keys. Thus 40 test sentences were submitted to KIS and LS in a database of eight
learned rules with identical keys. In order to study the KIS overhead, we plotted the difference in
access times of KIS and LS against the number of rules consulted.
5.2 Results

Experiment 1

Linear search (LS) is linear in the number of sentences, while key indexed search (KIS) is constant and much faster.

![Experiment 1, Access Time for LS and KIS](image)

It is interesting to note that a successful KIS look-up requires, on the average, approximately 35 to 40ms\(^1\). This should be attributed separately to constructing the key, matching the key with the correct key argument of the rule predicate and to applying the rule. The large variation is in part explained by differences in sentence length. Note the covariation of LS and KIS, indicating that most time is spent applying the rule.

Examining the equation of the LS line, it becomes clear that we can interpret the gradient (~7ms) as the average cost of an unsuccessful rule application. Since every call eventually succeeds somewhere, the constant term (~30ms) is thus the average difference between a successful and unsuccessful application.

---

\(^1\)All times refer to SICStus, run on a SUN 3/60 under UNIX.
Experiment 2

Time for KIS increases with the length of sentence, the dependence being asymptotically linear, but this effect is, at all sensible lengths, dominated by other effects for which we can offer no rational explanation. The good correlation with a straight line below was accomplished by averaging over five different sentences of each length; deviations from the mean value of 30 to 40 percent were quite common.

![Graph of Experiment 2, KIS vs Sentence Length](image)

\[ y = 4.7 + 2.7x, \text{ Corr. factor} = 0.99 \]

The time spent applying a learned rule will depend on the structure of the rule itself. A long sentence will in general correspond to a rule consisting of more conditions than that of a shorter sentence. We observed an increase with length of accessing time in linear search as well, partially explaining the covariation in the previous experiment. We also noted a variation of accessing time, in size rivaling the former, that we deemed to originate from qualitative, rather than quantitative, differences between rules. We draw two conclusions concerning KIS and sentence length:

1) The construction of the key does not significantly contribute to the linear behavior.

2) Variations of at least the same order of magnitude as the linear increase occur for reasons not easily explained by inspecting the input string.
Experiment 3

Time for KIS increases exponentially with number of lexical ambiguities, but KIS is still acceptably fast at seven ambiguities, this being far more than typical for a database application.

![Graph showing ln(Time) vs. No of Amb. Word]

The idea of plotting the logarithm of time consumed originates in the assumption that KIS will make access time proportional to the number of possible keys; the construction of the experiment should thus make it proportional to $2^n$, where $n$ is the number of lexical ambiguities. The slope of the curve should then be very close to ln2, which it is not; the slope is in fact 0.53 and ln2 is approximately 0.70. This anomaly appears to depend on fairly obscure facts about the complexity of the key-construction algorithm; its consequence is that the average time taken to construct a key and index on it decreases with the number of keys, dropping from $\sim$5ms (no ambiguities) to $\sim$1.5ms (7 ambiguities). The results of the next experiment provide a more accurate estimation of the cost for the key-indexing operation.

The cost of unsuccessfully inspecting 65 rules in the case of LS is 490 ms giving an average of 7.5 ms per look-up, which is in good agreement with the first experiment.
Experiment 4

Not only did the KIS performance resemble that of LS, they were sufficiently strongly coupled to provide a good estimate of the key-indexing overhead:

**Experiment 4, Access Time for KIS and LS**

![Graph showing access time for KIS and LS](image)

A value for the overhead can be derived by plotting the difference of the accessing times for KIS and LS versus the number of rules consulted:

**Experiment 4, Difference of KIS and LS**

![Graph showing difference of KIS and LS](image)

We observe a linear increase in the overhead which can be ascribed to one successful key-matching operation per rule; thus the gradient gives us a good estimate of its value, 3.0ms per successful key matching. We associate the constant, 0.9ms, with the key construction.
6. Summary and conclusions

Our experiments would seem to show that it is quite feasible to apply explanation-based learning to a substantial natural-language application. Extraction of learned rules can be made to apply to the whole analysis process, with the possible exception of query simplification, and the EBL interpreter is acceptably fast. There is a readily-implementable method which "simplifies" learned rules, by removing conditions which can already be called at compile-time.

The main problem, as pointed out by Minton, is that of efficient utilization of the learned operators; the benefits of a learned rule may not outweigh the increased costs in search time of having it in the data base. In the NL domain, however, the problem can be circumvented by using each rule's lexical constraints as a filter, vastly reducing the number of potentially applicable rules. This can be done by adding an atom, a key, whose print-name summarizes the lexical constraints, as an extra first argument to each learned rule and utilizing the "first functor" clause-indexing mechanism provided by most Prolog implementations.

Such an indexing strategy was implemented and timing studies were made comparing its access speed to that of linear search. The dependence of data base size, length of input string and number of lexical ambiguities in the input string were examined. The indexing scheme required access time independent of the number of learned rules. It equalled the time required by linear search in a data base consisting of one single learned rule, on the average 35 - 40 ms. This suggests that the overhead of the indexing scheme is small, which is also supported by our estimation of the time consumed by the indexing mechanism to ten percent of the time spent applying the rule once it has been found.

The indexing scheme required, on the average, access time proportional to the length of the input string, but other factors seem to dominate this.

Lexical ambiguities force the construction of, in the worst case, keys corresponding to every possible combination of lexical categories. Thus, we are faced with exponential growth in the number of lexical ambiguities. Yet, the indexing scheme is still acceptably fast at seven ambiguities, this being far more than typical for a database application.

The access time of a learned rule with a particular key is proportional to the number of rules with that key due to a linear search among these rules. There are, however large variations, as in the case of LS, in time required by the rule applying parts of the search methods.

It would now appear that the logical next step is to test the EBL method on a full-scale system. In particular, the ultimate success of failure of the scheme depends on the distribution of query-types with regard to frequency of use; although there are good reasons for expecting the two or three hundred most common types to account for a large percentage of all queries, it is clearly crucial to determine just what these figures are. We expect that a detailed study of these questions will form the basis for a future paper.

Acknowledgments

A large part of this work was performed during February and March of 1989, while the first author was a guest researcher at ICOT. I would very much like to thank my hosts, in particular Shunichi Uchida, Ryoichi Sugimura and Koiti Hasida, both for inviting me and for a number of productive discussions.

The second author would like to thank the people at SICS for contributing to a very conductive and inspiring atmosphere to work in. Especially, I would like to thank Annika Wärn, Mats Carlsson and Thomas Sjöland for patiently helping me when I encountered difficulties.
References


[Hallnäs 87] L. Hallnäs, Partial Inductive Definitions, SICS research report, 1987


[Rayner 88] Rayner, M. Applying Explanation-Based Generalization to Natural-Language Processing, Proc. FGCS-88, Tokyo, 1988


Appendix 1: examples using the CHAT-80 system

1: does france border spain?

query([[does,A,B,C,?],(answer[[]:-D])]:- 
  name(A), %A is a name 
  verb_form(B,E,inf,F), %B is the inf. form of verb E 
  verb_type(E,G+trans), %E is transitive 
  name(C), %C is a name 
  verb_type(E,H+trans), %E is transitive (again) 
  trans(E,I,A,J,C,D,[],-(+-0)),K), %Slot-filling: the "trans" template 
  %for E, giving logical form D. 
  %Variables I and J filled by A and C. 
  name_template(A,I), %Name template for A, variable I. 
  name_template(C,J). %Name template for C, variable J.

Timing information:

Processing time: 220 %Normal processing 
Time for successful lookup: 40 %Learned rule 
Looked-up answer: answer([[]:-borders(france,spain)] 
Times for unsuccessful lookup: [0,0,0,0,0,0,0,0,0,0,0,0,0,0] 
Av. time for unsuccessful lookup: 1.42

yes

2: what is the population of belize?

query([[A,B,be,C,of,D,?],(answer[[E]]):-F]) :- 
  int_pron(A,undef), 
  verb_form(B,be,G+fin,3+B), 
  noun_form(C,1,1sin), 
  name(D), 
  property(I,J,E,K,D,F,[],-(+-0)),L), 
  name_template(D,K).

Timing information:

Processing time: 400
Time for successful lookup: 20
Looked-up answer: answer([[A]]):-population(belize,A)
Times for unsuccessful lookup: [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0]
Av. time for unsuccessful lookup: 0.0
3: which country's capital is London?

query([A,B,'s,C,is,D,?],(answer([E]):-D (F,G)) :-
    int_art(A,H=E,si,nt_det(I-J)),
    noun_form(B,K,si),
    noun_form(C,L,si),
    name(D),
    property(L,M,D,I,J,G,[],-(-(0)),N),
    thing(K,I,J,F,[I,O],
    name_template(D,M)).

Note that the slot-filling information for K "country" and L "capital" are quite different: in this domain, "country" corresponds to a property and "capital" to a relation. EBG captures this automatically.

Timing information:

Processing time: 340
Time for successful lookup: 0
Looked-up answer: answer([A]):-london^country(A),capital(A,london)
Times for unsuccessful lookup: [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0]
Av. time for unsuccessful lookup: 0.0

4: which is the largest African country?

query([A,B,th,e,C,D,E,?],
    (answer([F]):-G^setof(H:I,(J,K,L,G),aggregate(max,G,F)) :-
    int_pron(A,undef),
    verb_form(B,be,M+fin,3+N),
    sup_adj(C,O),
    adj(D,F),
    noun_form(E,Q,si),
    thing(Q,R,I,J,[I,S],
    sign(O,+),
    attribute(O,R,I,T,H,K),
    restriction(D,R,I,L)).

Timing information:

Processing time: 380
Time for successful lookup: 20
Looked-up answer: answer([A]):-
    B^setof(C:D,(country(D),area(D,C),african(D)),B),
    aggregate(max,B,A)
Times for unsuccessful lookup: [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0]
Av. time for unsuccessful lookup: 0.0
5: how large is the smallest american country?

query([[how,A,is,small,smallest,B,C,?]],
      (answer([D]):-
       E^F^setof(G:H, (I,area(H,G),J),F),aggregate(min,F,E),K)) :-
      adj(A,quant),
      adj(B,I),
      noun_form(C,M,sin),
      thing(M,feature&place&N,H,I,[]),O),
      attribute(small,feature&place&N,H,measure&area,G,area(H,G)),
      restriction(B,feature&place&N,H,J),
      attribute(A,feature&place&N,E,P,D,K).

"smallest" isn't generalized - this is not a bug, it's a feature! Since DAI
doesn't contain any other word like "smallest" (superlative form of negative
&measure adjective), compression automatically chooses it at generalize-time.

Timing information:

Processing time: 320
Time for successful lookup: 20
Looked-up answer: answer([A]):-
B^C^set(G:D,E,(country(E),area(E,D),
           american(E)),C),aggregate(min,C,B),area(B,A))
Times for unsuccessful lookup: [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0]
Av. time for unsuccessful lookup: 0.0

22
Relative clauses have been defined as operational.

6: what is the ocean that borders African countries and that borders Asian countries?

query([A,B, the, C|D], (answer([E]): - F, G, H)) :-
  int_pron(A, undef),
  verb_form(B, be, I+fin, 3+J),
  noun_form(C, K, sin),
  rel(3+sin, L, M, D, [and|N], [], []),
  rel(3+sin, O, #(1, l, 0, 1), N, [?], [], []),
  thing(K, P, E, F, [J], Q),
  i_rel(L, F-E, R, [], @true, [], [], -(+-+(0))),
  i_rel(O, F-E, S, [], @true, [], [], +/--(+-0))),
  close_tree(R, G),
  close_tree(S, H).

Subrules:

Syntax of the relative clauses (actually there are two copies of each of the following three rules).

rel(A, rel(B, s(np(A, wh(B)), [])),
       verb(C, active, D+fin, [], pos),
       [arg(dir, np(3+E, nucleus(generic, [adj(F)], G), [], []), [I])], [], []),
       #(1, l, 0, 1), [that, H, F, I|J], J, K, K) :-
       verb_form(H, C, D+fin, A),
       verb_type(C, L),
       verb_type(C, M+trans),
       adj(F, N),
       noun_form(I, G, E).

Slot-filling for the relative clause.

i_rel(rel(A, s(np(E, wh(C-D)), []), verb(E, active, F, G, pos),
       [arg(dir, np(H, nucleus(I, [adj(J)], K), [], []), []), []], []), A, rel_pred(quant(id, C-D, @true, @true, [], C-L),
       id, M, [quant(I, N-O, @P, @Q & @true, [], [N-R]) & S, T, S, T, [], U) :-
       verb_type(E, V+trans),
       trans(E, C, L, N, R, M, [], -(+-(-U))), W),
       thing(K, N, O, P, [], X),
       restriction(J, N, O, Q),
       dif(I, det(the(plu)))

23
Reduction to logical form.

close_tree(rel_pred(quant(id,A-B,@C,@D,[],E-B),
    id,F,[quant(generic,G-H,@I,@@J & @K,[],L-H)]) & @M,
    H^(((I,true),J,K),((C,true),D),(true,F,true,true),M)).

Timing information:

Processing time: 760
Time for successful lookup: 20
Looked-up answer: answer([A]):= %Before query simplification!
ocean(A),
    B^(((country(B),true),african(B),true),
        ((true,true),true),
        (true,borders(A,B),true,true),true),
    C^(((country(C),true),asian(C),true),
        ((true,true),true),
        (true,borders(A,C),true,true),true)
Times for unsuccessful lookup: [0,0,0,0,20,0,0,0,20,0,0,0,0,20,0]
Av. time for unsuccessful lookup: 5.33
Appendix 2: program code

The generalizer

:- dynamic 'none one many'/1.

generalize(Goal,Rule,Sub_rules) :-
    generalize(Goal,normal,Rule,Sub_rules).

generalize(Goal,Status, (G_goal :- Conds), Sub_rules) :-
    generalize1(Goal,G_goal,Status,Conds1,[],Sub_rules,[]),
    simplify_conditions(Conds1,Conds).

generalize(true,true,_,C,C,S,S) :- !.

generalize1((H,T),(G_H,G_T),_,In_conds,Out_conds,In_r,Out_r) :- !,
    generalize1(H,G_H,_,In_conds,Next_conds,In_r,Next_r),
    generalize1(T,G_T,_,Next_conds,Out_conds,Next_r,Out_r).

generalize1((H:T),(G_H;G_T),_,In,Out,In_r,Out_r) :- !,
    (generalize1(H,G_H,_,In,Out,In_r,Out_r);
        generalize1(T,G_T,_,In,Out,In_r,Out_r)).

generalize1(Goal,G_goal,Status,In,Out,In_r,Out_r) :-
    non_operational_user_goal(Goal,Status),!,
    expand_goal(Goal,G_goal,Body,G_body),
    generalize1(Body,G_body,_,In,Out,In_r,Out_r).

generalize1(Goal,G_goal,_,[G_goal|Out],Out,Rules,Rules) :-
    built_in_goal(Goal),!,
    call(Goal).

generalize1(Goal,G_goal,normal,[G_goal|Out],Out,[Rule|Out_r],Out_r) :-
    operational_goal(Goal),!,
    generalize(Goal,force_non_operational,Rule,_).

generalize1(Goal,_,_,_,_,_,_) :-
    nl,nl,write('*** Error: no way to deal with '),
    write(Goal),nl,!,
    fail.

expand_goal(Goal,G_goal,Body,G_body) :-
    functor(Goal,P,N), functor(Head,P,N),
    clause(Head,Body),
    copy_term([Head,Body],[G_head,G_body]),
    Goal = Head, G_goal = G_head.

25
% Really this should be done better - e.g. by caching the result?

non_operational_user_goal(Goal,Status) :-
    (Status == force_non_operational,
    \+operational_goal(Goal),
    \+built_in_goal(Goal)).

simplify_conditions(Conds0,Conds) :-
    call_unique_solutions_now(Conds0,Conds1,Flag),!,
    (Flag = no,
      Conds0 = Conds;
      Flag = yes,
      simplify_conditions(Conds1,Conds)).

call_unique_solutions_now([],[],no).
call_unique_solutions_now([Goal|Rest],Rest1,yes) :-
    \+uncallable builtin_goal(Goal),
    single_solution(Goal,Unique_solution),
    Goal = Unique_solution,
    call_unique_solutions_now(Rest,Rest1,\). call_unique_solutions_now([Goal|Rest],[Goal|Rest1],Flag) :-
    call_unique_solutions_now(Rest,Rest1,Flag).

uncallable built_in_goal(G) :-
    built_in_goal(G),
    insufficiently instantiated(G).

single_solution(Goal,Unique_solution) :-
    none_one_many(Goal,one(Unique_solution)).

none_one_many(Goal,Result) :-
    asserta('none one many'('start marker')),
    none_one_many_1(Goal,Result1),
    retract('none one many'('start marker')),
    Result = Result1.

none_one_many_1(Goal,several) :-
    call(Goal),
    none_one_many_2(Goal),!.

none_one_many_1(_,one(Unique)) :-
    'none one many top'(solution(Unique,G)),!,
    retract('none one many'(solution(Unique,G))).

none_one_many_1(_,none).
none_one_many_2(Goal) :-
  'none one many top'('start marker'),
  ground_copy(Goal,Goal1),
  asserta('none one many'(solution(Goal,Goal1))),
  fail.

none_one_many_2(Goal) :-
  'none one many top'(solution(G,Old_goal)),
  ground_copy(Goal,Goal1),
  dif(Goal1,Old_goal),
  retract('none one many'(solution(G,Old_goal))).

'none one many top'(Top) :-
  'none one many'(X),!,
  X = Top.

ground_copy(G,G1) :-
  copy_term(G,G1),
  numbervars(G1,0,_) .

Indexing routines

:- dynamic rule/2.

% Routines for saving learned rules:

put_rule(New_rule) :-
  copy_term(New_rule,Rule_to_put),
  copy_term(New_rule,ebl_rule(Sentence,Conditions)),
  make_repr_sentence(Sentence,Conditions),
  make_key(Sentence,KeyWord),
  asserta(rule(KeyWord,Rule_to_put)).

make_repr_sentence(Sent,Conds) :-
  % The call to apply_conds finds some instantiation of the
  % uninstantiated words in Sent and the call to ambiwords ensures
  % that no mistake due to lexical ambiguities is introduced when
  % constructing the key.
  apply_conds(Conds),
  ambi_words(0,Sent).

apply_conds([]).
apply_conds([C|Cs]) :- C,apply_conds(Cs).

ambi_words(0,[]).
ambi_words(N1,[N|Ws]) :-
  ambiguous(W),
  ambi_words(N,Ws),
  N1 is N + 1.
ambi_words(N,[W|Ws]) :-
    ambi_words(N,Ws).

ambiguous(Word) :-
    lex(Word,C1),
    lex(Word,C2),
    C1 \== C2.

% Routines for key construction:

make_key(List,KeyWord) :-
    make_key_list(List,KeyList),
    atom_chars(KeyWord,KeyList).

make_key_list([],[]).
make_key_list([W|Ws],[K|Ks]) :-
    lex(W,C),
    trans(C,T),
    atom_chars(T,[K]),
    make_key_list(Ws,Ks).

% Key letter table:
trans(verb,v).
trans(noun,s).
trans(adj,a).
trans(prep,p).
trans(num,r).
trans(adv,b).
trans(conj,k).
trans(pron,q).
trans(int,i).
trans(det,d).
trans(name,n).
trans(term,t).

% Routines for accessing learned rules:

get_rule(Sentence,Rule_to_use) :-
    make_key(Sentence,KeyWord),
    rule(KeyWord,Rule),
    copy_term(Rule,Rule_to_use),
    copy_term(Rule,Rule_to_apply),
    apply_rule(Sentence,Rule_to_apply).

apply_rule(Sent,ebl_rule(Sent,Conds)) :-
    apply_conds(Conds).