A note on two simple transformations for improving the efficiency of an ILP system

Vítor Santos Costa[1], Ashwin Srinivasan[2], Rui Camacho[3]

[1] COPPE/Sistemas, UFRJ, Brazil and LIACC, Universidade do Porto, Portugal
[2] Oxford University Comp. Lab., Wolfson Bldg., Parks Rd, Oxford, UK
[3] LIACC and FEUP, Universidade do Porto, Portugal

Abstract. Inductive Logic Programming (ILP) systems have had noteworthy successes in extracting comprehensible and accurate models for data drawn from a number of scientific and engineering domains. These results suggest that ILP methods could enhance the model-construction capabilities of software tools being developed for the emerging discipline of "knowledge discovery from databases." One significant concern in the use of ILP for this purpose is that of efficiency. The performance of modern ILP systems is principally affected by two issues: (1) they often have to search through very large numbers of possible rules (usually in the form of definite clauses); (2) they have to score each rule on the data (usually in the form of ground facts) to estimate "goodness". Stochastic and greedy approaches have been proposed to alleviate the complexity arising from each of these issues. While these techniques can result in order-of-magnitude improvements in the worst-case search complexity of an ILP system, they do so at the expense of exactness. As this may be unacceptable in some situations, we examine two methods that result in admissible transformations of clauses examined in a search. While the methods do not alter the size of the search space (that is, the number of clauses examined), they can alleviate the theorem-proving effort required to estimate goodness. The first transformation simply involves eliminating literals using a weak test for redundancy. The second involves partitioning the set of literals within a clause into groups that can be executed independently of each other. The efficacy of these transformations are evaluated empirically on a number of well-known ILP datasets. The results suggest that for problems that require the use of highly non-determinate predicates, the transformations can provide significant gains as the complexity of clauses sought increases.

1 Introduction

An early Knuth classic [12] describes how the use of run-time statistics gathered from the execution of a program can be used to suggest optimisations. The paper is an empirical study of executing FORTRAN programs selected randomly from semi-protected files stored on Stanford University's computer disks[1]. He summarises the results of the study thus:-

[1] Knuth describes two other less successful methods to obtain data for his experiments. These were: rummaging waste-baskets and recycling bins; and posting a sentry at

> In the early days of computing, machine time was king, and people worked hard to get extremely efficient programs. Eventually machines got larger and faster, and the pay off for writing fast programs was measured in minutes or seconds instead of hours... Therefore most of the emphasis in software development has been in making programs easier to write, easier to understand and easier to change. There is no doubt that this emphasis has reduced the total system costs in many installations, but there is also little doubt that the corresponding lack of emphasis on efficient code has resulted in systems which can be greatly improved... Frequency counts give an important dimension to programs, showing programmers how to make their routines more efficient with comparatively little effort.

By "frequency counts" Knuth is referring here to the frequency with which some recognisable constructions actually occur during execution of a FORTRAN program. He outlines two desirable outcomes of such studies. First, that dynamic profiles of program execution can direct the programmer to simple optimisations that can result in more efficient code. Second, that such profiles (provided they are representative of typical profiles, of course) can direct the design of compilers by indicating what programmers want to do. In this paper we illustrate, on a smaller scale, an instance of realising the first of these outcomes within the province of Inductive Logic Programming (ILP).

Since 1991, ILP systems have been used to construct predictive models for data drawn from diverse domains. These include software analysis [2], engineering [7, 9], environment monitoring [8], biochemistry [10, 11, 16], and language processing [4, 27]. The systems that have been used to construct these models have—at a very coarse level—been fairly similar, namely: search engines that repeatedly examine sets of candidates (the "search space") to find good rules. The time-complexity of most of these systems therefore largely depends on: (1) the size of the search space; and (2) the cost of evaluating the "goodness" of a rule. Recently, stochastic methods have been proposed for reducing the complexity of both these aspects (for example, see [20–22]), at the expense of exactness. That is, lower complexity ILP systems are possible if we are willing to accept sub-optimal rules, or imprecise estimates of the goodness of a rule. Such compromises may not always be acceptable, and we present here two simple transformations that are suggested by examining the dynamic profile obtained by using an ILP system in a fairly conventional manner to construct a model for data. Both transformations do not alter the number of rules examined; but are shown empirically, in some cases, to alter significantly the cost of evaluating a rule.

The paper is organised as follows. In Section 2 we outline a simple procedure constituting an ILP algorithm. This constructs models for data using the greedy set-covering approach common to many extant implementations. Section 3 tabulates the dynamic profile of using an ILP system implementing this procedure,

a card-reader to accost programmers. The former was found to largely yield non-working programs, and the latter too time-consuming to be practical.

when constructing a model on a benchmark dataset. Section 4 describes the two transformations. Section 5 contains an empirical study of the efficacy of the transformations, using some well-known datasets. Section 6 concludes the paper. Prolog code implementing the transformations is in Appendix A.

2 A Simple ILP Implementation

We adopt the following informal prescription for an ILP algorithm (based on [14]). Given: (a) background knowledge B in the form of a Prolog program; (b) some language specification $\mathcal{L}$ describing the hypotheses; (c) an optional set of constraints I on acceptable hypotheses; and (d) a finite set of examples E at least some of whose elements constitute a set of "positive" examples E^+ such that none of the E^+ are derivable from B; the task is to find some hypothesis H in $\mathcal{L}$, such that: (1) H is consistent with the I; and (2) The E^+ are derivable from B, H.

Several prominent ILP systems (for example, CProgol [15]; Golem [17]; and FOIL [19]) use a simple iterative procedure to construct such a hypothesis one clause at a time. Figure 1 shows one version of such a procedure. This is reproduced with minor changes from [22] and we refer to the reader to that paper for proofs of correctness and complexity arguments. Here we will assume that: (a) on each iteration, $search(\cdots)$ will examine a finite number of clauses; (b) the best clause amongst these alternatives requires $search(\cdots)$ to at least check the derivability of each example in $Train_i$; and (c) the check for derivability terminates.

$generalise(B, I, \mathcal{L}, E)$: Given background knowledge B, hypothesis constraints I, a finite training set $E = E^+ \cup E^-$, returns a hypothesis H in $\mathcal{L}$ such that $B \cup H$ derives the E^+.

1. $i = 0$
2. $E_i^+ = E^+$, $H_i = \emptyset$
3. while $E_i^+ \neq \emptyset$ do
   (a) increment i
   (b) $Train_i = E_{i-1}^+ \cup E^-$
   (c) $D_i = search(B, H_{i-1}, I, \mathcal{L}, Train_i)$
   (d) $H_i = H_{i-1} \cup \{D_i\}$
   (e) $E_p = \{e_p : e_p \in E_{i-1}^+ \ s.t. \ B \cup H_i \models \{e_p\}\}$.
   (f) $E_i^+ = E_{i-1}^+ \setminus E_p$
4. return H_i

Fig. 1. A simple ILP implementation. The function $search(\cdots)$ is some search procedure that returns one clause after evaluating alternatives in some search-space of clauses, all of which are in $\mathcal{L}$.

3 Dynamic Statistics

Knuth describes two different types of statistics obtainable from the execution of programs by a compiler. These broadly relate to what the compiler is doing during program execution and why it is doing it (that is, which constructs in the programs are responsible for the main activities of the compiler). These statistics may then suggest optimisations that alleviate obvious execution bottlenecks. Here is one way to adopt this approach within ILP: sample from the space of ILP-system/ILP-datasets pairs. For each such pair, obtain dynamic statistics of the form just described. An average computed over all such profiles may then lead to a consideration of useful optimisations.

The principal practical difficulty in adopting the approach just outlined is the nature of current ILP implementations. These are encoded in a variety of different programming languages, each employing quite different constructs. As a first step therefore, we adopt the following more pragmatic approach: select a single ILP system (preferably one which is as general-purpose as possible) written in Prolog; execute the ILP system using a standard Prolog engine on one or more datasets; and examine dynamic profiles using the tools provided by the Prolog engine. There are two principal advantages in this. First, by selecting an ILP system written in Prolog, the same profiling tools can be applied to the execution of the hypothesis as well as the learning program itself[2]. Second, modern compilers or interpreters for the Prolog language appear to have converged to relatively uniform constructs and have good tools for constructing dynamic profiles.

Here we have selected the ILP system Aleph[3]. This is a Prolog program that can emulate a number of different ILP methods. We use Aleph to execute the simple clause-by-clause search mentioned earlier, using a search function that performs a restricted form of breadth-first branch-and-bound search (the restrictions ensure that the search only examines a finite number of alternatives). Further, every clause in the search is scored using an evaluation function that is simply the difference in the number of positive and negative examples derived by the clause (in conjunction with the background knowledge and clauses found earlier). For each clause in the search, this requires the search function to check derivability over all the examples. Aleph uses YAP[4], a high-performance Prolog compiler developed at The Universidade do Porto. To ensure derivability checks terminate, we use a special feature provided within YAP that allows depth-bounded evaluation. We shall use two tools that provide the kind of dynamic information described by Knuth. The first tool is the program `gprof`, an Unix tool that can be used to describes how the YAP Prolog compiler divides execution time. The second tool is a high-level profiler included in YAP. This profiler counts calls to procedures made by the user program (here, the ILP system Aleph).

[2] We are indebted to a referee of this paper for pointing this out.
[3] `http://web.comlab.ox.ac.uk/oucl/research/areas/machlearn/Aleph/`
[4] `http://www.ncc.up.pt/~vsc/Yap/`

We are still left with a choice of datasets. As a first step, we obtain an execution-profile from a single non-trivial application. We have elected to use the data concerning the prediction of carcinogenicity tests on rodents [23]. This dataset has a number of attractive features: it is an important practical problem; the background knowledge consists of large numbers of non-determinate predicate definitions (these can be a source of inefficiency when checking the derivability of an example); and our previous experience suggests that a fairly large search space needs to be examined to obtain a good clause. While obtaining dynamic statistics using a single ILP system and a single dataset is a significant compromise on the ideal presented at the outset of this section, the results are nevertheless interesting. Figure 2 summarises the information obtained from `gprof`.

Function	Involved in	Time (s)
`absmi`	WAM emulator	26110
`p_flags`	Meta-calls	840
`PredProp`	Meta-calls	830
`p_execute0`	Meta-calls	740
`a_eq`	Meta-calls	690
`a_le`	Meta-calls	450
`p_pred_goal`	Meta-calls	440
`FloatOfTerm`	Miscellaneous	1120
`CallProlog`	Miscellaneous	710
`GetExpProp`	Miscellaneous	630
`IUnify_complex`	Miscellaneous	460
`Eval`	Arithmetic expression evaluation	2040
`i_recorded`	Internal database access	440

Fig. 2. Prolog compiler activities when the ILP system Aleph analyses carcinogenesis outcome data. The profile is a summary of execution times (rounded up to the nearest 10s) reported by the `gprof` program accompanying the records the activities of the YAP compiler.

In Fig. 2 the proportion of times for the main activities are approximately: 74% for WAM (Warren Abstract Machine) emulation; 11% on "meta-calls" (explained shortly); 8% on miscellaneous activities; 6% on arithmetic evaluation; and 1% on internal database accesses. The high proportion of time allocated to WAM emulation simply means that most of the time is not spent in built-ins written in `C`, but in either use-code or built-ins written in Prolog. This is not surprising. What is unexpected is the high proportion of time spent on executing meta-calls. These are part of the compiler implementation of the Prolog built-in predicate `call/1`. Frequency counts obtained from the high-level YAP profiler (not shown here due to space restrictions) clearly indicate that the principal source of meta-call activity is the check for example derivability. For each clause of the form $Head : -Body$, this involves: (a) matching $Head$ against an example (positive or negative); and (b) a meta-call to $Body$. Typically, $Body$ is a conjunction of goals and (b) results in queries of the form $call((G_1, ..., G_n))$. Calling such a conjunction of goals is implemented in YAP as the following:

```
call((X,Y)) :- !, meta_call(X), call(Y).
call(X) :- meta_call(X).
```

Meta-calls are expensive and repeated failures, backtracks, and re-trials results in repeated re-invocation of this code. There are two things we could to alleviate this: (1) reduce the conjunction of goals to as few as possible; and (2) detect situations where any amount of backtracking over previous goals would not change the result of the computation of successive goals (the two sets of goals are *independent*). These two observations form the basis of the transformations described next.

4 Two Transformations to Reduce Query Costs

4.1 Transformation 1: Eliminate Redundant Literals

We are concerned here with clauses of the form $Head : -Body$ where $Body$ is conjunction of goals $G_1, ..., G_n$ (in the Prolog sense, or literals in the more traditional logical sense. Each of the G_i are calls to user-defined or built-in predicates). From the previous section it is evident that there is at least one meta-call for each element of the conjunction. In general, let N be an upper-bound on the number of solutions to any of the G_i. It follows straightforwardly that in the worst-case, the number of meta-calls is $O(N^n)$. Lowering N requires a re-appraisal of predicate definitions used by clauses (either by re-encoding of some predicates, or by removing them entirely). We do not pursue this here, and concentrate instead on lowering the value of n.

A straightforward reduction in the number of goals being tested (thus lowering n) is achieved by eliminating obviously redundant ones. A simple logical check for this during the execution of the steps described in Fig. 1 is as follows. On iteration i of the procedure in Fig. 1, let P denote $B \cup H_{i-1}$. Let C be a clause being examined in a search. Then, a literal l in C is redundant iff $P \cup \{C\} \equiv P \cup \{C'\}$ where $C' = C - \{l\}$. From this, it is easy to show that l is redundant iff $P \cup \{C\} \models \{C'\}$. This constraint, while providing an exact test for literal-redundancy, is expensive to implement (with a resolution-based theorem prover like that in YAP, it requires checking for the proof of inconsistency from the set $P \cup \{C\} \cup \{\neg C'\}$). Instead, we will employ a simpler test based on the subsumption relation [18]. Recall that C subsumes C' iff there is some substitution θ such that $C\theta \subseteq C'$ and if C subsumes C' then $\{C\} \models \{C'\}$. Clearly therefore if C subsumes C' then $P \cup \{C\} \models \{C'\}$ and l is redundant (the reverse is not true, and hence we call it a "weak" test for redundancy).

The clausal-subsumption test is relatively inexpensive to perform and can result in a reduction of goals being tested. While there is nothing profound about this test, it is not clear from descriptions of existing ILP systems whether they actually employ it during the search[5]. In general, while the efficacy of the transformation will depend on the enumeration operator employed by the search function (that is, whether it is prone to introduction of redundant literals), we expect it to be more useful when the background predicates are non-determinate.

4.2 Transformation 2: Exploit Goal Independence

We observe that when executing a conjunction of goals $G_1, ..., G_n$, failure of a goal G_i will result in attempting to generate more solutions for goals earlier in the sequence. Sometimes this effort is useless, as these solutions would not alter the computation of G_i. There is thus a notion of *goal independence* that we can exploit.

In pure logic programs, goals depend on each other because they share variables. Given a function $vars(T)$ that returns all unbound terms in the term T, two goals G_i and G_j are said to *share*, that is the relation $Shares(G_i, G_j)$ holds, when:

$$i = j \ \vee \ vars(G_i) \cap vars(G_j) \neq \emptyset$$

The definition ensures that the relation $Shares$ is reflexive. It is also symmetric and transitive, as all the $=, \cap, \neq, \vee$ relations are both symmetric and transitive. Therefore, $Shares$ is an equivalence relation. Given a set of goals $\{G_1, \ldots, G_n\}$, we shall name the equivalence classes established by $Shares$ as $I_1, \ldots, I_m$.

Intuitively, we say that a goal G_i in a class I_k is independent of a goal G_j on a class I_l, $k \neq l$, that is, that the computation for G_j can never bind a variable in G_i and therefore that results for G_j can never influence results for G_i in a clause. More precisely, let the conjunction of goals $G = G_1, ..., G_n$ be represented as the set $\{G_1, \ldots, G_n\}$. We claim that (i) if the sets $I_1, \ldots, I_m$ are satisfiable, then so is G; and that (ii) if one of the sets $I_1, \ldots, I_m$ is not satisfiable, G is not satisfiable. Therefore, to prove that G is satisfiable it is sufficient to prove that $I_1, \ldots, I_m$ are satisfiable.

To prove (i) we first notice that if the $I_1, \ldots, I_m$ are satisfiable we must be able to find a solution for each one, say $\sigma_1, \ldots, \sigma_m$. Moreover, from the $Shares$ relation we know that if a variable appears in a σ_k it cannot appear in a $\sigma_l, l \neq k$. We thus can compose $\sigma_1, \ldots, \sigma_m$ and still obtain a valid substitution $\sigma = \sigma_1, \ldots, \sigma_m$.

Next, consider a goal G_i. The goal must belong to an equivalence class for $Shares$, that is $\exists k, G_i \in I_k$. Without loss of generality, we can reorder this sets so that $k = 1$. We next show by finite induction on $l \geq m$ that $G_i\sigma = G_i\sigma_1$, and therefore that $G_k\sigma$ is satisfiable:

[5] Programs like WARMR [6] perform extensive checks to ensure that clauses subsumed by others that have previously been shown to be "useless" are not examined further. This is more elaborate than the proposal here, which simply checks for redundant literals within a clause.

- $G_i\sigma_1$ is of course $G_i\sigma_1$, and,
- If $G_i\sigma_1 \ldots \sigma_{l-1} = G_i\sigma_1$, then $G_i\sigma_1 \ldots \sigma_{l-1}\sigma_l = G_i\sigma_1\sigma_l$. But, none of the right-hand applies to a variable in $G_i\sigma_l$ so we have $G_i\sigma_1 \ldots \sigma_l = G_i\sigma_1$.

Thus, for every goal G_i, $G_i\sigma$ and so G_i are satisfiable. Therefore, all goals G_i are satisfiable, hence the conjunction G is satisfiable. To prove (ii) it is sufficient to notice that if a subset of a conjunction is not satisfiable, then the conjunction may not be satisfied.

Checking whether goals are independent is expensive. Ideally, we would like to do so only once when we compile a newly induced clause, not for every time we run the clause against an example. This means we will need to approximate the $Shares$ predicate by more general predicates. One solution is to replace $vars(G_i)$ and $vars(G_j)$ by $\alpha(vars(G_i))$ and $\alpha(vars(G_j))$, where α is such that $V \subset \alpha(V)$. We thus ensure the approximation relation or *abstraction* will also be an equivalence relation and will have a set of equivalence classes I's. Moreover, from the definition of α it follows that if $G_i \in I_l \wedge G_j \in I_l$, then there is an I'_m for the abstraction such that $G_i \in I'_m \wedge G_j \in I'_m$. Intuitively, the point is that this abstraction will preserve dependence, although independent goals may not be recognised.

We will perform the transformation while processing a new clause of the form $Head : -G_1, \ldots, G_n$. Our goal is to find the (approximated) equivalence classes in the $G_1, \ldots, G_n$ and generate code that executes them independently.

Our initial observation is is that if at compile-time variables in literal G form the set $V(G)$, then at run time we will have $vars(G) = vars'(V(G))$, where $vars'$ domain is a set of terms instead of on single term. For the sake of simplicity we next abuse terminology somewhat and instead of writing our use of the abstraction function as $\alpha(vars(G))$ we will just use an $\alpha(V(G))$. Next, we base our first abstraction on the idea that at compile-time variables may be in one of two types:

- *Existentially quantified* variables are variables that appear in at least one of the literals G_i but that do not appear in $Head$. For an existentially quantified variable E we will name the set of run-time values it can take as $\mathcal{E} = \{E', E'', E''', \ldots\}$. We know beforehand that for two different variables E_1 and E_2, $\mathcal{E}_1 \neq \mathcal{E}_2$.
- All other variables are assumed to be *universally quantified*. Universally quantified variables appear in the head of clause, and it is conservative to assume that at run-time they were bound to each other before the body of the clause was called.

  We name the set of run-time variables that may appear in a (compile-time) universally quantified variable U_i as $\mathcal{U}_i$. We do not know whether two universally quantified variables U_i and U_j are bound together, so we must assume that $\mathcal{U}_i \cap \mathcal{U}_j \neq \emptyset$. On the other hand, we do know that a new copy of all existentially quantified variables is created when we enter the body of a clause, so $\mathcal{U}_i \cap \mathcal{E}_j = \emptyset$. We name the union of all sets $\mathcal{U}_i$ as $\mathcal{U}$.

Our first abstraction α' summarises the previous observations:

- $\alpha'(\emptyset) = \emptyset$;
- $\alpha'(\{E_i\}) = \mathcal{E}_i$;
- $\alpha'(\{U_i\}) = \mathcal{U}$; and
- $\alpha'(x \cup y) = \alpha'(x) \cup \alpha'(y)$.

Unfortunately, this first abstraction is too general: all goals that include an universally quantified variable or that share variables with one such goal will be considered dependent. Better results stem from adding a new category of compile-time variables:

- Ground (or closed) variables are universally quantified variables that are known to be bound at run-time to ground (or closed) terms. This information may stem from user declarations or from other analysis. Clearly, the set of run-time variables for a ground variable G_i is $\emptyset$.

Our next abstraction α uses the class G:

- $\alpha(\emptyset) = \emptyset$;
- $\alpha(\{E_i\}) = \mathcal{E}_i$;
- $\alpha(\{U_i\}) = \mathcal{U}$;
- $\alpha(\{G_i\}) = \emptyset$; and
- $\alpha(x \cup y) = \alpha(x) \cup \alpha(y)$.

This is sufficient for our purposes (better approximations may be obtainable from the literature on independent and-parallelism [3]). To take advantage of this transformation it is sufficient to implement the following algorithm:

1. Given the original clause:

$$H\texttt{:-}\ G_1,\quad \ldots,\quad G_i,\quad \ldots,\quad G_n.$$

classify all variables in $G_1, \ldots, G_n$ and calculate the equivalence classes for the approximated sharing relation.

2. Number the literals according to the equivalence class they belong to:

$$H\texttt{:-}\ G_{1j},\quad \ldots,\quad G_{ik},\quad \ldots,\quad G_{nl}.$$

3. Reorder the literals in the clause according to the class they belong to:

$$H\texttt{:-}\ G_{a1},\quad \ldots,\quad G_{b1}, \ldots,\quad G_{cm},\quad \ldots,\quad G_{dm}.$$

The transformation does not introduce or remove goals, so according to the switching lemma it is correct for pure programs.

4. We are interested in any solution, if one exists. Thus we need to compute every class once and if a class has no solution, the computation should wholly fail. The following program transformation uses `cut` to guarantee such a computation:

$$H\texttt{:-}\ G_{a1},\quad \ldots,\quad G_{b1},\ !\ , \ldots,\ !\ ,\quad G_{cm},\quad \ldots,\quad G_{dm}.$$

As with the previous transformation, we expect goal independence to be useful when the predicates involved are non-determinate.

5 Empirical Evaluation

The transformations of the previous section are evaluated on four real-world tasks previously examined by ILP systems, namely biochemical problems of predicting mutagenesis and carcinogenesis; an engineering problem of deciding on optimal mesh partitions; and a natural language task of classifying word-tokens (or "tagging" the tokens) in sentences taken from a standard corpus. For each task we compare the ratio of times for theory construction obtained with and without the transformations. The tasks selected are such that two of them (the biochemical ones) consist of highly non-determinate background predicates and the others, of largely determinate predicates. Given that the transformations we have proposed are better suited to problems with non-determinate predicates, our prior expectation is thus:

> Benefits, if any, from the transformations will be observed in the biochemical tasks.

5.1 Domains and Data

The tasks selected are well-known within the ILP literature. Space restrictions prevent us from presenting the problems here. Summary descriptions can be found in [21] and [22]. For complete details, we refer the reader to: [10, 26] (mutagenesis); [23] (carcinogenesis); [7] (mesh); and [4] (tag). All data used here are available electronically by contacting the second author.

5.2 Algorithms and Machines

All experiments use the ILP program Aleph (Version 2) and the Prolog compiler YAP (Version 4.3.1). The current version of Aleph is available at: `http://web.comlab.ox.ac.uk/oucl/research/areas/ machlearn/Aleph`, and that of YAP at: `http://www.ncc.up.pt/~vsc/Yap/`.

In the experiments we distinguish between the following: (1) Aleph, to denote Aleph performing a restricted breadth-first branch-and-bound search, without the transformations proposed here, and with an evaluation function that assigns a clause a score that is the difference between the positive and negative examples derivable by using the clause. The number of clauses explored in any search is limited to 100,000; (2) Aleph+R, to denote Aleph with the first transformation incorporated (redundancy check); and (3) Aleph+R+I, to denote Aleph with both transformations incorporated (redundancy check, followed by independence check).

Note that we do not explore the following options: (a) Aleph with independence check only; and (b) Aleph with independence check followed by the redundancy check. Both these are of less interest, as the presence of redundant literals is always undesirable and is best corrected first. All experiments were performed on a SPARC Ultra 30 equipped with a 300MHz processor and 256 megabytes of random access memory.

5.3 Method

We adopt the following method:

For each task (Mutagenesis, Carcinogenesis, Mesh, and Tag)

- Restrict the hypothesis language $\mathcal{L}$ to clauses with at most $3, 4, 5, 6$ literals.
- For each instance of $\mathcal{L}$, construct theories consistent with the data using Aleph, Aleph+R, and Aleph+R+I. Record the time taken to construct the theory in each case ($T_{Aleph}, T_{Aleph+R}, T_{Aleph+R+I}$).
- Estimate the transformation efficacies from ratios of times recorded.

The following details are relevant: (1) Increasing the number of literals allowed in clauses usually increases the size of the search space explored. We are interested in examining the effect of the transformations as this size increases; (2) In each case, we obtain the time taken by the ILP algorithm to construct a theory that is consistent with the data. We recognise of course that in practice, theories that are to some degree inconsistent with the data may be acceptable. We are avoiding these here only to prevent confounding the study with secondary effects that may arise from a comparison of theories with incomparable accuracies; and (3) In every case, the theories constructed by Aleph, Aleph+R, and Aleph+R+I should be identical. This follows from the fact that the transformations proposed are admissible. This has also been verified empirically with the theories obtained.

5.4 Results and Discussion

Figure 3 tabulates the ratio of times for theory construction using the method just described. In the figure, ratios less than 1 indicate an improvement over Aleph without any transformations (percentage gains or losses from a ratio r is simply $|r - 1| \times 100$). The principal details in Fig. 3 are these: (1) The transformations are effective for large clause lengths on the biochemical tasks, and ineffective on the mesh-design and tagging tasks; and (2) When effective, the transformations work better when the sizes of the space to be searched increases.

The ratios reported in Fig. 3 do not account for the fact that in practice, the transformations are applied in sequence. It is instructive to consider the sequential changes in times for theory construction. This is shown in Fig. 4. As before, percentage gains or losses from a ratio r is $|r - 1| \times 100$.

These results appear to confirm our prior expectation, namely that benefits, if any, will manifest themselves in the biochemical tasks. They also provide evidence for some other observations:

- If theories required are known to consist fairly short clauses (less than 4 literals), then there appears to be little to gain from these transformations.
- Predicates used by mesh and tag problems are almost entirely functional and the transformations act as overheads that provide no benefit. On the other hand, the greater the non-determinacy of background predicates, the greater appear to be the gains. This is illustrated by the difference in entries between

Max. Clause Length	Time Ratios							
	MUT		CANC		MESH		TAG	
	(A)	(B)	(A)	(B)	(A)	(B)	(A)	(B)
3	1.03	1.17	1.05	1.13	0.98	1.02	1.15	1.03
4	1.00	1.16	0.93	0.92	1.00	1.07	1.19	1.01
5	0.81	0.86	0.51	0.49	1.00	1.09	1.17	1.14
6	0.77	0.77	0.25	0.21	1.02	1.13	1.24	1.31

Fig. 3. Ratio of times for theory construction. All ratios are against the time for theory construction using Aleph without transformations. MUT refers to the mutagenesis task, CANC to carcinogenesis, MESH to mesh-design, and TAG to tagging. The ratio (A) is with the check for redundancy (that is, it refers to $T_{Aleph+R}/T_{Aleph}$). The ratio (B) is with the check for redundancy and goal independence (that is, it refers to $T_{Aleph+R+I}/T_{Aleph}$).

Max. Clause Length	Time Ratios							
	MUT		CANC		MESH		TAG	
	(A)	(B)	(A)	(B)	(A)	(B)	(A)	(B)
3	1.03	1.14	1.05	1.08	0.98	1.04	1.15	0.90
4	1.00	1.16	0.93	0.98	1.00	1.06	1.19	0.85
5	0.81	1.05	0.51	0.95	1.00	1.08	1.17	0.97
6	0.77	1.00	0.25	0.82	1.02	1.10	1.24	1.05

Fig. 4. Effect of applying the transformations in sequence. The ratio (A) refers to $T_{Aleph+R}/T_{Aleph}$ and is thus identical to the column (A) in Fig. 3. The ratio B refers to $T_{Aleph+R+I}/T_{Aleph+R}$ and measures the effect of performing the independence check *after* the check for redundancy. Thus the entries $0.81, 1.05$ under MUT (for length = 5) are to be interpreted as follows. The redundancy check resulted in a 19% speed-up. Adding the independence check resulted in a 5% slow-down after this.

the two biochemical tasks. The non-determinacy of background predicates for the carcinogenesis task is significantly greater than those for mutagenesis. For example, the `atm/5` appears in both problems, with the same meaning: given a compound name, it returns a description of the atoms (in a chemical sense) in that compound. In carcinogenesis, for a given compound, this predicate can succeed up to 214 times. In contrast, this number is just 40 for mutagenesis. As greater non-determinacy usually increases the theorem-proving effort required—especially when the lengths of clauses is high—benefits from the transformations proposed is accentuated correspondingly.
– The entries under column (B) in Fig. 4 suggest that as clause length increases, the independence check appears to be increasingly effective for the non-determinate domains and increasingly ineffective for the determinate ones. The former trend persists in further experiments: systematically increasing clause lengths to 10 see the entries in column (B) decrease to 0.97 (MUT) and 0.46 (CANC). No such consistent picture emerges with the determinate domains (that is, increasing clause lengths does not lead to a steady increase in the corresponding ratio).

These observations lead us to the following prescription:

– If background predicates are largely functional, or theories are required to contain only very short clauses (4 literals or less), then do not use these transformations;
– If background predicates are non-determinate and theories require clauses of moderate length, then only use the transformation that eliminates redundant literals;
– If background predicates are non-determinate and theories require long clauses, then use both transformations.

6 Concluding Remarks

As ILP systems move from the realm of research to one of technology, implementation issues and concerns of efficiency become increasingly important. Here we have described two simple clause-transformation techniques that are directed towards reducing the theorem-proving effort that is at the heart of many ILP systems. Empirical evaluation suggests that the transformations can provide efficiency gains for problems that require complex theories that use highly non-determinate background predicates. It is of interest to see if these results are replicated on other datasets with similar characteristics, and with other ILP systems that use possibly different enumeration strategies.

The approach adopted in this paper is in the long tradition source-to-source program transformations: changes at the source-level that can improve efficiency without altering correctness [13]. While the two suggestions here by no means exhaust the transformations of this type, it is only one side of the story. Efficiency gains are as much to be made by tailoring compilers of logic programs to account for what ILP systems do. In concluding his empirical study, Knuth states: "...the design of compilers should be influenced by what programmers want to do. An alternative point of view is that programmers should be strongly influenced by what their compilers do; a compiler writer in his infinite wisdom may in fact know what is really good for the programmer and would like to steer him towards a proper course. This viewpoint has some merit, although it has often been carried to extremes in which programmers have to work harder and make unnatural constructions just so the compiler writer has an easier job." Our experience suggests that compiler writers for logic programs do not subscribe to the alternative point of view.

Acknowledgments

This work has been partly supported by Fundação da Ciência e Tecnologia under the project Dolphin (PRAXIS/2 /2.1/TIT/1577/95), and by the PROTEM5 CNPq-NSF Collaboration Project $CLoP^n$. A.S is supported by a Nuffield Trust Fellowship at Green College, Oxford.

References

1. R. Benigni. (Q)SAR prediction of chemical carcinogenicity and the biological side of the structure activity relationship. In *Proceedings of The Eighth International Workshop on QSARs in the Environmental Sciences*, 1998. Held in Baltimore, May 16–20, 1998.
2. I. Bratko and M.Grobelnik. Inductive learning applied to program construction and verification. In *Third International Workshop on Inductive Logic Programming*, pages 279–292, 1993. Available as Technical Report IJS-DP-6707, J. Stefan Inst., Ljubljana, Slovenia.
3. M. Codish, M. Bruynooghe, M. G. de la Banda, and M. Hermenegildo. Exploiting goal independence in the analysis of logic programs. *Journal of Logic Programming*, 32(3), 1997.
4. J. Cussens. Part-of-Speech Tagging Using Progol. In S. Džeroski and N. Lavrač, editors, *Proceedings of the Seventh International Workshop on ILP*, volume 1297 of *LNAI*, pages 93–108. Springer, 1997.
5. A.K. Debnath, R.L Lopez de Compadre, G. Debnath, A.J. Schusterman, and C. Hansch. Structure-Activity Relationship of Mutagenic Aromatic and Heteroaromatic Nitro compounds. Correlation with molecular orbital energies and hydrophobicity. *Journal of Medicinal Chemistry*, 34(2):786 – 797, 1991.
6. L. Dehaspe, H. Toivonen, and R.D. King. Finding frequent substructures in chemical compounds. In *Proceedings of the Fourth International Conference on Knowledge Discovery and Data Mining (KDD-98)*, pages 30–36. AAAI Press, 1998.
7. B. Dolsak and S. Muggleton. The application of Inductive Logic Programming to finite element mesh design. In S. Muggleton, editor, *Inductive Logic Programming*, pages 453–472. Academic Press, London, 1992.
8. S. Dzeroski, L. Dehaspe, B. Ruck, and W. Walley. Classification of river water quality data using machine learning. In *Proceedings of the Fifth International Conference on the Development and Application of Computer Techniques Environmental Studies*, 1994.

9. C. Feng. Inducing temporal fault dignostic rules from a qualitative model. In S. Muggleton, editor, *Inductive Logic Programming*, pages 473–486. Academic Press, London, 1992.
10. R.D. King, S.H. Muggleton, A. Srinivasan, and M.J.E. Sternberg. Structure-activity relationships derived by machine learning: The use of atoms and their bond connectivities to predict mutagenicity by inductive logic programming. *Proc. of the National Academy of Sciences*, 93:438–442, 1996.
11. R.D. King, S.H. Muggleton, and M.J.E. Sternberg. Drug design by machine learning: The use of inductive logic programming to model the structure-activity relationships of trimethoprim analogues binding to dihydrofolate reductase. *Proc. of the National Academy of Sciences*, 89(23):11322–11326, 1992.
12. D. E. Knuth. An Empirical Study of FORTRAN Programs. *Software—Practice and Experience*, 1:105–133, 1971.
13. D. B. Loveman. Program improvement by source-to-source transformation. *JACM*, 24(1):121–145, 1977.
14. S. Muggleton. Inductive Logic Programming: derivations, successes and shortcomings. *SIGART Bulletin*, 5(1):5–11, 1994.
15. S. Muggleton. Inverse Entailment and Progol. *New Gen. Comput.*, 13:245–286, 1995.
16. S. Muggleton, R. King, and M. Sternberg. Predicting protein secondary structure using inductive logic programming. *Protein Engineering*, 5:647–657, 1992.
17. S.H. Muggleton and C. Feng. Efficient induction of logic programs. In *Proceedings of the First Conference on Algorithmic Learning Theory*, Tokyo, 1990. Ohmsha.
18. S. Nienhuys-Cheng and R. de Wolf. *Foundations of Inductive Logic Programming*. Springer, Berlin, 1997.
19. J.R. Quinlan. Learning logical definitions from relations. *Machine Learning*, 5:239–266, 1990.
20. M. Sebag and C. Rouveirol. Tractable Induction and Classification in First-Order Logic via Stochastic Matching. In *Proceedings of the Fifteenth International Conference on Artificial Intelligence (IJCAI-97)*. Morgan Kaufmann, Los Angeles, CA, 1997.
21. A. Srinivasan. A study of two probabilistic methods for searching large spaces with ILP. *Data Mining and Knowledge Discovery (under review)*, 1999.
22. A. Srinivasan. A study of two sampling methods for analysing large datasets with ILP. *Data Mining and Knowledge Discovery*, 3(1):95–123, 1999.
23. A. Srinivasan and R.D. King. Carcinogenesis predictions using ILP. In N. Lavrac and S. Dzeroski, editors, *Proceedings of the Seventh International Workshop on Inductive Logic Programming (ILP97)*, volume 1297 of *LNAI*, pages 273–287, Berlin, 1997. Springer. A version also in Intelligent Data Analysis in Medicine, Kluwer.
24. A. Srinivasan, R.D. King, and D.W. Bristol. An assessment of submissions made to the Predictive Toxicology Evaluation Challenge. In *Proceedings of the Sixteenth International Conference on Artificial Intelligence (IJCAI-99)*. Morgan Kaufmann, Los Angeles, CA, 1999.
25. A. Srinivasan, R.D. King, S.H. Muggleton, and M.J.E. Sternberg. The Predictive Toxicology Evaluation Challenge. In *Proceedings of the Fifteenth International Conference on Artificial Intelligence (IJCAI-97)*. Morgan Kaufmann, Los Angeles, CA, 1997.
26. A. Srinivasan, S.H. Muggleton, R.D. King, and M.J.E. Sternberg. Theories for mutagenicity: a study of first-order and feature based induction. *Artificial Intelligence*, 85:277–299, 1996.

27. J. Zelle and R. Mooney. Learning semantic grammars with constructive inductive logic programming. In *Proceedings of the Eleventh National Conference on Artificial Intelligence*, pages 817–822. Morgan Kaufmann, 1993.

A Prolog code for the transformations

The following Prolog code assumes the definitions of predicates for manipulating ordered sets. These are available within the library modules distributed with YAP.

```
% Transformation 1
remove_redundant((Head:-Body),(Head1:-Body1)):-
        goals_to_list((Head,Body),ClauseL),
        remove_subsumed(ClauseL,[Head1|Body1L]),
        (Body1L = [] -> Body1 = true; list_to_goals(Body1L,Body1)).

% Transformation 2
reorder_clause((Head:-Body), Clause) :-
        term_variables(Head,LHead),
        number_goals_and_get_vars(Body,LHead,1,_,[],Conj),
        calculate_independent_sets(Conj,[],BSets),
        compile_clause(BSets,Head,Clause).

number_goals_and_get_vars((G,Body),LHead,I0,IF,L0,[g(I0,LVF,NG)|LGs]) :- !,
        I is I0+1,
        get_goal_vars(G,LHead,LVF,NG),
        number_goals_and_get_vars(Body,LHead,I,IF,L0,LGs).
number_goals_and_get_vars(G,LHead,I,I,L0,[g(I,LVF,NG)|L0]) :-
        get_goal_vars(G,LHead,LVF,NG).

get_goal_vars(G,LHead,LVF,G) :-
        term_variables(G,LV0),
        sort(LV0,LVI),
        ord_subtract(LVI,LHead,LVF).

calculate_independent_sets([],BSets,BSets).
calculate_independent_sets([G|Ls],BSets0,BSetsF) :-
        add_goal_to_set(G,BSets0,BSetsI),
        calculate_independent_sets(Ls,BSetsI,BSetsF).

add_goal_to_set(g(I,LV,G),Sets0,SetsF) :-
        add_to_sets(Sets0,LV,[g(I,LV,G)],SetsF).

add_to_sets([],LV,Gs,[[LV|Gs]]).
add_to_sets([[LV|Gs]|Sets0],LVC,GsC,[[LV|Gs]|SetsF]) :-
        ord_disjoint(LV,LVC), !,
        add_to_sets(Sets0,LVC,GsC,SetsF).
add_to_sets([[LV|Gs]|Sets0],LVC,GsC,SetsF) :-
```

```
        ord_union(LV,LVC,LVN),
        join_goals(Gs,GsC,GsN),
        add_to_sets(Sets0,LVN,GsN,SetsF).

join_goals([],L,L):- !.
join_goals(L,[],L):- !.
join_goals([g(I1,VL1,G1)|T],[g(I2,VL2,G2)|T2],Z) :-
        I1 < I2, !,
        Z = [g(I1,VL1,G1)|TN],
        join_goals(T,[g(I2,VL2,G2)|T2],TN).
join_goals([H|T],[g(I2,VL2,G2)|T2],Z) :-
        Z = [g(I2,VL2,G2)|TN],
        join_goals(T,[H|T2],TN).

compile_clause(Goals,Head,(Head:-Body)):-
        compile_clause2(Goals,Body).

compile_clause2([[_|B]], B1):-
        !,
        glist_to_goals(B,B1).
compile_clause2([[_|B]|Bs],(B1,!,NB)):-
        glist_to_goals(B,B1),
        compile_clause2(Bs,NB).

% remove literals subsumed in the body of a clause
remove_subsumed([Head|Lits],Lits1):-
        delete(Lit,Lits,Left),
        \+(\+(redundant(Lit,[Head|Lits],[Head|Left]))), !,
        remove_subsumed([Head|Left],Lits1).
remove_subsumed(L,L).

% determine if Lit is subsumed by a body literal
% this is not a very efficient implementation
redundant(Lit,Lits,[Head|Body]):-
        copy_term([Head|Body],Rest1),
        member(Lit1,Body),
        Lit = Lit1,
        subsumes(Lits,Rest1).

subsumes(Lits,Lits1):-
        \+(\+((numbervars(Lits,0,_),numbervars(Lits1,0,_),subset1(Lits,Lits1)))).

% General utilities
list_to_goals([Goal],Goal):- !.
list_to_goals([Goal|Goals],(Goal,Goals1)):- list_to_goals(Goals,Goals1).

glist_to_goals([g(_,_,Goal)],Goal):- !.
glist_to_goals([g(_,_,Goal)|Goals],(Goal,Goals1)):- glist_to_goals(Goals,Goals1).

goals_to_list((true,Goals),T):- !, goals_to_list(Goals,T).
```

```
goals_to_list((Goal,Goals),[Goal|T]):- !, goals_to_list(Goals,T).
goals_to_list(true,[]):- !.
goals_to_list(Goal,[Goal]).

subset1([],_).
subset1([Elem|Elems],S):- member1(Elem,S), !, subset1(Elems,S).

member1(H,[H|_]):- !.
member1(H,[_|T]):- member1(H,T).

member(H,[H|_]).
member(H,[_|T]):- member(H,T).

delete(H,[H|T],T).
delete(H,[H1|T],[H1|T1]):- delete(H,T,T1).
```