Journal of Machine Learning Research 14 (2013) 3649-3681 Submitted 4/12; Revised 7/13; Published 12/13

Efficient Program Synthesis Using Constraint Satisfaction in Inductive Logic Programming

John Ahlgren AHLGREN@EE.CITYU.EDU.HK
Shiu Yin Yuen KELVINY.EE@CITYU.EDU.HK
Department of Electronic Engineering
City University of Hong Kong
Hong Kong, China

Editor: Luc De Raedt

Abstract

We present NrSample, a framework for program synthesis in inductive logic programming. NrSample uses propositional logic constraints to exclude undesirable candidates from the search. This is achieved by representing constraints as propositional formulae and solving the associated constraint satisfaction problem. We present a variety of such constraints: pruning, input-output, functional (arithmetic), and variable splitting. NrSample is also capable of detecting search space exhaustion, leading to further speedups in clause induction and optimality. We benchmark NrSample against enumeration search (Aleph's default) and Progol's A^* search in the context of program synthesis. The results show that, on large program synthesis problems, NrSample induces between 1 and 1358 times faster than enumeration (236 times faster on average), always with similar or better accuracy. Compared to Progol A^*, NrSample is 18 times faster on average with similar or better accuracy except for two problems: one in which Progol A^* substantially sacrificed accuracy to induce faster, and one in which Progol A^* was a clear winner. Functional constraints provide a speedup of up to 53 times (21 times on average) with similar or better accuracy. We also benchmark using a few concept learning (non-program synthesis) problems. The results indicate that without strong constraints, the overhead of solving constraints is not compensated for.

Keywords: inductive logic programming, program synthesis, theory induction, constraint satisfaction, Boolean satisfiability problem

1. Introduction

Inductive logic programming (ILP) is a branch of machine learning that represents knowledge as first-order horn clauses. By using its expressive relational representation, it can overcome limitations inherent in propositional representations (Russell et al., 1996). ILP can be used for both concept learning and program synthesis, as the knowledge representation is at the same time declarative and procedural (Blackburn et al., 2006; Sterling and Shapiro, 1994).[1] In other words, induced solutions can both be regarded as human readable descriptions and as executable programs. Another advantage of ILP is its rigorous foundation in logic, making it suitable for theoretical analysis (Nienhuys-Cheng and de Wolf, 1997; Plotkin, 1970, 1971). A comprehensive survey of the ILP research field and its applications can be found in Muggleton et al. (2012).

1. In this paper, we use the term *concept learning* to refer to non-program synthesis problems, although program synthesis problems could also be considered concept learning problems.

State-of-the-art ILP systems such as Progol and Aleph use the technique of inverse entailment to induce theories (Muggleton, 1995; Srinivasan, 2001). Inverse entailment works by first constructing a bottom clause, from which all clauses that subsume it (or are supersets of it) become candidates. The search space hence becomes a lattice structure with a partial generality order, with the bodyless clause as top element and bottom clause as bottom element.

Mode declarations, as introduced in Muggleton (1995), may be used to constrain the search space further by specifying which variables are input and output, as well as requiring variables to be of a certain type. This input-output specification implicitly defines a logical constraint on the chaining of literals as the clause is computed from left to right. We describe mode declarations in Section 2.2.1. Mode declarations can be user provided or automatically constructed by analyzing the examples (McCreath and Sharma, 1995).

Inverse entailment is the method of constructing a lower bound on the search space. However, it does not force a certain ordering on the search. Thus various search strategies exist; some well known ones are Progol's A^* search (Muggleton, 1995), QG/GA (Muggleton and Tamaddoni-Nezhad, 2008), Aleph's enumeration (its default) (Srinivasan, 2001), and simulated annealing (Serrurier et al., 2004). What all these methods share in common is that the candidate generation mechanism employed is not adapted to avoid creation of candidates that violate the mode declarations. The mode declaration types are automatically handled by the bottom clause construction algorithm, but correct input-output chaining may be violated since candidates contain a subset of the bottom clause's literals. Such candidates are intended to be omitted by the syntactic bias a user provides using mode declarations, and should hence be considered redundant. One approach, taken by Aleph's enumeration, is to check input-output validity of the candidate before evaluation, but this still incurs the overhead of having to construct the candidate. Another approach, taken by algorithms using refinement operators, such as Progol's A^*, is to be indifferent about input-output violated candidates, evaluating all generated candidates. This is due to the fact that top-down (or bottom-up) searches may have to expand undesirable candidates in order to reach desirable ones. Both these approaches suffer from wasted computations as candidates in the search space violating the given mode declarations are still generated (and even evaluated).

In this paper, we present NrSample ("Non-redundant Sampling Algorithm"), a general constraint satisfaction framework for ILP that ensures only candidates that conform to mode declarations are generated. NrSample is implemented (along side other algorithms we use in our benchmarks) in our ILP system Atom.[2] This is to be contrasted with the aforementioned approaches, whereby a candidate is first generated and then tested for mode declaration correctness. We shall refer to candidates that are correct with respect to their mode declarations as *mode conformant*.

We achieve mode conformance by representing the input-output logic given by the mode declarations as propositional clauses (Russell et al., 1996). By solving the accompanying Boolean satisfiability problem (SAT), we obtain new candidates that by construction necessarily satisfy all constraints, and thus are mode conformant.

Moreover, other constraints naturally arise as a result of pruning the search space. Whenever a candidate is consistent (covers no negative examples), so are all specializations. Whenever a candidate is inconsistent (covers at least one negative example), so are all generalizations. As we will show, such pruning constraints are also easily represented as propositional clauses. More conventional approaches may use memorization to avoid evaluating candidates more than once, but

2. Source code available at `http://ahlgren.info/research/atom`, mirror available at `http://www.ee.cityu.edu.hk/~syyuen/Public/Code.html`.

the candidates may still be constructed multiple times. We shall refer to candidates that violate the constraints (mode declarations, pruning, or any other constraints) as *invalid*, and the rest as *valid*.

We represent our constraints as propositional clauses. Any SAT solver may then be used to solve the constraints and obtain a model representing a valid candidate. For efficiency, we use the Chaff algorithm (Moskewicz et al., 2001), which forms the basis of many state-of-the-art DPLL-based algorithms (Davis-Putnam-Logemann-Loveland algorithm) (Davis et al., 1962). Various selection strategies may be used within the SAT solver to guide the search into interesting regions early. By using a *complete* SAT solver (all DPLL-based algorithms are complete), any search strategy is also guaranteed to terminate as soon as all non-redundant candidates have been explored.

A survey and discussion of program synthesis in ILP can be found in Flener and Yılmaz (1999). Constraints have been used in ILP before, although not to constrain the bottom clause literals. Constraint logic programming with ILP (Sebag and Rouveirol, 1996) turns negative examples into constraints, primarily as a means of dealing with numerical constraints. Theta-subsumption with constraint satisfaction (Maloberti and Sebag, 2004) uses constraints to speed up theta-subsumption testing (during coverage). Condensed representations (De Raedt and Ramon, 2004) are used to deal with redundancies in frequent Datalog queries. In the context of theory induction, schemata may be used to guide the construction of logic programs (Flener et al., 2000). Our framework differs from all the above in that the constraints specify which of a bottom clause's literals must be (or may not be) present in any candidate. The constraints are thus propositional in nature, and relative to a computed bottom clause. Our approach attempts to minimize the amount of redundant candidates *constructed*.

This paper is organized as follows. First, we describe propositional constraints and candidate generation in Section 2. Next, Section 3 describes our search algorithm that uses these constraints. In Section 4, we perform benchmarks to test our hypothesis that NrSample can outperform generate-and-test methods, in particular when the search space is large. Finally, Section 5 concludes the paper.

2. Propositional Constraints and SAT Solving

Although our framework enables the use of arbitrary propositional constraints to define redundancies during an ILP search for candidates, we will present two common instances of such constraints.

Firstly, NrSample constrains candidate generation to only those that conform to the mode declarations. Secondly, it prunes too general or specific solutions after each evaluation: which depends on whether the candidate is consistent or inconsistent.

NrSample achieves non-redundancy by storing all the constraints as propositional formulae. Candidate solutions during a search contain a subset of the bottom clause's literals (for the subsumption order, we discuss variable splitting in Section 2.4). Hence, each candidate can be represented as a bit string where bit b_i signifies occurrence of body literal at position i or lack thereof. Seen from a slightly different perspective, we represent the occurrence or non-occurrence of a literal at position i in the bottom clause by a propositional variable b_i and $\neg b_i$, respectively. With respect to a bottom clause, there is thus a one-to-one correspondence between each Boolean assignment and candidate solution. The propositional constraints correspond to the syntactic bias induced by the user generated mode declarations and the search lattice pruning. The idea is that a solution is evaluated for coverage if and only if it corresponds to a propositional model for the constraints (a variable assignment that makes all constraints true). This enables us to invoke a SAT solver to

retrieve models for the constraints, which are then easily converted into candidate solutions. After each candidate is evaluated, pruning constraints related to generality order redundancies are added.

In the following sections, we describe how we create new constraints and retrieve models from the constraint database.

2.1 Clauses and Propositional Formulae

We start by defining the notion of a search space *candidate*.

Definition 1 *Let B be a definite clause (a clause with a head). A clause C is a* candidate from B *if and only if C's head is the same as B's head, and C's body is a subsequence of B's body.*

Here, B is intended to be the bottom clause, and C a clause that is created by possibly removing body literals of B. Note that B itself is a candidate from B. Usually, we are only interested in candidates from a specific bottom clause, in which case we omit the reference to B. Note also that the definition of subsequence forces the body literals of C to appear in the same order as those of B.

Example 1 *With bottom clause $B = h(X,Y) \leftarrow q_1(X,Z), q_2(Z,Y), q_3(X,Y,Z)$, the clause $C = h(X,Y) \leftarrow q_1(X,Z), q_3(X,Y,Z)$ is a candidate from B since they have the same head and $(q_1(X,Z), q_3(X,Y,Z))$ is a subsequence of $(q_1(X,Z), q_2(Z,Y), q_3(X,Y,Z))$. $D = h(X,Y) \leftarrow q_3(X,Y,Z), q_1(X,Z)$ is however not a candidate from B, since the sequence $(q_3(X,Y,Z), q_1(X,Z))$ is not a subsequence of $(q_1(X,Y), q_2(Y,Z), q_3(X,Y,Z))$.*

Definition 1 defines a search space lattice of candidates spanned by subset order.[3] The more general subsumption order can be explored using variable splitting, as discussed in Section 2.4.

To create a one-to-one correspondence between first-order horn clauses and propositional formulae with respect to a bottom clause, we represent each literal of the bottom clause from left to right as propositional variables $b_1, b_2, \ldots, b_n$, where n is the number of literals in the body of the bottom clause. Given a clause C which has some of the body literals in B, the propositional formula for C is then the conjunction of all propositional variables in B with positive sign if they occur in C and negative sign otherwise.

Definition 2 *Let C be a candidate from B, where B has n body literals. The propositional formula for C is $P_C^B = \bigwedge_{i=1}^{n} l_i$, where $l_i = b_i$ if C contains the ith literal in B, and $l_i = \neg b_i$ otherwise. We write P_C when there is no confusion about which bottom clause we are referring to.*

Note that P_C is a conjunction of *all* literals $b_1, \ldots, b_n$, where n is the number of body literals in B. In particular, non-occurrence of a literal has to be specified with its corresponding negative propositional literal. This way, each propositional formula has precisely one model, corresponding to the candidate itself.

Example 2 *Continuing from our previous example, $P_C^B = b_1 \wedge \neg b_2 \wedge b_3$ and $P_B^B = b_1 \wedge b_2 \wedge b_3$.*

The purpose of representing clauses as propositional formulae (with respect to some bottom clause) is that we can solve the formulae to acquire a model, which can then trivially be converted into a candidate.

3. Technically, by subsequence order, but as we do not consider re-orderings, there is no risk of confusion.

Definition 3 *Let B be a clause with n body literals, F^B a propositional formula containing a subset of the propositional variables $\{b_1,\ldots,b_n\}$, and M a model for F^B. The propositional formula (from B) generated by M is:*

$$P_M^B = \bigwedge_{i=1}^{n} l_i, \text{ where } l_i = \begin{cases} b_i & \text{if } M(b_i) = true \\ \neg b_i & \text{if } M(b_i) = false \end{cases}.$$

If M generates the propositional formula P_C, C is the candidate (from B) generated by M.

Example 3 *Let B be defined as in Example 1 and $F^B = b_1 \wedge (b_2 \vee b_3)$. Then $M = \{b_1 = true, b_2 = false, b_3 = true\}$ is a model for F^B. The propositional formula generated by M is $b_1 \wedge \neg b_2 \wedge b_3$. The candidate generated by M is $h(X,Y) \leftarrow q_1(X,Z), q_3(X,Y,Z)$.*

Usually, we are not only interested in what candidate a specific model generates, but rather, all candidates generated by all models of a propositional formula.

Definition 4 *Let B be a clause with n body literals and F^B a propositional formula. The propositional formulae generated by F^B are the propositional formulae generated by all models of F^B. The candidates (from B) generated by F^B are the candidates corresponding to those propositional formulae.*

Example 4 *Let B and F^B be as in the previous example. In all models of F^B, b_1 is true, and at least one of b_2, b_3 is true. This gives 3 models for F^B, and the propositional formulae generated by F^B are $b_1 \wedge b_2 \wedge \neg b_3$, $b_1 \wedge \neg b_2 \wedge b_3$, and $b_1 \wedge b_2 \wedge b_3$. The candidates generated by F^B are $h(X,Y) \leftarrow q_1(X,Z), q_2(Z,Y)$, $h(X,Y) \leftarrow q_1(X,Z), q_3(X,Y,Z)$, and $h(X,Y) \leftarrow q_1(X,Z), q_2(Z,Y), q_3(X,Y,Z)$, respectively. Intuitively, the formula F^B tells us that a candidate must have the first literal ($q_1(X,Z)$) and at least one of the two that follow it. By looking at all models for F^B, we can retrieve these candidates explicitly.*

Our constraints are represented as a propositional formula (more specifically, a conjunction of clauses, as we will see later). The constraints represent which candidates are allowed, so we start with the constraint *true* to allow any candidate from *B*. To retrieve an allowed (non-redundant) candidate, we compute a model for the constraints using our SAT solver. This model then generates our candidate solution as of Definition 3. The set of all allowed candidates are those generated by our constraints as of Definition 4.

Example 5 *As an example of how constraints work, assume we want an algorithm that never generates a previously evaluated candidate. For each candidate C, P_C is the corresponding propositional formula. Since our constraints specify the set of allowed candidates through its models, adding $\neg P_C$ will prevent candidate C (and only C) from being generated again.*

Now we show which constraints to add in order to prevent mode violations and redundancies that occur due to the generality order. The latter includes blocking out already visited candidates as in the example above, but prunes more of the search space.

2.2 Mode Declaration Constraints

Intuitively, mode declarations show where the inputs and outputs of a literal are located, and which types it takes. The bottom clause is then constructed using the mode declarations so that no literal is introduced until all its inputs have been instantiated by previous literals. Here we briefly explain mode declarations, referring the reader to Muggleton (1995) for a more general description.

2.2.1 MODE DECLARATIONS EXPLAINED

A mode declaration is a specification that limits the usage of variables (in the literals of clauses) to some specific type (for example, numbers or lists) and specify whether they will be used as input or output. Input variables need to be instantiated (computed) by a previous literal in the sequence of body literals, or by the head. Output variables occurring in body literals do not have any restrictions: they may or may not have been previously instantiated. Output variables in the head need to be instantiated by the body, or by an input variable to the head.

Mode declarations are used to restrict the number of literals in the bottom clause—by requiring input-instantiation—as well as providing computational speedup, by restricting attention to only the relevant data types. These aspects are being taken care of by the bottom clause construction algorithm. We introduce the syntax and semantics of mode declarations using an example of program synthesis.

Example 6 *Consider the mode declaration* $modeh(*, member(-constant, +list))$. *modeh refers to the head of a clause, as opposed to an atom used in the body. The first argument to modeh—where the asterisk is located—is where the maximum number of query answers is specified. This is known as the* recall. *The asterisk specifies that there is no upper limit (infinitely many), which is clearly the case for list membership, as there is no bound on the number of elements it may contain. When querying for the parents of a person, clearly the recall can be set to two (and for grandparents, to four). Recall does not interfere with our constraint framework: they can be used or ignored, and we will therefore make no further mention of them in this paper. The mode declaration declares a predicate member/2 (the* 2 *specifies the arity), and when it is used in the head of a clause, it outputs a constant (indicated by the minus sign) and requires a list as input (indicated by the plus sign). Thus we expect* $member(E, [3,5])$ *to be a valid query, since the list* $[3,5]$ *is used as input. On the other hand, the query* $member(3, L)$ *is invalid, since L is not instantiated.*

Now consider adding the mode declaration $modeb(*, +list = [-constant|-list])$. *This declares =/2 as a body literal (modeb) with a list on the left hand side as input, to obtain the head and tail of the list as outputs on the right hand side. Intuitively, this mode declaration introduces a predicate useful for splitting a list into its head and tail.*

In conjunction with the modeh declaration above, we may now generate the mode conformant clause $member(E, L) \leftarrow L = [E|T]$, *which is the proper base case definition of list membership. The occurrence of L in the head is as input (given by the modeh declaration), so L is already instantiated in the body literal as required. Also, E in the head is required to be output, a requirement fulfilled by the body literal as it instantiates E.*

An example of a clause that is not mode conformant is $member(E, L) \leftarrow X = [E|T]$, *since X is not an input type (it has not previously been instantiated). Another example of a non-conformant clause is* $member(E, L) \leftarrow L = [H|T]$, *this time because E is declared to be an output variable, but never instantiated (we obtain no meaningful answer to a query such as* $member(E, [2])$, *since E is never computed in the clause).*

Mode declarations can also be used to restrict the search itself, since candidates, by containing subsequences of the bottom clause's literals, may break an input-output chain. The next example illustrates this.

Example 7 *Assume we have computed bottom clause*

$$member(A,B) \leftarrow B = [C|D], member(A,D), member(C,B)$$

from an example. The mode declarations are

$$\begin{array}{l} modeh(*, member(+const, +clist)), \\ modeb(*, +clist = [-const|-clist]), and \\ modeb(*, member(+const, +clist)). \end{array}$$

In this case, A and B are both already instantiated in the head, as specified by the mode declarations (the plus signs).

The clause $member(A,B) \leftarrow member(A,D)$ *is a candidate, albeit not a mode conformant one, since D is never instantiated before appearing in the body literal (as required by the third mode declaration). We would need to include the bottom clause literal* $B = [C|D]$ *first, as it is the only literal that would instantiate D. Including both, we would then have the proper recursive clause for list membership:*

$$member(A,B) \leftarrow B = [C|D], member(A,D).$$

In this case we get a simple rule of the form "if the bottom clause's second body literal is included in a candidate, so must the first". In general we may have more than one possible instantiation, and literals may contain more than one input variable, making the rules more complex. In the next section, we work through the logic of mode declarations in detail.

2.2.2 MODE DECLARATIONS AS PROPOSITIONAL CLAUSES

A candidate is obtained from the bottom clause by taking a subsequence of the bottom clause's body literals and copying the head. As we have seen in Example 7, there is no guarantee that a randomly chosen candidate will respect the mode declarations. A common solution to this is to check for validity before evaluating a candidate. Our goal is to avoid the generation of unnecessary candidates in the first place.

For our purposes, there are two aspects of input-output chaining that affect the constraints:

1. Each body literal input must be instantiated from previous body literal outputs or from inputs to the head (inputs to the head are instantiated by the query).

2. The head outputs must be instantiated from head inputs or body literal outputs.

Definition 5 *Let C be a clause with n body literals. Let* I_i *and* O_i *be the set of input and output variables of the ith literal, respectively (as defined by the mode declarations). Denote the input and output variables of h by* I_h *and* O_h*, respectively. C is* input-instantiated *if and only if for all* $v \in I_i$*, we have* $v \in I_h$ *or* $v \in O_k$ *for some* $k < i$*. C is* output-instantiated *if and only if, for all* $v \in O_h$*, we have* $v \in I_h$ *or* $v \in O_k$ *for some k. C is* mode conformant *if and only if C is both input- and output-instantiated.*

Example 8 *With mode declarations* $modeh(*, h(+any, -any))$, $modeb(*, q_1(+any, -any))$ *and* $modeb(*, q_2(+any, -any))$, *the clause* $C = h(X,Z) \leftarrow q_1(X,Y), q_2(Y,X)$ *is input-instantiated, since* X *in* $q_1(X,Y)$ *grabs its input from the head input, and* Y *in* $q_2(Y,X)$ *grabs its input from the output of* $q_1(X,Y)$ *(which appears before* $q_2(Y,X)$*). It is however not output-instantiated, since the head output* Z *does not grab output from any output of the body literals (and is not in the head as input). Essentially, this means that in a query such as* $h(5,Ans)$, *the* 5 *will be "propagated" through all literals (*C *is input-instantiated), but our query variable Ans will not be bound (*C *is not output-instantiated). The clause* $D = h(X,Z) \leftarrow q_1(Y,Z)$ *is output-instantiated since* Z *in the head grabs output from* $q_1(Y,Z)$, *but not input-instantiated since* Y *in* $q_1(Y,Z)$ *is not instantiated. The clause* $E = h(X,Z) \leftarrow q_1(X,Y), q_2(Y,Z)$ *is both input- and output-instantiated, and hence mode conformant. Finally,* $F = h(X,Z) \leftarrow q_1(A,B)$ *is neither input- nor output-instantiated.*

Lemma 6 Given mode declarations, a bottom clause is always input-instantiated but not necessarily output-instantiated.

Proof Input-instantiation is a direct consequence of the way in which the bottom clause is constructed: we only add body literals when all their inputs have been instantiated by the head or previous body literals. To see that a bottom clause may not be output-instantiated, it is enough to consider a mode head declaration in which the output does not correspond to any body literal outputs: $modeh(*, h(+type1, -type2))$, $modeb(*, b(+type1, -type1))$. With type definitions $type1(a) \wedge type2(b)$, example $h(a,b)$ and background knowledge $b(a,a)$, we get the bottom clause $B = h(X,Y) \leftarrow b(X,X)$. B is indeed input-instantiated (X is instantiated in the head), but not output-instantiated since Y has no instantiation. ■

Our implementation of mode constraints is straightforward. We simply mimic the logic behind Definition 5. Informally, for each input variable v of a literal $q_i(\ldots)$, we require that it appears in at least one previous literal (as output) or the head (as input). Since the bottom clause head is always present in a candidate, no constraints apply when an input can be caught from the head. For each output variable v of the head that is not also an input to the head, we require that it appears in at least one output variable in a body literal.

Definition 7 *Let* B *be a bottom clause with* n *body literals. Let* I_i *and* O_i *be the input and output variables of body literal* i *(as given by the mode declarations), respectively. Similarly, denote by* I_h *and* O_h *the input and output variables of the head (as given by the modeh declaration), respectively. The mode constraints* F *of bottom clause* B *is a conjunction of clauses, constructed as follows:*

1. *For each* $v \in I_i$, $v \notin I_h$, *include clause* $\{\neg b_i\} \cup \{b_j : j < i, v \in O_j\}$.
2. *For each* $v \in O_h$, $v \notin I_h$, *include clause* $\{b_j : v \in O_j, j \in \{1,\ldots,n\}\}$.
3. *No other clauses are included.*

Note that if an output variable of B's head cannot be instantiated by any body literals, F will contain the empty clause (generated by the second rule) and therefore be inconsistent. This is correct, since no candidate from B will be output-instantiated.

The following theorem, which we prove in Appendix A, shows that our mode constraints are sound and complete.

Theorem 8 *Let F be a propositional formula for the mode constraints of B. Let C be a candidate from B. F generates C if and only if C is mode conformant.*

Proof See Appendix A. ■

2.3 Pruning Constraints

A candidate solution C is typically evaluated by comparing it against the set of all positive and negative examples. If the candidate covers a negative example, it is said to be *inconsistent*, otherwise *consistent*. Since our search lattice defines a subset order on the literals, where the top element is the empty clause and the bottom element is the bottom clause, the generality order is the subset order for body literals.

Definition 9 *Let C and D be candidates from B. We say that C is* more general *than D (with respect to B) if and only if C's body is a subsequence of D's body. We write $C \subseteq_B D$, omitting B whenever the bottom clause is clear from context. If C is more general than D, D is* more specific *than C.*

The following is a trivial consequence of the subset relation.

Lemma 10 *Let G and S be candidates. If $G \subseteq S$, then $G \Longrightarrow S$. The converse does not necessarily hold.*

Proof The subset relation is a special case of subsumption, for which the implication is well known (Nienhuys-Cheng and de Wolf, 1997). An application of self-resolution to any recursive clause demonstrates that the converse does not hold. ■

It follows that if $G \subseteq S$, G covers at least as many examples as S. In particular, if S covers a negative example and is thus inconsistent, all generalizations also cover that negative example, and are therefore also inconsistent. Since inconsistent solutions are of no interest to us, all generalizations of an inconsistent clause are redundant. On the other hand, if G covers no negative examples (G is consistent) and p positive examples, no specialization S can cover more than p positive examples. Hence the specializations of a consistent clause are redundant.

We would like to store information about which regions have been pruned during our search, so there will never be any redundant evaluation with respect to the subset order. To achieve this, we need to know what the pruning formulae look like. We start with an example.

Example 9 *Let us say that $B = h(X) \leftarrow q_1(X), q_2(X), q_3(X), q_4(X)$ and $C = h(X) \leftarrow q_3(X), q_4(X)$. If C is inconsistent, all generalizations will also be inconsistent. We can represent C by the bit string* 0011*, where a* 0 *or* 1 *at position i indicates absence or presence of literal i, respectively. All generalizations of C are given by clauses that do not contain $q_1(X)$ and $q_2(X)$, so we can represent them using the schema* 00$**$*. Logically, this corresponds to $\neg b_1 \wedge \neg b_2$. Conversely, all specializations of C are given by clauses that contain $q_3(X)$ and $q_4(X)$, so their schema is $**11$. Logically, this is the propositional formula $b_3 \wedge b_4$. This suggests the conjunction of all literals in C give all specializations, whereas the negated conjunction of all literals missing in C gives all generalizations.*

Next, we define such formulae.

Definition 11 *Let C be a candidate from B. For any candidate X from B, let V_X be all the propositional variables corresponding to the first-order body literals of B that occur in X. The* propositional formula for all generalizations of *C is*

$$P^B_{\uparrow C} = \bigwedge_{b_i \in V_B - V_C} \neg b_i.$$

The propositional formula for all specializations of *C is*

$$P^B_{\downarrow C} = \bigwedge_{b_i \in V_C} b_i.$$

We may then prove that our informal derivations are sound and complete.

Theorem 12 *Let C be a candidate from B.*

1. *$\neg P_{\uparrow C}$ generates G if and only if G is not a generalization of C.*
2. *$\neg P_{\downarrow C}$ generates S if and only if S is not a specialization of C.*

Proof See Appendix B. ■

The theorem is used in the following way: After evaluating a candidate C for coverage, we know whether it is consistent or inconsistent. If C is inconsistent, we prune all generalizations $P_{\uparrow C}$. This is done by inserting the requirement that no candidate generated by $P_{\uparrow C}$ is allowed: $\neg P_{\uparrow C} = \bigvee_{b_j \in V_B - V_C} b_j$. Similarly, if C is consistent, we add $\neg P_{\downarrow C} = \bigvee_{b_i \in V_C} \neg b_i$. Note that the constraints are always propositional clauses.

Example 10 *If $B = h(X) \leftarrow q_1(X), q_2(X), q_3(X), q_4(X)$ and $C = h(X) \leftarrow q_2(X), q_3(X)$, all generalizations of C are given by $\neg b_1 \wedge \neg b_4$ and all specializations by $b_2 \wedge b_3$. If C turns out to be inconsistent, all generalizations are also inconsistent, so we add the constraint $b_1 \vee b_4$. In particular, since any model either sets b_1 or b_4 to $true$, both C and the empty clause (top element) are excluded. If C turns out to be consistent, all specializations cover no more positive examples, so we add the constraint $\neg b_2 \vee \neg b_3$. Any model now sets b_2 or b_3 to $false$, which, for example, blocks both C and the bottom clause.*

2.4 Variable Splitting Constraints

Atom's and Progol's bottom clause construction assumes that equivalent ground terms are related when lifting instances. For example, when lifting

$$h(x,z) \leftarrow b_1(x,y), b_2(y,z)$$

the two occurrences of y are assumed to be more than a coincidence:

$$h(X,Z) \leftarrow b_1(X,Y), b_2(Y,Z).$$

Most of the time, this is not a problem since there will eventually be an example that reveals the coincidentally linked terms. However, Tamaddoni-Nezhad and Muggleton (2009) provide an example of a half adder, which cannot be induced with this restriction. Progol's A* solves this by

variable splitting during its search, but this method does not work when we represent clauses as binary strings, or, as in our case, propositional formulae. The solution, also taken by the Aleph ILP system (Srinivasan, 2001), is then to let the bottom clause represent these equality assumptions (variable splitting may also add many more literals to the bottom clause, but this will not affect our constraints).

As an optimization, we can ensure that no redundant candidates are generated due to these equalities, so we add constraints requiring that a variable splitting equality is used in at least one other literal. Since an added equality $X = Y$ matters only if both X and Y occur in the formula, we get the following variable splitting constraints.

Definition 13 *Let C be a candidate from B with a (possibly empty) prefix subsequence of k variable splitting equalities $V_i = W_i, i = 1,\ldots,k$ (V_i is a different variable from W_i). Let B_{V_i}, B_{W_i} be the set of variables corresponding to literals in C that are not variable splitting equalities ($i > k$) and contain V_i and W_i, respectively. For each b_i, $i = 1,\ldots,k$, we add two variable splitting constraints:*

1. $\{\neg b_i\} \cup B_{V_i}$, *and*
2. $\{\neg b_i\} \cup B_{W_i}$.

Note that input- and output-instantiation constraints will be added to the constraint database in the form of mode constraints; the equality constraints need only specify when equalities are redundant.

Example 11 *Let B be the bottom clause*

$$h(X,Y) \leftarrow V = W, q_2(X,V), q_3(Y,W), q_4(X,Y).$$

If C is a candidate from B containing the literal $V = W$, we require that both the variables V and W occur in C (other than as equalities generated from bottom clause construction). V occurs in the second literal and W in the third, so our propositional constraints are $\{\neg b_1, b_2\}$ and $\{\neg b_1, b_3\}$.

For instance, this prevents the candidate $h(X,Z) \leftarrow X = Y, b_2(X,A), b_3(A,Z)$ from being generated, as the equality is redundant since Y is never used. Instead, only the logically equivalent candidate obtained by removing the equality will be generated: $h(X,Z) \leftarrow b_2(X,A), b_3(A,Z)$.

2.5 Functional Constraints

Many predicates have the property that their truth value does not matter, only what they compute as output. We call such predicates *functional*. In particular, when chaining arithmetic operations as predicates, we are not concerned with their truth values, but only about obtaining a numeric output from inputs. (The predicate must also be *pure*, that is, free of any side effects, at least as far as can be measured by other predicates in use.)

The idea is that for functional predicates, we require that *at least one of its output variables occur in another literal*.

Example 12 *The is/2 operator computes an arithmetic expression given by its second argument and unifies the result with its first argument. With mode declaration*

$$modeb(1, is(-Real, +Expr), [functional])$$

we declare it functional. Then,

$$average(X,Y,A) \leftarrow S\,is\,X+Y, A\,is\,S/2$$

is a viable candidate to evaluate since the output S is used to compute the average of X and Y, and A is used as the final answer that is instantiated by the head. However,

$$average(X,Y,A) \leftarrow S\,is\,X+Y, D\,is\,Y-X, A\,is\,S/2$$

is logically equivalent but should not be a viable candidate, since the output D is never used (it occurs only as a singleton). Since the is-operator is always true when the left hand side variable is uninstantiated, 'D is Y − X' is a useless operation and the redundant candidate may safely be blocked from consideration.

Definition 14 *Let B be a bottom clause with n literals. A modeb declaration for a predicate can declare it a* functional predicate*. If the ith literal of B is declared a functional predicate, and none of its outputs occur in the head (as input or output), we define its corresponding* functional constraint *to be the propositional clause:*

$$\{\neg b_i\} \cup \{b_x : x \in \{1,2,\ldots,n\} \wedge x \neq i \wedge O_i \cap (I_x \cup O_x) \neq \emptyset\}.$$

If one or more of the ith literal's outputs occur in the head, no constraint is generated (clearly the predicate is always useful then).

Essentially, functional predicates filter out clauses which *only* have an effect through their predicate's truth value. Note that functional predicates are not required to have an effect on literals appearing later: its outputs may also be compared with previous known values, whether it may be inputs or outputs. Note that a functional predicate always generates at most one propositional clause, since we only require that *at least one* of its output variables occurs in another literal.

Example 13 *Elaborating more on Example 12, consider the bottom clause*

$$B = average(X,Y,A) \leftarrow S\,is\,X+Y, D\,is\,Y-X, A\,is\,D+X, A\,is\,S/2.$$

If we declare the predicate is/2 as functional, one constraint we would obtain is $\{\neg b_2, b_3\}$*. Intuitively, the constraint reflects the fact that if the second literal is used (D is Y − X), then D must appear elsewhere since otherwise the literal does nothing useful in functional terms. The only other occurrence of D is in the third literal, so* $b_2 \longrightarrow b_3$*. With this constraint, our candidate C from the previous example would never be generated, since the model for C sets* b_2 *to true and* b_3 *to false.*

Another constraint we would get is $\{\neg b_1, b_4\}$*. This constraint is due to the output S in the first literal, which must be used in its only other occurrence, the fourth literal. This one does however not prevent C, as* b_4 *is true in C's model.*

We do not get any constraints from the third and fourth literal output A; this is because A already occurs in the head.

3. NrSample: Constraint-Based Induction

NrSample's search algorithm is simple: after a bottom clause has been constructed, the mode constraints are generated. The SAT solver is then invoked to acquire a valid candidate, which is then evaluated. Pruning constraints are then added based on the candidate's consistency/inconsistency. The procedure of invoking the SAT solver to obtain candidates proceeds until a termination criterion is triggered (usually the maximum number of nodes to explore) or the search space is exhausted with respect to the constraints (no model can be acquired).

The induction procedure follows a sequential covering algorithm: The first available example is used to construct a bottom clause, leading to a space of possible candidates. If a viable candidate can be found from this search space, all positive examples it covers are removed. If no candidate is found, the example itself is moved to the generalized theory. This process repeats until all positive examples are covered. The bottom clause construction algorithm is equivalent to Progol's and is somewhat involved; we refer the reader to Muggleton (1995) for a detailed description. Briefly, it uses the mode declarations to compute the set of ground truths and lift them to variable instances. Each ground truth then becomes a body literal in the bottom clause. Pseudo-code for NrSample is given in Algorithm 1.

```
 1. While there is a positive example available:
 2.   Generate bottom clause from the example.
 3.   Generate mode constraints.
 4.   While no termination criterion is satisfied:
 5.     Call SAT solver to retrieve a model for the constraints.
 6.     If no model can be retrieved, terminate search (exhaustion).
 7.     Convert model into a (valid) candidate.
 8.     Evaluate candidate (using any evaluation function).
 9.     Update best-so-far candidate.
10.     Generate pruning constraints based on evaluation.
11.   Pick best candidate, remove cover (sequential covering).
```

Algorithm 1: NrSample's Search Algorithm.

In the following sections we focus on important details of SAT solving, search order, and maximum clause lengths (step 5), as well as efficient candidate evaluation (step 8).

3.1 Solving the Constraint Database

Note that all the constraints presented in Section 2 are in disjunctive form; in other words, they are propositional clauses. Each constraint represents a logical requirement for what needs to be satisfied, so our constraint database is a conjunction of clauses. Hence our database is in conjunctive normal form (CNF), which allows us to invoke many well studied SAT solvers without the need for CNF conversion (Russell et al., 1996).

For the general case, the Boolean satisfiability problem is known to be NP-complete (Cook, 1971). It may however not be clear that our SAT problems are NP-complete since the constraints contain regularities not assumed in the general case. By Definition 7, all mode declarations have at most one negative literal (that is, they are dual horn clauses). Satisfiability is hence decidable

in linear time (Dowling and Gallier, 1984). The pruning constraints, on the other hand, have all positive or all negative literals by Definition 11, leading to NP-completeness (Schaefer, 1978).

In practice, SAT has been extensively researched and high performance algorithms for quickly solving real world cases of the Boolean satisfiability problem exist. Chaff (Moskewicz et al., 2001) is a well known algorithm upon which many modern state-of-the-art SAT solvers are based, claiming to be able to solve problems with millions of variables.[4]

To ensure termination within a reasonable amount of time, we also provide a maximum number of literal assignments for the SAT solver. Our default value is 50000.

We will address the issue of SAT overhead from an empirical point of view in Section 4, but there is a point to be made about the NP-completeness of pruning constraints. The fact that finding a non-redundant candidate solution is NP-complete is not a flaw of our approach, but expresses the difficulty of constructing non-redundant solutions in ILP (with respect to pruning). Although not always true, insofar as this difficulty translates into low probabilities of randomly sampling a non-redundant candidate, algorithms that are indifferent about pruning constraints are no better off. In such cases, then, our algorithm would simply stop searching (due to the literal assignment limit, by default 50000), whereas other algorithms are likely to sample redundant candidates, therefore wasting time.[5]

3.2 Selection Strategies and Traversal Order

Since our approach relies on using a SAT solver to retrieve valid candidates, the search strategy—that is, the traversal order for candidate generation—is itself embedded into the SAT solver's model construction.

A *model* is a (propositional) assignment of truth values to all literals which satisfies all constraints. DPLL-based SAT solvers are a class of complete model builders (Russell et al., 1996; Davis et al., 1962). They construct models by first selecting an unassigned literal and assigning it either to true or false (according to some selection strategy), then propagating the effects of this assignment to all propositional clauses in the database. Propagation of an assignment $b_i = true$ works as follows: all clauses containing the literal b_i are removed (since they are now covered), and all clauses containing its negation $\neg b_i$ will have that literal removed (since that literal is not covered). When assigning $b_i = false$, the role of b_i and $\neg b_i$ are simply interchanged. Clauses that now only contain one literal—called *unit clauses*—indicate a definite assignment, so its effects are also propagated to all clauses.

When no more propagations are possible, either all literals have been assigned and we have a model, or we have reached a new choice point where any unassigned literal may be assigned true or false (again, this is handled by a selection strategy).

In what follows, we will assume that the DPLL-based algorithm follows the traditional technique of backtracking to the first literal that has not been tried both ways (that is, has not previously been assigned both to true and false). This is known as chronological backtracking; the case of non-chronological backtracking will be treated later.

Propagation may produce an empty clause, signifying a contradiction. This triggers a process of backtracking through the stack of assignments, searching for the most recent assigned literal that

4. This claim is made on Chaff's official webpage, `http://www.princeton.edu/~chaff/zchaff.html`. Visited on 2012-08-14.
5. The argument that DPLL-based SAT solvers can get stuck in entire subtrees with no solution, whereas the solution is readily available in a sibling branch, is not valid when random restarts are employed.

has not been tried with both assignments. This literal's assignment is then flipped, and we re-enter the process of propagating, backtracking, and selecting literals.

Freedom to pick any unassigned literal and assign it either true or false corresponds to choosing a literal from the bottom clause and deciding whether to include it or not (recall that a propositional literal b_i correspond to the ith first-order literal of the bottom clause). The restriction imposed by DPLL-based solvers comes from the fact that, if our choices of literals fail to satisfy the constraints, we may not change the assignments arbitrarily, but rather, must let the SAT solver flip all assignments (in reverse chronological order) that have not been tried both ways first.

Example 14 *Consider making the sequence of assignments* $(b_3 = true, b_1 = false, b_8 = true)$*, upon which the database is inconsistent (the empty clause was generated at the point where we assigned* $b_8 = true$*). This corresponds to choosing that the third and eighth literal of the bottom clause should be included in the candidate, whereas the first should not. The remaining literals have yet to be specified for inclusion/exclusion. If we had full freedom to choose any search strategy, we may for example want to start from scratch and exclude the third literal, not caring about the first and eighth. DPLL's backtracking algorithm prevents us from doing so however: it will first attempt to flip the last made assignment* $b_8 = true$ *into* $b_8 = false$*, thus producing the sequence* $(b_3 = true, b_1 = false, b_8 = false)$*. This makes it clear that we have no say in what the third, first, and eighth literal should be until the DPLL algorithm has backtracked to those choice points. Propagation may also force certain assignments, but this is desirable since it is this mechanism that excludes invalid candidates. After propagation, we are free to select any of the remaining unassigned literals in our next choice point, and assign it to either true or false.*

Example 14 is a simplification, as modern DPLL-based algorithms do not necessarily flip the last literal not tried both ways. Instead, they may go back further, to an older choice point and flip its literal, known as non-chronological backtracking or backjumping. This affects the predictability of the search (the example above is no longer valid), but it remains true that we have full freedom to select an assignment when, and only when, a choice point is reached. Chaff (Moskewicz et al., 2001) uses backjumping, but for a predictable search order, our algorithm uses chronological backtracking.

Some simple selection strategies include randomly assigning a literal, or always selecting the literal that occurs the most often in the database. Chaff introduced an efficient selection strategy known as VSIDS (Variable State Independent Decaying Sum) (Moskewicz et al., 2001). Which strategy would do best for ILP depends on the problem domain.

As we will see later, our benchmarks compare NrSample's performance against an algorithm that emulates its traversal path without using propositional constraints. Hence it must be reasonably easy for the emulation to also emulate the selection strategy without such constraints. This excludes VSIDS, as it explicitly depends on counting occurrences of propositional literals in the constraint database. Our selection strategy is to assign the literals in reverse order, to false first: $(b_n = false, b_{n-1} = false, \ldots, b_1 = false)$. This way, we always start with the top element, and backtracking starts flipping b_1 before b_2, b_2 before b_3, and so on.

Example 15 *If we assume that no backtracking occurs until the last assignment, we can visualize the sequence of candidates generated according to the aforementioned selection strategy (assigning literals to falsehood in reverse order). In bit string notation (0 signifies false and 1 true), with a bottom clause containing* 3 *literals, this sequence is:* $(000, 100, 010, 110, 001, 101, 011, 111)$*. The first candidate,* 000*, comes from assigning the literals to false in reverse order:* $(b_3 = 0, b_2 =$

$0, b_1 = 0$). *We then backtrack, flipping the last made assignment* $b_1 = 0$ *into* $b_1 = 1$. *This gives the assignment* $(b_3 = 0, b_2 = 0, b_1 = 1)$, *that is, the candidate* 100. *Next,* b_1 *has been tried both ways, so we remove the assignment.* b_2 *is now flipped, so we have the partial assignment* $(b_3 = 0, b_2 = 1)$. *Since we always start by assigning to falsehood, we then add* $b_1 = 0$, *obtaining the assignment* $(b_3 = 0, b_2 = 1, b_1 = 0)$, *and thus the candidate* 010. *This process continues until all candidates have been tried. Note that in practice, the assignments are unlikely to always reach the last literal before backtracking occurs due to the presence of constraints.*

Note that the restrictions imposed by DPLL-based SAT solvers are due to its backtracking mechanism, which is used to ensure completeness. It is also possible to use non-DPLL-based SAT solvers (Selman et al., 1995) with NrSample, for which no restrictions are imposed regarding literal selection strategy. However, care must be taken to ensure termination when no model is available: a timeout may be used when a solution cannot be found after a certain amount of time has elapsed or a maximum number of literal assignments tried. This ensures termination but not completeness.

That we can never have full control of search order traversal using our propositional constraints is clear, since the very reason for using them is to prevent certain candidates from being generated. However, it is in part possible to overcome limitations of DPLL-based backtracking by directing the search to desirable regions. This can be achieved by inserting a constraint to specify that region. When all candidates have been depleted, we negate this constraint, which forces the SAT solver to explore the complement region. This process can be done recursively, further partitioning subregions. We call such temporary constraints *regional constraints*.

Example 16 *Assume we want to start by exploring all candidates containing only a subset of the first 3 literals. The regional constraint is then* $\neg b_4 \wedge \neg b_5 \wedge \ldots \wedge \neg b_n$, *which translates into propositional clauses* $\{\neg b_4\}, \{\neg b_5\}, \ldots, \{\neg b_n\}$. *Once the search space has been exhausted, we negate this formula to obtain the complement region:* $b_4 \vee b_5 \vee \ldots \vee b_n$. *While exploring any of these two regions, we can apply the same principle to further partition regions into subregions.*

Care must be taken to treat regional constraints differently, as failure to satisfy them does not necessarily mean that no valid candidate exists: it only specifies that this subregion has been fully explored; what remains is now the complement region.

3.3 Maximum Clause Lengths

ILP systems give users the option to alter default parameters related to theory induction. For example, Progol allows users to set a limit on the number of iterations in the bottom clause construction, as well as on the maximum number of candidates to be evaluated (per bottom clause). These limits do not interfere with NrSample's constraint framework, since bottom clause construction is separated from search.

However, it is useful to also restrict the number of body literals allowed in any candidate solution during the search.[6] Although this restriction could be implemented as propositional constraints in NrSample, it is tedious to do so as each combination that is not allowed has to be specified explicitly. For example, with a bottom clause containing $n = 4$ body literals, the following formulae are needed to restrict all its candidates to at most $c = 2$ of those: $\{\neg b_1, \neg b_2, \neg b_3\} \wedge \{\neg b_1, \neg b_2, \neg b_4\} \wedge$

6. This is achieved using the query $set(c, N)$ in Progol and $set(clauselength, N)$ in Aleph.

$\{\neg b_1, \neg b_3, \neg b_4\} \wedge \{\neg b_2, \neg b_3, \neg b_4\}$. In general, we need to specify that no clause with $c+1$ literals is allowed, so we get $\binom{n}{c+1}$ constraints. A more efficient way to implement this restriction is to do so in the SAT solver itself. Since the Chaff algorithm works by assigning a truth value to a propositional variable and unit propagating the implied assignments, we keep track of the number of variables assigned true at all times. Whenever the limit is exceeded, backtracking occurs, ensuring that we never generate a solution with more than c positive assignments. As this is a non-standard feature of DPLL solvers, we have embedded our own implementation of the Chaff algorithm into NrSample.

Moreover, we generalize this technique so that our algorithm stores a mapping of which literals were generated from what mode declarations during bottom clause construction, and then keep track of how many times a mode declaration has been used during SAT solving. This enables us to specify upper bounds on the number of times a certain mode may be used in a candidate solution, a feature which is particularly useful to prevent large number of recursive literals in a predicate. This is implemented by using an extended mode declaration $modeb(Recall, Atom, Max\_Occurs)$ in which the third argument *Max_Occurs* specifies the maximum number of times a literal generated from this mode can occur in any candidate solution. Note that the bottom clause may still contain more occurrences, although not all of them may be used at the same time in a candidate. A search strategy based on this idea was proposed in Camacho (2000), where the number of occurrences of a predicate is progressively incremented.

3.4 Lexicographic Evaluation Function

The quality of a candidate is computed using an evaluation function. Different evaluation functions are possible. For example, if the number of positive and negative examples covered is P and N respectively, and L is the number of literals in the body of the candidate, Progol's A^* computes the fitness as $P-N-L-E$, where E is the fewest literals needed to get an input-output connected clause.[7] By default, Aleph uses the evaluation function P/T, where T is the total number of examples to be evaluated.

NrSample's default evaluation function assumes that our quality measure obeys a lexicographic ordering for $(P,-L)$. That is, $(P_1,-L_1) < (P_2,-L_2)$ if and only if $P_1 < P_2$ or $P_1 = P_2$ and $L_2 < L_1$. Intuitively, this states that (consistent) candidates are first compared by coverage, and only when they cover equally much are they compared by number of literals (fewer literals is better). We never insert an inconsistent candidate into the knowledge base, regardless of search algorithm. If no consistent candidate is found, we add the example itself to the knowledge base.

With respect to our quality measure, there are two observations that will reduce the number of examples evaluated.

First, consistency of a candidate is defined solely in terms of negative example coverage; there is hence no need to evaluate any positive examples to determine consistency. Since we never add an inconsistent candidate, there is no need to evaluate positive example coverage when the candidate is inconsistent. Thus all candidates should first be evaluated for negative coverage, and never for positive whenever they turn out to be inconsistent. Negative coverage evaluation can stop as soon as we detect one covered negative example.[8]

7. This corresponds to the number of positive assignments in a minimal model for the mode constraints.
8. More generally, with noise tolerance T, we stop after covering T negative examples. For $T = 0$ we have the noise free setting.

Second, we may safely abort positive coverage computation when there are not enough positive examples left for a candidate to beat the best-so-far. For example, if we have a best-so-far candidate that covers 20 out of 50 positive examples, and our candidate under evaluation covers 3 out of the first 34 examples, we may safely abort the last 16 examples since, even if they were all covered, it would still only amount to $3+16=19$ which is less than the already 20 previously covered. This optimization is possible since we need not consider clause lengths when two clauses have different coverage.

4. Experimental Results

Our benchmarks have two purposes.

First, we want to directly measure the effects of using propositional constraints in NrSample. Put differently, we would like to know whether producing, storing, and solving propositional constraints provides any real benefit over simply generating each candidate and then checking whether it conforms to the input-output specification and is logically non-redundant. To this end, we use an algorithm called *emulate_nrsample*, which searches for candidates in exactly the same order as NrSample, but without using constraints. Since both NrSample and emulate_nrsample traverse the search space in identical ways, comparing their execution time effectively measures the performance difference between solving the constraints and discarding invalid candidates.

Second, we want to measure NrSample with well established algorithms. Here we turn to the more general question of how useful NrSample is as an induction algorithm. To answer this, we compare NrSample against Progol's A^* and Aleph's enumeration search. For a fair comparison, all algorithms are implemented in our ILP system Atom, using the same libraries and backend.

4.1 The Search Algorithms

Progol's A^* is best-first search applied to ILP using the evaluation function $P-N-L-T$ where P and N are the number of positive and negative examples covered, respectively, L the number of body literals in the clause, and T the minimum number of literals needed to get an output instantiated candidate.

Enumeration search constructs the candidates of the search lattice level by level starting from the top. First, the top element is created, followed by all candidates with one body literal, then all candidates with two body literals, and so on. Seen as bit strings $(s_i)_1^n$, where 1s indicate that literal b_i from the bottom clause is used, we start with the candidate corresponding to bit string "11...00", the number of 1s corresponding to the current level as described above, and take previous permutations until we reach "00...11". In other words, in each layer we start by generating the candidate containing all leftmost literals of the bottom clause (as many as the depth we are considering), and cycle through all permutations until we reach the candidate that has all right-most literals.

Enumeration search uses the same evaluation function as NrSample, with all optimizations as described in Section 3. Progol's A^* cannot use any of these optimization as the heuristic needs to know the precise number of positive and negative examples covered in order to decide which nodes to expand.

The search strategy used by NrSample assigns right-most literals to false first (that is, in the sequence $b_n, b_{n-1}, \ldots, b_1$), so that the top element is explored first. It will then start flipping assignments in reverse order, so that b_1 will be assigned true first, then b_2, etc. For details, see Section 3.2, and, in particular, Example 15. We use no random restarts, making our algorithm deterministic.

emulate_nrsample generates each candidate in turn and then checks for mode and input-output conformance. When all invalid candidates have been discarded, the traversal order is the same as that of NrSample. Thus emulate_nrsample avoids the overhead of solving constraints, at the expense of potentially generating a large amount of unnecessary candidates.[9]

4.2 Test Problems and Data Sets

All comparisons are done using a set of well known ILP problems.

Three concept learning problems are used: ANIMALS, GRAMMAR, and TRAIN. These are taken from the data set in the Progol 4.4 distribution.[10]

ANIMALS is a classification problem: a list of animals is to be divided into mammals, fishes, reptiles, and birds. GRAMMAR presents examples of well formed and ill formed sentences, as well as a tiny dictionary of words grouped by word classes. The goal is to learn phrase structure rules. TRAIN is a binary classification problem: trains are either eastbound or westbound, depending on properties of their cars. The goal is to determine these dependencies.

Our remaining tests are program synthesis problems: MEMBER, SORTED, REVERSE, LENGTH, ADD, APPEND, SUBLIST.[11] MEMBER learns conventional Prolog list membership. SORTED is a unary predicate determining whether a list is sorted or not. REVERSE is true if and only if its second argument—a list—is a reversed form of its first (the relation is symmetric, but this is not exploited). LENGTH determines the number of elements in a list. The lists contain integers, which confuses the learning algorithms as it is not clear that the integer value of the list elements have nothing to do with the list length. This makes the problem significantly harder to solve. ADD defines formal addition in Peano arithmetic between two natural numbers. They are represented using the successor function (for example, integer 3 is represented as $s(s(s(0)))$. APPEND defines list concatenation. Finally, SUBLIST defines the ordered subset relation for lists: $sublist([],A)$, $sublist([A|B],[A|C]) \leftarrow sublist(B,C)$, $sublist(A,[B|C]) \leftarrow sublist(A,C)$.

Each of the program synthesis problems come in two variants. In the first variant, we use a set of predicates that is particularly well suited to each concept. For example, for APPEND, this would be operations to construct and deconstruct lists. Tables 8 and 9 in Appendix C shows the number of positive and negative examples, as well as mode declarations, for all problems. We refer to these problems—including the concept learning problems ANIMALS, GRAMMAR, and TRAIN—as the *small problems*.

In the second variant, we use a fixed set of predicates across *all* program synthesis problems. That is, we include mode declarations that allow for constructing and deconstructing lists, comparing/ordering integers, and performing elementary arithmetic on numbers. The precise definition of this primitive set of predicates is given in Appendix D. We refer to these problems as the *large problems*, and distinguish them from the small problems by using a prefix "L:". For example, MEMBER refers to the small problem and L:MEMBER to the large problem. As the primitive set is primarily

9. From a technical point of view, emulate_nrsample does not actually generate candidates as clauses, but rather, as a set representing which literals from the bottom clause each candidate is made up of. This is enough to check whether it is mode and input-output conformant. If the candidate is valid, the full candidate is generated and checked for coverage. This improves the performance of emulate_nrsample. The same optimization is used for enumeration. Progol's A^* does not check for input-output conformance of candidates, as it may need to expand invalid nodes to reach valid ones.
10. Available in the distribution of Progol 4.4 at `http://www.doc.ic.ac.uk/~shm/Software/progol4.4/progol4_4.tar.gz`. Visited on 2012-08-14.
11. Distributed with Atom's source code, see previous footnote.

intended for list and integer manipulation, the concept learning problems are not included in the large problems.

As the large problems use a fixed set of predicates—of which many predicates are intended for arithmetic computations, and thus functional in nature—we take the opportunity to test NrSample with functional constraints (*NrSFun*) against NrSample without functional constraints (*NrS*). Functional predicates are described in Section 2.5, and all functional predicates in the primitive set (see Appendix D) have the keyword "functional" in their modeb declaration. For *NrSFun*, this generates functional constraints, whereas *NrS* simply ignores the keyword.

The large amount of primitive predicates causes a combinatorial explosion of possibilities, thereby creating very large bottom clauses. We believe these large data sets better correspond to practical use of program synthesis, as the primitive predicates are general enough to induce all aforementioned concepts without any tweaking.

For each program synthesis problem, we generate 10 data sets. Each has its own (independently generated) positive and negative examples. We now describe how the examples in each of these data sets are generated.

All tests except ADD involve lists. Thus we need an algorithm to sample lists. We assume the lists hold a finite number of constants as elements. First, we note that the sample space is infinite, as lists can get arbitrarily long. Second, except for a finite subset, lists of increasing sizes necessarily must have diminishing probabilities.[12] This is also reasonable, since we expect simple observations (short lists) to be more common than elaborate ones. We first define the length L of a sampled list to be geometrically distributed with success probability 0.25: $P(L = k) = 0.75^k \cdot 0.25$. This makes sampling short lists more likely than long lists but puts no upper bound on the length of a list. For each of the L positions, we uniformly sample for a single digit constant ($\{0, 1, \ldots, 9\}$).

With our list generator, the positive examples are generated in the obvious way: for a positive example of MEMBER, sample a list and randomly pick an element of the list. For a negative example, sample a list not containing all domain elements and randomly pick an element not in the list. For APPEND, randomly sample two lists, and then append them to obtain the appended list. For a negative example, randomly sample three lists, and verify that the third is not obtained by appending the first and second. Also, we ensure no duplicate examples are generated.

For the ADD problem, we sample uniformly from $\{0, 1, 2, 3, 4\}$, providing the first $5 \cdot 5 = 25$ ground instances as positive examples. The reason for limiting formal addition to small examples is due to a depth limit of $h = 30$ when performing queries: each application of the successor function requires one call, so the largest number that can be handled by queries is 30. Such computational limits are necessary to make ILP problems tractable; all limits used in our benchmarks are listed in Section 4.3. L:ADD is different from ADD in that it does not represent numbers using the successor function but using lists of zeros. The length of the list is the number to be represented, that is, 3 is represented as $[0, 0, 0]$. This is because the primitive set in the large problems was primarily chosen with list manipulation in mind.

The concept learning data sets—ANIMALS, GRAMMAR, and TRAIN—are taken from Progol without modifications. For each of these three problems, we consider 10 different random orderings of the examples. Since the greedy sequential covering algorithm depends on the order of the positive examples, this affects the results.

12. Give the lists an enumeration and let p_n be the probability of sampling list n. $\sum p_n = 1$, which implies $p_n \to 0$.

4.3 Cross Validation

We consider two measures of the quality of an algorithm: how accurate the solution is, and the time it takes to generate a solution (execution time). Our benchmarks are performed using cross validation.

On the concept learning problems—ANIMALS, GRAMMAR, and TRAIN—which contain few positive examples, we perform leave-one-out experiments.

On the program synthesis problems, we use hold-out validation with 10 different data sets. Accuracy is computed as the fraction of all correct predictions among all predictions made.

For the sake of tractability, we also impose some computational limits to all benchmarked algorithms. During bottom clause construction, a maximum of $i = 3$ iterations are performed. For each bottom clause constructed, a maximum of $n = 1000$ (valid) candidates are explored. Candidates are restricted to a maximum clause length of $c = 4$. For each query, we use a recursion depth limit of $h = 30$ and maximum resolution limit of $r = 10000$. A time limit of $t = 600$ seconds is imposed on any induction. Upon reaching this time limit, the entire induction aborts, although post processing is still necessary to ensure all remaining examples are put back into the knowledge base. Except for the time limit, these restrictions are commonly used in Progol and Aleph (the values, of course, depend on the problem domain).

Progol A^* evaluates all candidates, regardless of mode declarations—this is necessary in order to choose the best node to expand in the search lattice. Aleph's enumeration explicitly states that the invalid candidates are to be ignored, as the following quotation from the Aleph homepage shows:

```
With these directives Aleph ensures that for any hypothesised clause
of the form H:- B1, B2, ..., Bc:

Input variables.
Any input variable of type T in a body literal Bi appears
as an output variable of type T in a body literal that appears before Bi,
 or appears as an input variable of type T in H.

Output variables.
Any output variable of type T in H appears as an output variable
of type T in Bi.
```

Without the time limit t, this algorithm may thus end up stuck in almost infinite generate-and-test loops when the number of valid candidates is small.

NrSample also needs a limit to ensure tractability while solving its propositional constraints: we set a limit of 50000 literal assignments per model constructed (equivalently: for each constructed candidate). When this limit is reached, the algorithm simply gives up the current bottom clause search, adds the example, and resumes sequential covering.

All benchmarks are performed on an Intel Core i7 (4×2.4 GHz) with 8 GB of RAM. All displayed numerical results are rounded to 3 decimals. For easier readability, trailing zeros are not shown (that is, we write "1" rather than "1.000").

Test	A_{NrS}	A_{Em}	A_{A^*}	A_E	A_{Em}/A_{NrS}	A_{A^*}/A_{NrS}	A_E/A_{NrS}
ANIMALS	0.9	0.9	0.952	0.971	1	1.058	1.079
GRAMMAR	0.981	0.981	0.952	1	1	0.970	1.019
TRAIN	1	1	0.9	1	1	0.9	1
MEMBER	0.983	0.983	0.935	0.983	1	0.951	1
SORTED	0.911	0.911	0.894	0.911	1	0.981	1
REVERSE	0.818	0.818	0.816	0.818	1	0.998	1
LENGTH	0.672	0.672	0.665	0.672	1	0.990	1
ADD	0.922	0.691	0.778	0.928	0.749	0.844	1.007
APPEND	0.804	0.411	0.783	0.590	0.511	0.974	0.734
SUBLIST	0.891	0.891	0.836	0.887	1	0.938	0.996

Table 1: Accuracy for Small Data Sets.

4.4 Results for Small Data Sets

Table 1 displays the accuracy of each algorithm as well as comparative ratios. A denotes Accuracy (defined in Section 4.3), with index NrS for NrSample, Em for emulate_nrsample, A^* for Progol's A^*, and E for enumeration. Since higher accuracy is better, the ratios compare favorably to NrSample when they are less than 1. Table 2 displays execution time—denoted T and measured in seconds—as well as comparative ratios. As lower execution time is better, a ratio larger than 1 is favorable to NrSample.

It is also interesting to know how often NrSample exhausts all valid candidates from search spaces. NrSample manages to exhaust all search spaces except REVERSE (95% exhausted), ADD (80%), and APPEND (60%). This suggests that—at least for NrSample—APPEND is the most difficult problem, followed by ADD and REVERSE.

As expected, NrSample has the same accuracy as emulate_nrsample on all small data sets except APPEND and ADD, where the latter times out. This is due to the fact that emulate_nrsample explores the same candidates as NrSample, provided no time out occurs. For the tests where emulate_nrsample did not time out, it sometimes completed the induction slightly faster than NrSample (ANIMALS, MEMBER, SORTED, LENGTH, and, most notably, TRAIN). On the tests with largest search spaces—APPEND, ADD, and REVERSE—NrSample performed substantially better than its emulated counterpart: 33, 4.5, and 9.4 times better, respectively.

As NrSample relies on input-output constraints, it is informative to divide the experimental results into concept learning—ANIMALS, GRAMMAR, and TRAIN—and program synthesis (all other problems).

On the concept learning problems, NrSample does not display any notable improvements over Progol A^* and enumeration. In particular, NrSample is substantially slower than both on the TRAIN problem: $1/0.07 \approx 14$ times slower than Progol's A^* although with better accuracy, and $1/0.016 \approx 63$ times slower than enumeration with same accuracy. On the other hand, NrSample is 63 times faster than Progol's A^* on GRAMMAR; indeed, it is the fastest algorithm on this problem, with comparable accuracy.

We note that accuracy on the concept learning problems is not always perfect (with any algorithm), despite their relative simplicity. This is because there are too few examples to always induce the correct definitions, even with leave-one-out. For example, in one instance of ANIMALS, all def-

Test	T_{NrS}	T_{Em}	T_{A^*}	T_E	T_{Em}/T_{NrS}	T_{A^*}/T_{NrS}	T_E/T_{NrS}
ANIMALS	0.021	0.018	0.057	0.018	0.857	2.714	0.857
GRAMMAR	0.018	0.021	1.133	0.021	1.167	62.944	1.167
TRAIN	2.575	1.438	0.18	0.04	0.558	0.070	0.016
MEMBER	0.163	0.156	28.637	0.156	0.957	175.687	0.957
SORTED	0.134	0.129	47.667	0.128	0.963	355.724	0.955
REVERSE	0.135	0.127	4.228	0.356	9.407	31.319	2.637
LENGTH	1.161	1.098	256.512	1.101	0.946	220.941	0.948
ADD	131.979	600.003	215.898	59.742	4.546	1.636	0.453
APPEND	18.088	600.019	43.082	600.027	33.172	2.382	33.173
SUBLIST	26.232	50.538	376.864	23.499	1.927	14.367	0.896

Table 2: Execution Time in seconds for Small Data Sets.

initions are correct except one, which is too general: $class(A, fish) \longleftarrow has\_legs(A, 0)$. The correct definition cannot be induced as the (only) example we need is being held out for validation.

Comparing NrSample against Progol A^* and enumeration on the program synthesis problems, we see that NrSample has better accuracy on all tests (except a tiny difference in favor of enumeration on ADD). It is also substantially faster than Progol's A^* on all problems, ranging from 1.6 to 356 times faster. It is slightly slower than enumeration on most tests: the exceptions are ADD, where enumeration is $1/0.453 = 2.2$ times faster, REVERSE, where NrSample is 2.6 times faster, and APPEND, where NrSample is 33 times faster.

4.5 Results for Large Data Sets

The large data sets test five algorithms: NrSample with functional constraints (*NrSFun*), NrSample without functional constraints (*NrS*), Emulated NrSample (*Em*), Progol's A^* (A^*), and enumeration E.

Table 3 shows accuracy on all problems, Table 4 shows accuracies compared with *NrSFun*. Since higher accuracy is better, lower values are in favor of *NrSFun*. Table 5 shows execution time, and Table 6 shows execution time compared with *NrSFun*. Since lower execution time is better, higher values are in favor of *NrSFun*.

The proportions of search spaces fully exhausted by *NrSFun* and *NrS* are given in Table 7. As pointed out earlier, NrSample detects exhaustion due to the unsatisfiability of its constraints, saving execution time. The results suggest that the most difficult problem in this regard is APPEND, followed by SUBLIST. SORTED and LENGTH are relatively difficult for *NrS*, which exhaust 60% and 70% respectively, whereas *NrsFun* fully exhaust all search spaces of both those problems.

As can be seen, NrSample with functional constraints induces with similar or better time performance compared to NrSample without functional constraints: on average 21 times faster. Notably, *NrSFun* is 11 times faster on MEMBER, 53 times faster on LENGTH, and 80 times faster on SORTED. All performance gains come without any accuracy penalty: accuracy is similar or better on all problems. Moreover, *NrSFun* is substantially more accurate on SUBLIST: 80% versus 48%.

Next, we note that emulate_nrsample times out on all data sets. As a result, it has worse execution time and accuracy compared to NrSample on all problems, demonstrating that the propositional framework is necessary for efficient constraint solving.

Test	A_{NrSFun}	A_{NrS}	A_{Em}	A_{A^*}	A_E
L:MEMBER	0.99	0.981	0.333	0.935	0.472
L:SORTED	0.834	0.827	0.027	0.397	0.72
L:REVERSE	0.555	0.555	0.349	0.562	0.411
L:LENGTH	0.665	0.663	0.335	0.413	0.366
L:ADD	0.83	0.808	0.676	0.784	0.83
L:APPEND	0.932	0.942	0.333	0.708	0.372
L:SUBLIST	0.803	0.478	0.352	0.826	0.517

Table 3: Accuracy for Large Data Sets.

Test	$A_{NrS}/A_{\text{NrSFun}}$	A_{Em}/A_{NrSFun}	$A_{A^*}/A_{\text{NrSFun}}$	A_E/A_{NrSFun}
L:MEMBER	0.991	0.336	0.944	0.477
L:SORTED	0.992	0.032	0.476	0.863
L:REVERSE	1	0.629	1.013	0.741
L:LENGTH	0.997	0.504	0.621	0.55
L:ADD	0.973	0.814	0.945	1
L:APPEND	1.011	0.357	0.76	0.399
L:SUBLIST	0.595	0.438	1.029	0.644

Table 4: Relative Accuracy for Large Data Sets.

Continuing, *NrSFun* is approximately as accurate or better than Progol A^* on all problems. *NrSFun* is substantially faster, with a speedup of between 7 and 45 times on all problems except APPEND and SUBLIST. On APPEND, A^* substantially sacrificed accuracy (71% versus 93%) to induce 8 times faster. Only on SUBLIST is the advantage uncontested, as A^* has both better execution time and slightly better accuracy (3.6 times faster with 83% versus 80% accuracy). On average, *NrSFun* is 18 times faster than A^*.

Finally, *NrSFun* has substantially better accuracy than enumeration on all problems except ADD, where they are tied. *NrSFun* also has substantially better performance, ranging from 1 to 1358 times faster. On average, *NrSFun* is 236 times faster than enumeration.

Test	T_{NrSFun}	T_{NrS}	T_{Em}	T_{A^*}	T_E
L:MEMBER	5.432	61.345	600.009	123.998	575.604
L:SORTED	4.964	397.797	600.012	109.236	251.13
L:REVERSE	0.442	0.43	600.016	3.258	600.02
L:LENGTH	4.427	232.445	600.01	197.314	571.642
L:ADD	8.843	25.73	600.004	256.356	20.817
L:APPEND	171.736	169.064	600.018	21.267	600.019
L:SUBLIST	584.989	600.031	600.012	160.673	600.017

Table 5: Execution Time in Seconds for Large Data Sets.

Test	$T_{NrS}/T_{\text{NrSFun}}$	T_{Em}/T_{NrSFun}	$T_{A^*}/T_{\text{NrSFun}}$	T_{E}/T_{NrSFun}
L:MEMBER	11.293	110.458	22.827	105.965
L:SORTED	80.136	120.873	22.006	50.59
L:REVERSE	0.973	1357.502	7.371	1357.511
L:LENGTH	52.506	135.534	44.571	129.126
L:ADD	2.91	67.851	28.99	2.354
L:APPEND	0.984	3.494	0.124	3.494
L:SUBLIST	1.026	1.026	0.275	1.026

Table 6: Relative Time Execution for Large Data Sets.

Test	*NrSFun*	*NrS*
L:MEMBER	0.9	0.85
L:SORTED	1	0.6
L:REVERSE	0.95	0.95
L:LENGTH	1	0.7
L:ADD	1	1
L:APPEND	0	0
L:SUBLIST	0.05	0

Table 7: Proportion Search Spaces Fully Exhausted.

5. Conclusions

We have provided a novel framework for non-redundant candidate construction in inductive logic programming (ILP), using propositional constraints relative to a bottom clause. In particular, we have treated the case of using search space pruning constraints, mode constraints (input-output constraints), and functional constraints, showing substantial speedups in program synthesis. Other algorithms embed pruning in some form, either implicitly through their search method—typically using refinement operators (Nienhuys-Cheng and de Wolf, 1997)—or explicitly, such as through memorization. However, they lack a mechanism to directly construct valid candidate solutions based on such constraints, leading to significant overhead in trial-and-error candidate generation.

We compared NrSample to Progol's A^* and Aleph's enumeration search. On the small program synthesis tests, NrSample outperformed both Progol A^* and enumeration on accuracy. Enumeration was marginally faster than NrSample on most tests; the exceptions are ADD, where enumeration was 2.2 times faster, REVERSE, where NrSample was 2.6 times faster, and APPEND, where NrSample was 33 times faster. NrSample was also substantially faster than A^*, ranging from 1.6 to 356 times faster.

On the large program synthesis tests, NrSample with functional constraints (*NrSFun*) always outperformed enumeration—ranging from 1 to 1358 times faster—as its naive search space traversal is often unable to find good solutions (it times out). *NrSFun* is on average 236 times faster, with substantially better accuracy on all large problems. Progol's A^* also has severe difficulties keeping up with NrSample in induction speed: *NrSFun* is on average 18 times faster than Progol's A^*, always with similar or better accuracy. Progol's A^* is only faster on two tests: APPEND and SUBLIST. However, on APPEND, Progol's A^* substantially trades accuracy for faster induction speed: accu-

racy is 71% versus 93%, for a performance gain of 8 times faster. SUBLIST is the only test in which it is a clear winner: it is 3.6 times faster with similar accuracy.

On the concept learning problems, NrSample does not seem to offer any advantage over more conventional algorithms. This is expected, as there are fewer input-output constraints to exploit. Therefore, NrSample's overhead of using a SAT solver is never compensated for. Input-output constraints can however occur in concept learning problems, and the TRAIN problem is an example as to how: it specifies that we may only attach cars in one direction, that is, from train name to its cars, not from cars to train name. With more such constraints, NrSample may offer an advantage. Moreover, it is possible to automatically generate mode declarations by inspecting the examples (McCreath and Sharma, 1995). This may create artificial but useful input-output constraints that speedup induction for arbitrary problems.

We also showed that NrSample's constraint mechanism is not easily replaced without a SAT solver: NrSample always has at least as good accuracy as its emulated counterpart, emulate_nrsample. In particular, emulate_nrsample is unable to keep up with NrSample on the large problems, consistently timing out on every data set.

Functional constraints (see Section 2.5) provide a significant performance advantage: *NrSFun* is always similar or faster than *NrS*, with similar or better accuracy. On average, *NrSFun* is 21 times faster than *NrS*.

We have shown that mode constraints are linear time solvable, whereas pruning constraints are NP-complete (see Section 3.1). Most SAT instances are—despite their NP-completeness—easy to solve in practice. As we have previously argued, the difficulty of finding non-redundant solutions is not limited to our algorithm, but rather, an inherent property of non-redundancy. Any algorithm not considering constraints may in such cases be unlikely to stumble upon a non-redundant solution.

We have shown how regional constraints—constraints used to direct the search to certain regions of the search space before others—allow for control over search order within our constraint framework (see Section 3.2).

Our constraint satisfaction approach is generalizable to problems with noise by simply modifying the definition of consistency so as to allow it to cover a user specified number of negative examples (instead of 0).

Our SAT solver is deterministic. In applications where this is undesirable, it is possible to use a non-deterministic selection strategy (for example, random literal assignment). Another source of randomness comes from setting a non-zero probability for random restarts. In both cases, pruning constraints still ensure that no redundant candidate is generated.

Acknowledgments

The work described in this paper was supported by a grant from the Research Grants Council of the Hong Kong Special Administrative Region, China [Project No. CityU 124409].

Appendix A. Proof of Mode Constraint Correctness

In this appendix we prove Theorem 8, which establishes that the mode constraints reflect Definition 5.

First, we show that the mode constraints only generate mode conformant candidates (soundness).

Theorem 15 *Let F be a propositional formula for the mode constraints of B. Let C be a candidate from B. If F generates C, then C is mode conformant.*

Proof Assume C is not mode conformant. Let $P_B = l_1 \wedge \ldots \wedge l_n$ where $l_i = b_i$ or $l_i = \neg b_i$. We have two cases: (1) C is not input-instantiated, or (2) C is not output-instantiated.

(1) A literal b_i in C has an uninstantiated input variable v. However, B is input-instantiated by Lemma 6, so $v \in I_h$, or $v \in O_k$ for some $k < i$ and b_k is not a body literal of C (since otherwise v would be instantiated). Now, $v \in I_h$ is impossible because then C would have v instantiated by its head input, as C and B have the same head. Let $k_1, \ldots, k_s$ be the indices for body literals in B for which $v \in O_{k_s}$, $k_s < i$. By Definition 7, F contains a clause $c = \{\neg b_i, b_{k_1}, \ldots, b_{k_s}\}$. Since C contains b_i but none of the preceding b_{k_j}, the model M_C for C has $M_C(b_i) = true$ and $M_C(b_{k_j}) = false$ for all b_{k_j}, $j = 1, \ldots, s$, so M_C does not satisfy c. Hence M_C is not a model for F.

(2) The head of C has an uninstantiated output variable v. Partition $b_1, \ldots, b_n$ into two sets: B_v for the b_k's that satisfy $v \in O_k$, and $B_{\neg v}$ for the rest. All literals of P_C are in $B_{\neg v}$, so P_C is a conjunction of literals where each literal of B_v is negative. Hence for the model M_C for P_C, $M_C(b_i) = false$ for all $b_i \in B_v$. But by Definition 7, F contains the clause $\bigvee_{b_i \in B_v} b_i$, so M_C is not a model for F. ■

Next, we show that the mode constraints can generate any mode conformant candidate (completeness).

Theorem 16 *Let F be a propositional formula for the mode constraints of B. Let C be a candidate from B. If C is mode conformant, then F generates C.*

Proof Assume C is mode conformant and let M_C be the model for $P_C = l_{s_1} \wedge \ldots \wedge l_{s_k}$. Then $M_C(l_{s_i}) = true$ for all $i = 1, \ldots, k$. Let f be a clause in F. Either f has the form $\{\neg b_n, b_{x_1}, b_{x_2}, \ldots\}$ or $\{b_{x_1}, b_{x_2}, \ldots\}$.

The first form appears when b_n has an input variable v that is not an output of B's head. b_{x_j} are the literals preceding b_n ($x_j < n$) in which v appears as output. If b_n does not appear in C, $M_C(b_n) = false$ and the clause is satisfied. If b_n appears in C, we note that since C is mode conformant, v appears in a previous body literal or the head. But it cannot appear in the head of C, since it is the same as the head of B. So v must be the output of a literal in C that appears before b_n, that is, one of the b_{x_j}. Hence $M_C(b_{x_j}) = true$ for this particular j, and the clause is satisfied.

The second form appears when the head of B has an output variable v. If the clause is empty, B has no body literal that outputs v, so no candidate from B is mode conformant either, contradicting our assumption that C is mode conformant. Now v appears in the head of C, again because it is the same as the head of B. Let $b_{x_1}, \ldots, b_{x_k}$ be the literals of the non-empty clause. Since C is mode conformant, v appears as output in a literal b_{x_j}, $j = 1, \ldots, k$, and hence $M_C(b_{x_j}) = true$. Therefore the clause is satisfied. ■

Appendix B. Proof of Pruning Constraint Correctness

In this appendix, we prove Theorem 12, establishing the correctness of NrSample's pruning constraints.

The next lemma will be needed in our correctness proof.

Lemma 17 *Let C and D be candidates from B. If $C \subseteq D$, all the negative literals of P_D occur in P_C.*

Proof If $C \subseteq D$, all the body literals of C occur in D, so $b_j \in P_C \Longrightarrow b_j \in P_D$. Let $\neg b_i \in P_D$. If $b_i \in P_C$, then $b_i \in P_D$, a contradiction. Since P_C contains all literals of $b_1, \ldots, b_n$, we have $\neg b_i \in P_C$. ■

The following theorem shows that $\neg P_{\uparrow C}$ and $\neg P_{\downarrow C}$ block the intended regions and no more.

Theorem 18 Let C be a candidate from B.

1. $\neg P_{\uparrow C}$ generates G if and only if G is not a generalization of C.
2. $\neg P_{\downarrow C}$ generates S if and only if S is not a specialization of C.

Proof (1) Assume $G \subseteq C$. If P_C has no negative literal, $\neg P_{\uparrow C} = false$, and the claim follows trivially. So assume P_C has negative literals $\bar{c_1}, \ldots \bar{c_k}$. Since $G \subseteq C$, $\bar{c_1}, \ldots \bar{c_k}$ are negative literals in P_G by Lemma 17. The model for G thus has $M_G(c_i) = false$ for all $i = 1, \ldots, k$. But $\neg P_{\uparrow C} = \{c_1, \ldots, c_k\}$, so it is false in M_G. For the converse, assume $G \not\subseteq C$. Then there is a positive literal $b \in P_G$ with $\neg b \in P_C$. Let M_G be the model for P_G: for all $g_i \in P_G$, $M_G(g_i) = true$, and for all negative $\bar{g_j} \in P_G$, $M_G(g_j) = false$. Note that $M_G(b) = true$. Since $\neg P_{\uparrow C}$ is a clause containing b , it is true under M_G.

(2) Assume $C \subseteq S$. If P_C has no positive literal, $\neg P_{\downarrow C} = false$, and the claim follows trivially. So assume P_C has positive literals $c_1, \ldots, c_k$. Assume M_S is a model for P_S. Since $C \subseteq S$, $c_1, \ldots, c_k$ are also in P_S, so $M_S(c_i) = true$ for all $i = 1, \ldots, k$. But $\neg P_{\downarrow C} = \{\neg c_1, \ldots, \neg c_k\}$, so it is false in M_S. For the converse, assume $C \not\subseteq S$. Then there is a positive literal $b \in P_C$ with $\neg b \in P_S$. Let M_S be the model for P_S: for all positive $s_i \in P_S$, $M_S(s_i) = true$, and for each negative $\neg s_j \in P_S$, $M_S(s_j) = false$. Note that $M_S(b) = false$. Since $\neg P_{\downarrow C}$ contains $\neg b$, M_S is a model for $\neg P_{\downarrow C}$. ■

Appendix C. Small Data Sets Details

Table 8 and 9 show the number of examples and mode declarations used in concept learning and program synthesis problems, respectively. Note that the number of examples in each data set is not necessarily indicative of complexity. For example, TRAIN has only 5 positive and 5 negative examples, but is harder to learn than Animals and Grammar. The complexity of a problem depends mainly on the mode declarations used for the body, as they give the set of all possible predicates to be used when constructing the bottom clause.

Appendix D. Primitive Set of Predicates

What follows are the definitions of the primitive set used in the large data set experiments (those prefixed by "L:").

```
:- modeb(1,nil(+list))?
:- modeb(1,+list = [-nonlist|-list],[functional])?
:- modeb(1,-list = [+nonlist|+list],[functional])?
:- modeb(1,listify(+nonlist,-list),[functional])?

listify(X,[X]).
```

Data Set	P	N	Mode Declarations
animal	16	42	$modeh(1, class(+animal, \#class))$ $modeb(1, has\_milk(+animal))$ $modeb(1, has\_gills(+animal))$ $modeb(1, has\_covering(+animal, \#covering))$ $modeb(1, has\_legs(+animal, \#nat))$ $modeb(1, homeothermic(+animal))$ $modeb(1, has\_eggs(+animal))$ $modeb(1, not\ has\_milk(+animal))$ $modeb(1, not\ has\_gills(+animal))$ $modeb(*, habitat(+animal, \#habitat))$ $modeb(1, class(+animal, \#class))$
grammar	14	7	$modeh(1, s(+wlist, -wlist))$ $modeb(1, det(+wlist, -wlist))$ $modeb(*, np(+wlist, -wlist))$ $modeb(*, vp(+wlist, -wlist))$ $modeb(1, prep(+wlist, -wlist))$ $modeb(1, noun(+wlist, -wlist))$ $modeb(1, verb(+wlist, -wlist))$
train	5	5	$modeh(1, eastbound(+train))$ $modeb(100, has\_car(+train, -car))$ $modeb(1, notopen(+car))$ $modeb(1, notlong(+car))$ $modeb(1, long(+car))$ $modeb(1, open(+car))$ $modeb(1, double(+car))$ $modeb(1, jagged(+car))$ $modeb(1, shape(+car, -shape))$ $modeb(1, load(+car, -shape, -int1))$ $modeb(1, wheels(+car, -int1))$ $modeb(1, infront(+car, -car))$

Table 8: Concept Learning Data Sets.

Data Set	P	N	Mode Declarations
member	100	50	$modeh(*, member(+const, +clist))$ $modeb(1, +any = \#any)$ $modeb(1, +clist = [-const \mid -clist])$ $modeb(*, member(+const, +clist))$
sorted	100	50	$modeh(1, sorted(+clist))$ $modeb(1, +const =< +const)$ $modeb(1, +clist = [-const \mid -clist])$ $modeb(1, +clist = [])$ $modeb(1, sorted(+clist))$
reverse	100	50	$modeh(1, reverse(+clist, -clist))$ $modeb(1, reverse(+clist, -clist))$ $modeb(1, +clist = [-const \mid -clist])$ $modeb(1, +clist = [])$ $modeb(1, append(+clist, [+const], -clist))$
length	100	50	$modeh(1, length(+clist, +int))$ $modeh(1, length(+clist, -int))$ $modeb(1, -intis + int + 1)$ $modeb(1, +intis\#int)$ $modeb(1, length(+clist, -int))$ $modeb(1, +int = 0)$ $modeb(1, +clist = [])$ $modeb(1, +clist = [-const \mid -clist])$
add	25	50	$modeh(*, add(+snum, +snum, -snum))$ $modeb(1, +snum = 0)$ $modeb(1, -snum = +snum)$ $modeb(1, dec(+snum, -snum))$ $modeb(1, inc(+snum, -snum))$ $modeb(1, add(+snum, +snum, -snum))$
append	100	50	$modeh(1, append(+list, +list, -list))$ $modeb(1, +list = [])$ $modeb(1, +const = +const)$ $modeb(1, +list = [-const \mid -list])$ $modeb(1, -list = [+const \mid +list])$ $modeb(1, append(+list, +list, -list))$
sublist	100	50	$modeh(1, sublist(+clist, +clist))$ $modeb(1, +clist = [-const \mid -clist])$ $modeb(1, +clist = [])$ $modeb(1, sublist(+clist, +clist))$

Table 9: Program Synthesis Data Sets.

```
nil([]).
nonlist(X) :- constant(X), X \= [].
list([]).
list([H|T]) :- nonlist(H), list(T).

:- modeb(1,inc(+number,-number),[functional])?
:- modeb(1,-number is +number + +number,[functional])?
:- modeb(1,neg(+number,-number),[functional])?
:- modeb(1,inv(+number,-number),[functional])?
:- _ is X+Y prevents _ is Y+X?
:- prevent neg(X,X)?
:- prevent inv(X,X)?

inc(X,Y) :- Y is X+1.
neg(X,Y) :- Y is -X.
inv(X,Y) :- Y is 1/X.

:- modeb(1,+number =:= +number)?
:- modeb(1,+number =\= +number)?
:- modeb(1,+number < +number)?
:- modeb(1,+number =< +number)?
:- prevent X =:= X? % reflexivity
:- X =:= Y prevents Y =:= X? % symmetry
:- prevent X =\= X?
:- X =\= Y prevents Y =\= X?
:- prevent X =< X?
```

We explain the operators prevent/1 and prevents/2 with examples. They are used during bottom clause construction, and are thus not specific to our benchmarked algorithms. Both operators are intended to prevent certain ground truths from occurring in the bottom clause (or their corresponding lifted instances). Example 17 illustrates its use. Similar ideas can be found in Fonseca et al. (2004).

Example 17 *The clause 'prevent $p(X,X)$' ensures that no bottom clause of the form $p(X,X)$ occurs, where matching is done so that X unifies with ground terms. So $p(a,b)$ is not allowed with this rule, but $p(a,a)$ is. The instance prevent $X =< X$ simply prevents trivial comparisons.*

As a matter of technicality, when using lifted bottom clauses, that is, with mode declarations, variables in the bottom clause must be treated as ground terms. For example, the prevent rule $X = X$ should not match with bottom clause literal $Y = 0$, as what we have in mind is to prevent trivial unifications of identical terms. By treating unlifted bottom clause literals (that is, ground truths), this problem does not arise.

Example 18 *With the rule 'prevent $p(X,X)$' from Example 17, the literal $p(A,B)$ is fails to match, since it is first grounded to $p(c_A,c_B)$, and unification then fails. Thus the literal $p(A,B)$ will not be prevented from appearing. On the other hand, $p(A,A)$ will be prevented, since it is treated as $p(c_A,c_A)$.*

Example 19 *The clause '$p(X,Y)$ prevents $q(Y,X)$' ensures that the literal $q(Y,X)$ does not occur* if *$p(X,Y)$ occurs as a previous literal in the bottom clause. Again, unification is used for matching, so that repeated variable occurrences matter. The instance used in our primitive set:*

```
_ is X+Y prevents _ is Y+X
```

simply exploits the commutativity of addition: if we have $X+Y$ (we do not care about the output variable), we do not need the addition $Y+X$ in the bottom clause.

References

Patrick Blackburn, Johan Bos, and Kristina Striegnitz. *Learn Prolog Now!* College Publications, 2006.

Rui Camacho. *Inducing Models of Human Control Skills using Machine Learning Algorithms*. PhD thesis, SciVerse, Pavel Brazdil, 2000.

Stephen A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the Third Annual ACM Symposium on Theory of Computing*, STOC '71, pages 151–158, New York, NY, USA, 1971. ACM. doi: 10.1145/800157.805047. URL http://doi.acm.org/10.1145/800157.805047.

Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem-proving. *Communications of the ACM*, 5:394–397, July 1962. ISSN 0001-0782. doi: http://doi.acm.org/10.1145/368273.368557. URL http://doi.acm.org/10.1145/368273.368557.

Luc De Raedt and Jan Ramon. Condensed representations for inductive logic programming. In *Proceedings of the 9th International Conference on Principles of Knowledge Representation and Reasoning*, pages 438–446. AAAI Press, 2004.

William F. Dowling and Jean H. Gallier. Linear-time algorithms for testing the satisfiability of propositional horn formulae. *Journal of Logic Programming*, 1(3):267–284, 1984.

Pierre Flener and Serap Yılmaz. Inductive synthesis of recursive logic programs: Achievements and prospects. *Journal of Logic Programming*, 41:141–195, 1999.

Pierre Flener, Kung-Kiu Lau, Mario Ornaghi, and Julian Richardson. An abstract formalization of correct schemas for program synthesis. *Journal of Symbolic Computation*, 30(1):93–127, 2000.

Nuno A. Fonseca, Vítor Santos Costa, Fernando M. A. Silva, and Rui Camacho. On avoiding redundancy in inductive logic programming. In *ILP*, pages 132–146, 2004.

Jérôme Maloberti and Michèle Sebag. Fast theta-subsumption with constraint satisfaction algorithms. *Machine Learning*, 55(2):137–174, May 2004. ISSN 0885-6125. doi: 10.1023/B:MACH.0000023150.80092.40. URL http://dx.doi.org/10.1023/B:MACH.0000023150.80092.40.

Eric McCreath and Arun Sharma. Extraction of meta-knowledge to restrict the hypothesis space for ILP systems. In *Proceedings of the Eighth Australian Joint Conference on Artificial Intelligence*, pages 75–82. World Scientific, 1995.

Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: engineering an efficient SAT solver. In *Proceedings of the 38th Annual Design Automation Conference*, DAC '01, pages 530–535, New York, NY, USA, 2001. ACM. ISBN 1-58113-297-2. doi: http://doi.acm.org/10.1145/378239.379017. URL http://doi.acm.org/10.1145/378239.379017.

Stephen Muggleton and Alireza Tamaddoni-Nezhad. QG/GA: a stochastic search for Progol. *Machine Learning*, 70:121–133, March 2008. ISSN 0885-6125. doi: 10.1007/s10994-007-5029-3. URL http://dl.acm.org/citation.cfm?id=1331425.1331452.

Stephen Muggleton, Luc De Raedt, David Poole, Ivan Bratko, Peter A. Flach, Katsumi Inoue, and Ashwin Srinivasan. ILP turns 20 - biography and future challenges. *Machine Learning*, 86(1): 3–23, 2012.

Stephen H. Muggleton. Inverse Entailment and Progol. *New Generation Computing*, 13:245–286, 1995. URL http://www.doc.ic.ac.uk/~shm/Papers/InvEnt.ps.gz.

Shan-Hwei Nienhuys-Cheng and Ronald de Wolf. *Foundations of Inductive Logic Programming*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1997. ISBN 3540629270.

Gordon D. Plotkin. A note on inductive generalization. *Machine Intelligence*, 5:153–163, 1970.

Gordon D. Plotkin. A Further Note on Inductive Generalization. *Machine Intelligence*, 6:101–124, 1971.

Stuart J. Russell, Peter Norvig, John F. Candy, Jitendra M. Malik, and Douglas D. Edwards. *Artificial intelligence: a modern approach*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1996. ISBN 0-13-103805-2.

Thomas J. Schaefer. The complexity of satisfiability problems. In *Proceedings of the Tenth Annual ACM Symposium on Theory of Computing*, STOC '78, pages 216–226, New York, NY, USA, 1978. ACM. doi: 10.1145/800133.804350. URL http://doi.acm.org/10.1145/800133.804350.

Michèle Sebag and Céline Rouveirol. Constraint inductive logic programming. In Luc De Raedt, editor, *Advances in ILP*. IOS Press, 1996.

Bart Selman, Henry Kautz, and Bram Cohen. Local search strategies for satisfiability testing. In *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, pages 521–532, 1995.

Mathieu Serrurier, Henri Prade, and Gilles Richard. A Simulated Annealing Framework for ILP. In Rui Camacho, Ross D. King, and Ashwin Srinivasan, editors, *ILP*, volume 3194 of *Lecture Notes in Computer Science*, pages 288–304. Springer, 2004. ISBN 3-540-22941-8.

Ashwin Srinivasan. *The Aleph Manual*, 2001. URL http://web.comlab.ox.ac.uk/oucl/research/areas/machlearn/Aleph/. http://web.comlab.ox.ac.uk/oucl/research/ areas/machlearn/Aleph/.

Leon Sterling and Ehud Shapiro. *The art of Prolog (2nd ed.): advanced programming techniques*. MIT Press, Cambridge, MA, USA, 1994. ISBN 0-262-19338-8.

Alireza Tamaddoni-Nezhad and Stephen Muggleton. The lattice structure and refinement operators for the hypothesis space bounded by a bottom clause. *Machine Learning*, 76(1):37–72, July 2009. ISSN 0885-6125. doi: 10.1007/s10994-009-5117-7. URL http://dx.doi.org/10.1007/s10994-009-5117-7.