A New Heuristic to use Least Generalizations in ILP

Esra Erdem
Department of Computer Sciences
The University of Texas at Austin, Austin, TX 78712, USA
Email: esra@cs.utexas.edu

Pierre Flener
Computing Science Department
Uppsala University, Box 311, S-75105 Uppsala, Sweden
Email: pierref@csd.uu.se

Abstract

The classical least generalization of a set $\mathcal{C}$ of clauses is a *single* clause. Such a unique clause is sometimes too general wrt some over-generality criterion, and this undesired situation may lead to memorization of the clauses in $\mathcal{C}$ at the end of learning process. Current ILP systems apply some heuristic algorithms to refine this situation where the classical definition of least generalization is used as an operator in a search space throughout the learning.

Our approach to this problem is different: we use the classical least generalization of $\mathcal{C}$ in a more declarative way to narrow the search-space at the very beginning of the learning algorithm, that is, we define the concept of least generalization of a set $\mathcal{C}$ of clauses in a different way as being a minimal-sized *set* of clauses, each member of this set being the classical least generalization of some subset of $\mathcal{C}$. The elements of these subsets are two by two *compatible*, in the sense that their classical least generalizations are not too general according to the over-generality criterion *admissibility*. We give an efficient algorithm for computing this re-defined concept, providing thus a new approach to disjunctive concept learning. We also prove a few theorems relating the problem-independent concept of compatibility with the problem-specific concept of admissibility: these theorems show how to speed up certain computations for these concepts. Finally, we outline an application of these considerations to a frequently occurring problem in the inductive synthesis of recursive (logic) programs, namely the induction of the operator combining the partial results (obtained through recursion) into the overall result.

1 Introduction

The classical definition of the concept of least general generalization (or simply: least generalization) of a clause set is that it is a *single* clause. Such a unique clause is sometimes too general wrt some over-generality criterion. If we choose θ-subsumption as underlying generality model, the least generalization under θ-subsumption (or lg-θ) of the clauses in Figure 1 is $combine(X,T,V) \leftarrow$, which is over-general if *combine* is meant to fit into the program presented in Figure 2 where $delOddElems(L,R)$ holds iff list R is integer-list L without its odd elements. In fact, the *combine* relation should be defined as shown in Figure 3.

$combine(1,[\,],[\,]) \leftarrow odd(1)$ (E_1)
$combine(2,[\,],[2]) \leftarrow even(2)$ (E_2)
$combine(3,[4],[4]) \leftarrow odd(3)$ (E_3)
$combine(6,[8],[6,8]) \leftarrow even(6)$ (E_4)

Figure 1: A set of clauses describing *combine*

$delOddElems([\,],[\,]) \leftarrow$
$delOddElems([HL|TL],R) \leftarrow delOddElems(TL,TR), combine(HL,TR,R)$

Figure 2: A definition for *delOddElems*

$combine(I,L,L) \leftarrow odd(I)$
$combine(I,L,[I|L]) \leftarrow even(I)$

Figure 3: A definition for *combine*

Most of the ILP systems apply some heuristic algorithms to find not-too-general definitions. For instance in GOLEM [cite info] a greedy covering algorithm is applied: the set $\mathcal{R}$ of lgg's of each pair of random sampling of positive examples are found wrt a background theory, then a subset $\mathcal{S}$ of $\mathcal{R}$ that covers the most number of positives without covering any negative examples is selected, and $\mathcal{S}$ is further generalized considering a random sampling of the uncovered positives. Therefore, the concept of least generalizations is used as an operator in a search space of a random sampling of positive examples. We suggest a more declarative way of using the concept of least generalizations: we introduce a new definition of the concept of least generalizations of a set of clauses.

Each of the clauses in Figure 3 is the classical lgθ of a subset of the given clause set. Thus the idea of re-defining the notion of least generalization (under θ-subsumption) of a clause set $\mathcal{C}$ as being a minimal-sized *set* of clauses, each member of this set being the classical least generalization of some subset of $\mathcal{C}$, is motivated by disjunctive predicate learning.

It is important to note that there are some relationships among the arguments of *combine* in Figure 1: the first two arguments are mandatorily used to construct the third argument. This is related to the syntactic and semantic construction of *combine* that is captured by a *construction mode* [4] that states it is mandatory to use the first and second arguments to construct the third argument of *combine*. Informally, a *construction mode* states which parameters are constructed from which other parameters, and also expresses whether such construction is mandatory or optional. In this sense, a *construction mode* is different from an input mode because a *construction mode* is a declarative notion whereas an input mode is an operational notion.

To use construction modes effectively with our definition of least generalization, we will use the powerful over-generality criterion, *admissibility* wrt a construction mode, introduced in [4]. The concept of *admissibility* wrt a construction mode captures what it means for a definite clause to satisfy the given construction mode for the relation in its head.

We will use the language of [4] designed for expressing construction modes, and the powerful admissibility criterion based on it. We will also prove a few theorems relating the problem-independent concept of compatibility with the problem-specific concept of admissibility: these theorems show how to speed up certain computations for these concepts.

In the rest of this paper, we thus first briefly study the notions of construction mode and admissibility in Section 2, then introduce the notion of compatibility in Section 3. In Section 4, we define the concept of the least generalization of a clause set in a different way and we give an efficient algorithm for computing it. In Section 5, we discuss an interesting application of the presented material, for a frequently occurring problem in the inductive synthesis of recursive (logic) programs, namely the induction of the operator combining the partial results (obtained through recursion) into the overall result. Finally, in Section 6, we review related work, outline future work, and conclude.

2 Construction Modes and Admissibility

Informally, a construction mode for a relation states which parameters are constructed from which other parameters, and also expresses whether such construction is mandatory or optional. For instance, in *append*, the third parameter is mandatorily constructed from the first two parameters.

Contrary to the well-known input modes (or call modes), there is no notion of different usages of a relation according to construction modes. Indeed, construction modes are a declarative notion, whereas input modes are an operational notion. However, a construction mode may not be unique for a given relation, because there may be several ways of expressing its data-flow. For instance, in *reverse*, the second parameter is mandatorily constructed from the first parameter, or vice-versa.

A construction mode m for r is here written in the more suggestive form $r(m(1), \ldots, m(n))$ where $m(i)$ stands for the modes *must*, *res*, and *may*, each $m(i)$ corresponds to the i'th argument of m, and $1 \leq i \leq n$. The intended semantics of a construction mode is thus as follows:

- mode *res* means the parameter in the corresponding position is the *result parameter* to be constructed from the other parameters;
- mode *must* means the parameter in the corresponding position is mandatory in constructing the parameter in the corresponding position of *res*;
- mode *may* means the parameter in the corresponding position is optional for constructing the parameter in the corresponding position of *res*.

For instance, $combine(must, must, res)$ is a mode (for *combine* of Figure 3), meant to express that the third parameter is a result parameter that must be constructed from the first two parameters. Note that $r(may, must)$ is not a mode, because no parameter is being designated as the result of construction from the two given parameters. Also, $r(res)$ is not a mode, because it does not indicate from what parameters the given result parameter must or may be constructed.

The concept of admissibility captures what it means for an *atom* to satisfy a construction mode for its relation. Before we give the relationship with admissibility and construction modes, we will introduce a few necessary concepts.

Let $r(t_1, \ldots, t_n) \leftarrow \mathcal{B}$ be a definite clause, where $\mathcal{B}$ is a conjunction of atoms, called the *body* of the clause, and $r(t_1, \ldots, t_n)$ is called the *head* of the clause. It is crucial that body $\mathcal{B}$ does not contain any equality atoms, because otherwise insufficient structure would be in the parameters in the head. For instance, instead of $insert(X, [Y|L], R) \leftarrow X \leq Y, R = [X, Y|L]$, we prefer $insert(X, [Y|L], [X, Y|L]) \leftarrow X \leq Y$.

As defined in [3], we refer to an equality-free definite clause as a *proper clause*, and two proper clauses defining the same concept as *reconcilable clauses.* The *leaves* of a term t, denoted by $leaves(t)$, are the set of the variables and constants in t. The *vertices* of a term t, denoted by $vertices(t)$, are the multi-set of the variables and functions (including the constants) in t.

For instance, $leaves(1 \cdot B \cdot 1 \cdot nil) = \{1, B, nil\}$, and $leaves(a \cdot T) = \{a, T\}$, whereas $vertices(1 \cdot B \cdot 1 \cdot nil) = \{1, \cdot, B, \cdot, 1, \cdot, nil\}$, and $vertices(a \cdot T) = \{a, \cdot, T\}$.

Now, here is our (here suitably simplified) definition of admissibility [3]:

Definition 1 (Clause admissibility and clause set admissibility)
A proper clause $r(t_1, \ldots, t_n) \leftarrow \mathcal{B}$ is *admissible* wrt a mode m for r if the following conditions hold:

$$vertices(Must) \sqsubseteq vertices(\langle Res, \mathcal{B}' \rangle)$$

$$leaves(Res) \subseteq leaves(\langle May, Must, \mathcal{B}', 0, nil, \ldots \rangle)$$

where $\sqsubseteq$ denotes multi-set inclusion, and $\mathcal{B}'$ is a tuple built of the atoms (seen as terms) of $\mathcal{B}$.
A set of reconcilable clauses is *admissible* wrt a mode m for the relation in their heads if each of its clauses is admissible wrt m.

For instance, the clause $min(X, Y, X) \leftarrow X \leq Y$ is admissible wrt $min(must, must, res)$, the clause $insert(X, [Y|L], [X, Y|L]) \leftarrow X \leq Y$ is admissible wrt $insert(must, must, res)$, and the clause $addPlateau(a, [\ ], [a, s(0)]) \leftarrow$ is admissible wrt $addPlateau(may, must, res)$. However, the clause $insert(X, [Y], [X, Z]) \leftarrow$ is not admissible wrt $insert(must, must, res)$.

Note that the two conditions are more general than sub-term (i.e., sub-tree) checking. This additional generality is crucial in many cases. For instance, in atom $efface(d, [g, e, d], [g, e])$, the vertices of $[g, e]$ are a sub-multi-set of the vertices of $[g, e, d]$, but $[g, e]$ is not a sub-tree of $[g, e, d]$.

3 Compatibility and Over-Generality

Compatibility is a problem-independent concept that will be used in our new of least generalizations of clause sets. Compatibility is relative to an over-generality criterion, which is problem-specific.

Definition 2 (Over-generality criterion)
An *over-generality criterion* is a boolean function over the set of definite clauses.

For instance, admissibility of a proper clause wrt a construction mode for the relation in its head is an over-generality criterion. Another criterion could be the non-coverage of any given negative examples, or even the combination of these two criteria.

Intuitively, compatibility of two clauses is meant to capture the fact that these two clauses perform something in the same way. Since this is a rather hard to define notion, we want to approximate it by requiring that the least generalization of these two clauses also performs the same thing in the same way. Even this is hard to capture, but it can also be approximated, namely by requiring that this least generalization not be overly general. Hence the following definition:

Definition 3 (Clause compatibility)
Two reconcilable clauses c and d are *compatible* (*with each other*) wrt an over-generality criterion G if $G(lg\theta(c,d))$ holds. We also say that c is *compatible with* d wrt G, and vice-versa.

For instance, the clauses $combine(2,[\,],[2]) \leftarrow even(2)$ and $combine(6,[8],[6,8]) \leftarrow even(6)$ are compatible wrt admissibility wrt $combine(may, must, res)$, because their lgθ, namely $combine(I,L,[I|L]) \leftarrow even(I)$, is admissible wrt that mode.

When a mode m or an over-generality criterion G has been clearly stated in context, we often drop the qualifiers "wrt mode m" (for admissibility) and "wrt criterion G" (for compatibility). If the chosen over-generality criterion is admissibility wrt some mode m, then we often say "compatible wrt m" instead of "compatible wrt admissibility wrt m".

In order to be completely general, we do not impose any constraints on the compatibility relation. It is symmetric (by construction), but we do not require it to be reflexive nor transitive.

For instance, compatibility wrt admissibility is not reflexive, because the definition does not require the two given clauses to be admissible wrt the given mode. For instance, the clause $q(d,[g],[e,g]) \leftarrow$ is not admissible wrt $q(may, must, res)$, therefore it is not compatible with itself (because its lgθ with itself is itself). In general, reflexivity is really irrelevant for our purposes.

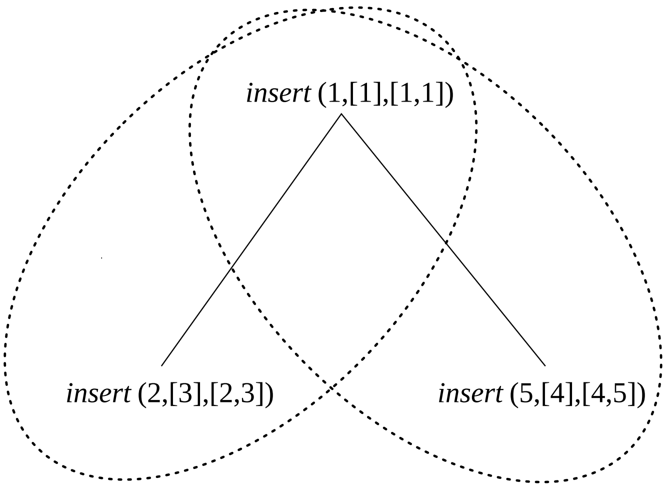
Also, compatibility wrt admissibility is not transitive. For instance, the clauses $insert(1,[1],[1,1]) \leftarrow$ and $insert(2,[3],[2,3]) \leftarrow$ are compatible wrt admissibility wrt $insert(must, must, res)$, because their lgθ, namely $insert(X,[Y],[X,Y]) \leftarrow$, is admissible; also, the clauses $insert(1,[1],[1,1]) \leftarrow$ and $insert(5,[4],[4,5]) \leftarrow$ are compatible, because their lgθ, namely $insert(X,[Y],[Y,X]) \leftarrow$, is admissible; but $insert(2,[3],[2,3]) \leftarrow$ and $insert(5,[4],[4,5]) \leftarrow$ are not compatible, because their lgθ, namely $insert(X,[Y],[Z,K]) \leftarrow$, is not admissible. Upon close inspection of the desired concept of compatibility, it turns out that transitivity is often impossible to achieve without sacrificing most of the power of our new definition of least generalizations of clause sets. Indeed, if compatibility were to be transitive, then some clauses would have to be compatible (e.g., the last two clauses above), although they do not construct their res_j parameters in the same way.

So far, we have only defined the compatibility of *two* clauses, but we will also need a notion of compatibility of a non-empty *set* of clauses.

Definition 4 (Clause set compatibility)
A non-empty set $\mathcal{C}$ of reconcilable clauses is *compatible* wrt an over-generality criterion G if any two clauses in $\mathcal{C}$ are compatible wrt G.

Note that a singleton set is thus trivially compatible wrt *any* over-generality criterion. Also notice that a compatible subset of a set $\mathcal{D}$ of reconcilable clauses is a clique of the graph with node set $\mathcal{D}$ and edge set induced by the compatibility relationship over $\mathcal{D}$. Remember that a *clique* (or: *complete sub-graph*) of a graph $\langle \mathcal{N}, \mathcal{E} \rangle$ is a subset $\mathcal{N}'$ of node set $\mathcal{N}$ such that any two nodes in $\mathcal{N}'$ are joined by an edge in edge set $\mathcal{E}$. For instance, the set

$\mathcal{D} = \{insert(1,[1],[1,1]) \leftarrow \quad, insert(2,[3],[2,3]) \leftarrow \quad, insert(5,[4],[4,5]) \leftarrow \quad\}$ has the following compatibility graph wrt admissibility wrt $insert(must, must, res)$:



where the edges are induced by the compatibility relationship, and the contours identify cliques.

Note that, due to the non-required reflexivity and transitivity of compatibility, cliques are not simply the equivalence classes of an equivalence relation: this considerably complicates clique finding, as seen in Section 4.

We now establish some interesting properties of compatibility wrt admissibility. They also only hold for (sets of) proper clauses whose bodies only involve atoms for test relations.

Checking whether a given set is compatible wrt admissibility seems to be an extremely tedious operation. Fortunately, the following theorem shows how to shortcut such a decision process (for sets of at least two clauses, because, as pointed out, a singleton set is trivially compatible wrt any over-generality criterion, so that no decision procedure is needed then).

Theorem 1 (Sufficient criterion for clause set compatibility wrt admissibility)
Let $\mathcal{C}$ be a set of at least two reconcilable clauses, and let m be a mode for the relation in their heads. If $lg\theta(\mathcal{C})$ is admissible wrt m, then $\mathcal{C}$ is compatible wrt m.

Proof: Let $lg\theta(\mathcal{C})$ be admissible wrt m. Let c and d be two arbitrary clauses in $\mathcal{C}$. By construction, $lg\theta(\mathcal{C})$ θ-subsumes $lg\theta(c,d)$. So, by Lemma 1 of [4], $lg\theta(c,d)$ is admissible wrt m. That is, by Definition 3 and since c and d are reconcilable, clauses c and d are compatible wrt m. Since c and d were chosen arbitrarily, set $\mathcal{C}$ is compatible wrt m, by Definition 4. □

Note the following useful property linking the two $lg\theta$ operators:

Property 1 If $c \in \mathcal{C}$, then $lg\theta(lg\theta(\mathcal{C}), c) = lg\theta(\mathcal{C})$.

The following theorem shows that the least generalization (under θ-subsumption) of a clause set is compatible with every element of that set. (This is trivial for singleton sets, so we restrict the theorem to sets of at least two elements, so that the proof can use the previous theorem.)

Theorem 2 (Representative of a compatible clause set)
Let $\mathcal{C}$ be a set of at least two reconcilable clauses, and let m be a mode for the relation in their heads. We have that $\mathcal{C}$ is compatible wrt m iff $\mathcal{C} \cup \{lg\theta(\mathcal{C})\}$ is compatible wrt m.

Proof: First assume that $\mathcal{C}$ is compatible wrt m. Then we have that:
$\mathcal{C} \cup \{lg\theta(\mathcal{C})\}$ is compatible wrt m
iff any clause $c \in \mathcal{C}$ is compatible with $lg\theta(\mathcal{C})$ wrt m (by Definition 4 and by the assumption)
iff $lg\theta(lg\theta(\mathcal{C}), c)$ is admissible wrt m (by Definition 3)
iff $lg\theta(\mathcal{C})$ is admissible wrt m (by $c \in \mathcal{C}$ and Property 1)
implies $\mathcal{C}$ is compatible wrt m (by Theorem 1)

iff true (by the assumption).
Conversely, assume that $\mathcal{C} \cup \{lg\theta(\mathcal{C})\}$ is compatible wrt m. Then we immediately have that $\mathcal{C}$ is compatible wrt m (as any subset of a set compatible wrt m is compatible wrt m). $\Box$

This basically means that the lgθ of a compatible set can be taken as a representative of that set, because it is compatible with *each* element in that set.

4 A New Definition of the Least Generalization of a Clause Set

Given a set $\mathcal{C}$ of definite clauses and an over-generality criterion G, we now want to define a (new) concept of least generalization of $\mathcal{C}$ wrt G. Basically, we want a *set* $\mathcal{L}$ of clauses as the least generalization. The idea of the defining least generalization of a clause set comes from Theorem 2: one can divide $\mathcal{C}$ into a minimal number of subsets that are compatible wrt G, and then take the set of their classical least generalizations as representatives, i.e., as $\mathcal{L}$.

Let us first get a feeling for the desired concept and algorithm, via some examples.

Example 1 Let $\mathcal{C}$ be the clause set of Figure 1. The entire set $\mathcal{C}$ is not compatible wrt $combine(must, must, res)$, because, for instance, clauses E_1 and E_4 are not compatible, since $lg\theta(E_1, E_4)$, namely $combine(X, T, V) \leftarrow$, is not admissible. So we need a division of $\mathcal{C}$ into a minimal number of subsets such that each one is compatible. In this case, the subsets $\mathcal{C}_1 = \{E_1, E_3\}$ and $\mathcal{C}_2 = \{E_2, E_4\}$ are compatible, because both $lg\theta(E_1, E_3)$, which is $combine(X, T, T) \leftarrow odd(X)$, and $lg\theta(E_2, E_4)$, which is $combine(X, T, [X|T]) \leftarrow even(X)$, are admissible. Therefore, we get the least generalization as shown in Figure 3 which is a partially correct and complete program for $combine$. $\Box$

Example 2 Let $\mathcal{I}$ be the following clause set for $insert$:

$$\begin{array}{ll} insert(1, [1], [1, 1]) \leftarrow & (E_1) \\ insert(2, [3], [2, 3]) \leftarrow & (E_2) \\ insert(5, [4], [4, 5]) \leftarrow & (E_3) \end{array}$$

The entire set $\mathcal{I}$ is not compatible wrt $insert(must, must, res)$, because, for instance, clauses E_2 and E_3 are not compatible, since $lg\theta(E_2, E_3)$, namely $insert(X, [Y], [Z, T]) \leftarrow$, is not admissible. So we need a division of $\mathcal{I}$ into a minimal number of subsets such that each one is compatible. In this case, the subsets $\mathcal{I}_1 = \{E_1, E_2\}$ and $\mathcal{I}_2 = \{E_1, E_3\}$ are compatible, because both $lg\theta(E_1, E_2)$, which is $insert(X, [Y], [X, Y]) \leftarrow$, and $lg\theta(E_1, E_3)$, which is $insert(X, [Y], [Y, X]) \leftarrow$, are admissible. Therefore, we get the following least generalization:

$$\begin{array}{l} insert(X, [Y], [X, Y]) \leftarrow \\ insert(X, [Y], [Y, X]) \leftarrow \end{array}$$

which is a partially correct (but incomplete) program for $insert$. $\Box$

In these two examples, while we are trying to find some subsets of the given clause set such that they are compatible wrt the given mode, we are actually trying to find the minimal number of cliques of the compatibility graph over this set such that these cliques contain all the given clauses. Note that $\mathcal{I}_1$ and $\mathcal{I}_2$ do not form a partition in Example 2, while $\mathcal{C}_1$ and $\mathcal{C}_2$ do form a partition in Example 1. Actually, in Example 2, we could have used a partition such as $\mathcal{I}'_1 = \{E_2\}$ and $\mathcal{I}'_2 = \{E_1, E_3\}$, but a cover such as $\mathcal{I}_1 = \{E_1, E_2\}$ and $\mathcal{I}_2 = \{E_1, E_3\}$ is better, because the more evidence in a subset, the more accurate its least generalization is. Therefore, in general now, what we try to find is a *node clique cover* (NCC) of the compatibility graph of the given clause set wrt a given over-generality criterion. Finding a node clique cover of a graph is referred to as the *node clique cover problem* (or shortly *clique cover problem*) in graph theory: we try to find the minimum number of cliques that cover the nodes of the given undirected graph, such that a node already covered by a clique may also be covered by another clique [13]. This is an NP-complete problem. Note that we do not try to find a minimum set cover, i.e., the smallest subset $\mathcal{S}$ of the given subsets of a finite set $\mathcal{U}$ such that the union of the members of $\mathcal{S}$ is equal to $\mathcal{U}$, which is a more general problem. Indeed, finding an NCC fits our goal better, in the sense that it is more natural to represent the compatibility relation wrt the given over-generality criterion over the clause set by a graph (as edges) rather than by a set; and this makes finding an NCC more efficient than finding a minimal set cover.

Now we can introduce our definition of the concept of least generalization of a set of clauses:

Definition 5 (Least generalization, under θ-subsumption, of a clause set, wrt an over-generality criterion)
The *least generalization under θ-subsumption* of a non-empty set $\mathcal{C}$ of clauses wrt an over-generality criterion G, denoted by $lg\theta(\mathcal{C}, G)$, is the set of classical least generalizations under θ-subsumption of the node cliques of the graph $\langle \mathcal{C}, \mathcal{E}_{\mathcal{G}} \rangle$ with node set $\mathcal{C}$ and edge set $\mathcal{E}_{\mathcal{G}}$ induced by the compatibility relation wrt G where the set of node cliques is a minimal cover of the graph $\langle \mathcal{C}, \mathcal{E}_{\mathcal{G}} \rangle$.

It remains to see how to (efficiently) solve the NCC problem. The NCC problem being an NP-complete problem, there does not exist a polynomial time algorithm for it, assuming $P \neq NP$. We approach NP-complete problems in two ways [10]:

- try to find as much improvement as possible over straightforward exhaustive search to find the optimal solution, such as by using branch-and-bound or dynamic programming;

- try to find a "good" solution within an acceptable amount of time, using problem-specific methods, called heuristics, such as neighborhood search, in which, using a pre-selected set of local operators, an initial solution is improved repeatedly until a "local optimum" solution, i.e., a "good" solution, has been obtained.

We used the first approach in [2], but it was very inefficient in time, so we now choose the second approach, that is, we use an approximation algorithm for the NCC problem.

The literature has a number of heuristic algorithms for the graph coloring problem, from which we can construct algorithms for the NCC problem. In [13], an algorithm for *edge clique covering* (ECC) is constructed without using the heuristics for graph coloring, but rather a heuristic introduced by Kellerman for the keyword conflict problem. We used the approximation algorithm introduced in [13] for the ECC problem to construct an approximation algorithm for the NCC problem. Note that there is an approximation algorithm for the NCC problem iff there is an approximation algorithm for the ECC problem [13].

We now switch the terminology to our particular problem, and talk about clauses instead of nodes, and about compatibility (wrt an over-generality criterion G) instead of connectedness via an edge. The approximation algorithm forms the minimal number of cliques, labeled $C_1, \ldots, C_p$, examining the clauses one by one. In the description of the algorithm, $CurrentClause$ is the label of the current clause being examined, and k is the number of the cliques that have been created so far. The heuristic (proposed by Kellerman) is that when a clause labeled i is being examined, the next clauses to be examined are the ones that are compatible with i wrt G, and labeled j, where $j < i$. This heuristic is improved in [13], adding one additional pass over all of the cliques to eliminate the cliques subsumed by the others. In the description of the algorithm, $ClausesToBeExamined$ is the set of the labels of the next clauses to be examined when the clause labeled $CurrentClause$ is being examined. The approximation algorithm is introduced in Figure 4.

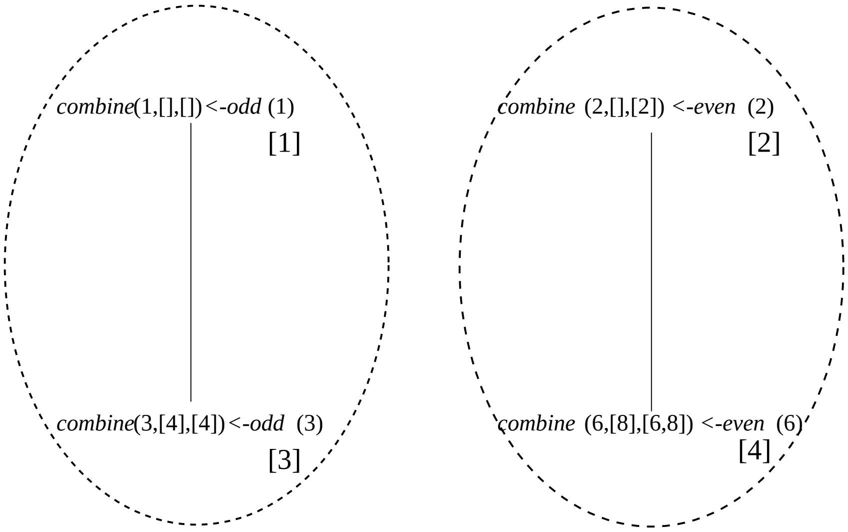
Example 3 Let us analyze how this algorithm works from the clause set $\mathcal{C}$ of Figure 1. Let $combine(must, must, res)$ be the construction mode.

Initially, all clauses in $\mathcal{C}$ are labeled starting from 1 as in the figure below:

Algorithm $lg\theta$
Inputs:
– a set $\mathcal{C}$ of n clauses, whose elements are labeled from 1 to n
– an over-generality criterion G
Output:
– $lg\theta(\mathcal{C}, G)$

```
Initialize the number of cliques: k ← 0;
Initialize the label of the current clause being examined: CurrentClause ← 1;
while CurrentClause ≤ n do
  begin
    ClausesToBeExamined ← {j | j < CurrentClause ∧ compatible(CurrentClause, j, G)};
    if ClausesToBeExamined = {} then
      begin {Create a new clique}
        k ← k + 1;
        C_k ← {CurrentClause};
        CurrentClause ← CurrentClause + 1
      end ;
    else {Try to insert CurrentClause into existing cliques C_1, ..., C_k}
      begin
        Initialize the clique into which CurrentClause may be inserted: l ← 1;
        Initialize the union of cliques into which CurrentClause is inserted: V ← {};
        while l ≤ k and V ≠ ClausesToBeExamined do
          begin
            if C_l ⊆ ClausesToBeExamined then
              begin
                C_l ← C_l ∪ {CurrentClause};
                V ← V ∪ C_l
              end ;
            l ← l + 1;
          end ;
        Update ClausesToBeExamined to account for those clauses that were covered by the
        cliques into which CurrentClause is inserted:
        ClausesToBeExamined ← ClausesToBeExamined \ V;
        while ClausesToBeExamined ≠ {} do
          begin {Add new cliques}
            Find the smallest l, with 1 ≤ l ≤ k, such that |C_l ∩ ClausesToBeExamined| is maximal:
            k ← k + 1;
            C_k ← (C_l ∩ ClausesToBeExamined) ∪ {CurrentClause};
            ClausesToBeExamined ← ClausesToBeExamined \ C_l
          end ;
        CurrentClause ← CurrentClause + 1
      end
  end ;
Suppose C_1, ..., C_k are the cliques produced so far: examine them one by one to see if the
clauses covered by a clique are also covered by a subset of the union of the remaining cliques:
if a clique is subsumed by the union of the remaining cliques, then eliminate it;
p ← the number of remaining cliques
return {lgθ(C_1), ..., lgθ(C_p)}
```

Figure 4: An efficient algorithm for the least generalizations of a set of clauses



At the beginning, there is no clique. We traverse all of the clauses in label order in order to check whether any of them is already in a clique or requires a new clique to be created.

First, we examine clause 1: as there are no clauses labeled less than 1, there is no clique that clause 1 can be inserted into. Therefore, we create a new clique $\mathcal{C}_1 = \{1\}$.

Next, we examine clause 2: there is one clause labeled less than 2, namely clause 1. However, clause 1 is not compatible with clause 2, so we cannot insert clause 2 into an existing clique. Hence, we create a new clique $\mathcal{C}_2 = \{2\}$.

Then, we examine clause 3: there are two clauses labeled less than 3, namely clauses 1 and 2. However, only clause 1 is compatible with clause 3. As all clauses labeled less than 3 are already in some existing cliques, if any of them is compatible with clause 3, then clause 3 can be inserted into a clique as well. Therefore, we eliminate clause 2 as a candidate, as it is not compatible with clause 3, and we try to insert clause 3 into existing cliques that cover clause 1, as clause 1 is compatible with clause 3. Hence, we insert clause 3 into $\mathcal{C}_1$, giving $\mathcal{C}_1 = \{1, 3\}$.

Finally, we examine clause 4: there are three clauses labeled less than 4, but only clause 2 is compatible with clause 4. Therefore, we try to insert clause 4 into existing cliques that cover clause 2; so we insert clause 4 into $\mathcal{C}_2$, giving $\mathcal{C}_2 = \{2, 4\}$.

As there is no other clause to be examined, and as neither $\mathcal{C}_1$ nor $\mathcal{C}_2$ is unnecessary to form a clique cover, the final cliques are $\mathcal{C}_1 = \{1, 3\}$ and $\mathcal{C}_2 = \{2, 4\}$. The final result is the set of classical $lg\theta$s of these cliques introduced in Figure 3 which is indeed the clause set aimed at in Figure 1. □

Both the heuristic proposed by Kellerman and the suggested improvement of this heuristic are polynomial-time [13]. Note that the outer while-loop implements the heuristic proposed by Kellerman, and that the last step implements the improvement over this heuristic. Hence, this approximation algorithm is polynomial-time.

5 An Application: The Program Closing Method

In a companion paper [3], we apply construction modes, admissibility, and the here re-defined notion of least generalization (wrt admissibility) of a clause set in order to provide a novel solution to part of the problem of inductive synthesis of recursive (logic) programs from incomplete specifications. Indeed, every now and then, inductive synthesizers appear, having the *basic synthesis algorithm* of [8], given clausal evidence for a top-level relation r.

Some synthesizers of this category are THESYS [20], *BMWk* [12, 16], SYNAPSE [7, 5], LOPSTER [14], CILP [15], CRUSTACEAN [1], METAINDUCE [11], DIALOGS [6, 8], etc. We here ignore the mechanics of Steps 1 and 2: there are various ways of achieving the results reported here (or similar ones) and we invite the reader to accept them as such, because our focus is on Step 3. Let us illustrate all this on an example.

Example 4 Take the informal specification: $delOddElems(L, R)$ holds iff list R is integer-list L without its odd elements. A specifier could give the following (formal) specification by incomplete clausal evidence:

$delOddElems([\,],[\,]) \leftarrow$
$delOddElems([1],[\,]) \leftarrow odd(1)$ $\qquad\qquad \leftarrow delOddElems([5],[5])$
$delOddElems([2],[2]) \leftarrow even(2)$
$delOddElems([3,4],[4]) \leftarrow odd(3)$
$delOddElems([6,7,8],[6,8]) \leftarrow even(6)$

Suppose Step 1 creates the open program (the undefined relation is called *combine* for convenience) of Figure 2.

Suppose Step 2 abduces the clausal evidence introduced in Figure 1 for the undefined relation.

Step 3 of the THESYS, *BMWk*, LOPSTER, CILP, CRUSTACEAN, and METAINDUCE systems induces $combine(X, T, V) \leftarrow$ as generalization of the positive evidence (usually by taking the classical lgθ), which is not an appropriate result, as argued in Figure 1. In fact, the *combine* relation should be conjunctively defined as in Figure 3.

In general, these systems can only infer single unit clauses from abduced positive and negative examples of the open relation, rather than sets of definite clauses (with bodies, thus) from clausal evidence of the open relation. The SYNAPSE and DIALOGS systems can however infer the correct generalization above, since their Step 3 is based on the *Program Completion Method* [3], which is itself based on construction modes, admissibility, and the here re-defined notion of least generalization (wrt admissibility). Other features under which our *Program Completion Method* improves on the methods of these other systems are discussed in the companion paper [3]. □

6 Conclusion

The classical least generalization of a set $\mathcal{C}$ of clauses is a *single* clause. Such a unique clause is sometimes too general wrt some over-generality criterion, and this undesired situation may lead to memorization of the clauses in $\mathcal{C}$ at the end of learning process. Current ILP systems apply some heuristic algorithms such as greedy cover algorithm (as in GOLEM [cite info], FOIL, [How about PROGOL?]) to refine this situation where the classical definition of least generalization is used as an operator in a search space of random sampling of the given examples. As a random sampling is selected, and a greedy algorithm, these systems may end up with a definition which is over-specialized wrt the intended relation to be learned.

Our approach to this problem is different: we introduce a new definition for the least generalization of $\mathcal{C}$, and give an efficient approximation algorithm, which is a modified version of an approximation algorithm for node clique cover problem of graph theory, that implements this new concept.

Our definition of the notion of least generalization of a clause set is essentially independent of the chosen generality model. We just chose the simple model of θ-subsumption for illustration purposes, and explored it in some depth by deriving some useful theorems for it. A similar exercise can be done for other, more powerful generality models that may or may not be relative background knowledge, such as inverse resolution, inverse implication, etc. Note that the given algorithm for computing this new concept can be straightforwardly adapted for any other generality model: it suffices to change the last line accordingly and to re-define compatibility accordingly.

Similarly for the over-generality criterion: the mentioned admissibility wrt a construction mode is just one possibility, introduced for the sake of being able to show an application, but the reader may use the new least generalization algorithm with any over-generality criterion. For instance, non-coverage of negative examples is a commonly used such criterion.

It should be noticed that we have taken a very general position as to what constitutes evidence to be generalized: indeed, we consider definite clauses as evidence. It is obvious that the (old-fashioned) approach of evidence as positive and negative literals (or examples) is covered by ours, as it suffices to view positive examples as unit clauses and negative examples as head-less clauses with one body atom.

We do not want to present our new least generalization algorithm as one of inductive learning/synthesis, because it may not "solve" the ILP problem, or the disjunctive predicate learning problem, by itself, but only as an auxiliary to more powerful and intricate techniques. The given application inside our Program Closing Method for inductive synthesis of recursive (logic) programs is within such a highly suitable situation, because the open relations there are always of the kind where some parameters are indeed mandatorily constructed from some parameters and optionally from others, and such that this is even known in advance (by virtue of pre-determination based on the underlying program schema).

In terms of related work, there does not seem to be much, other than our own already mentioned previous work. The INDICO learning technique [19] features an interesting method for conjecturing the heads of possible clauses, hence providing already much of the discriminating information that otherwise has to be discovered together with the characterizing information when starting from most-general clause heads. The method first partitions (i.e., it does not divide) the evidence (which must be unit clauses) into subsets according to the functors (e.g., type constructors) appearing in it; then it computes the classical lgθ of each obtained subset so as to produce a series of clause heads, from which a top-down clause specialization process can then be started. This method is obviously related to, but more specialized than, our clique finding algorithm.

In terms of future work, our results regarding compatibility and re-defined least generalizations of a set of clauses are totally generic, so that any combination of generality model and over-generality criterion can be used.

Of course, (refinements of) our suggested construction modes may be used (maybe in conjunction with input modes) by other researchers, even without their having to use the new least generalization definition and algorithm, and we look forward to seeing such applications of our proposal in the ILP literature.

Acknowledgments

We thank Baudouin Le Charlier and Pierre-Yves Schobbens (both at the University of Namur, Belgium), Necip Fazıl Ayan (Bilkent University), and the Machine Learning Group at the University of Texas at Austin, for their helpful suggestions on a preliminary version of this paper. The referees' comments have also significantly contributed to the quality of this paper.

References

[1] D.W. Aha, S. Lapointe, C.X. Ling, and S. Matwin. Inverting implication with small training sets. In F. Bergadano and L. De Raedt (eds), *Proc. of ECML'94*, pp. 31–48. *LNAI* 784, Springer-Verlag, 1994.

[2] E. Erdem. *An MSG Method for Inductive Logic Program Synthesis.* Senior Project Final Report, Bilkent University, Ankara (Turkey), May 1996.

[3] E. Erdem and P. Flener. Completing Open Logic Programs by Constructive Induction. Submitted.

[4] E. Erdem and P. Flener. A New Declarative Bias for ILP: Construction Modes. *Work-in-Progress Reports of the Tenth International Conference on Inductive Logic Programming*, 2000.

[5] P. Flener. *Logic Program Synthesis from Incomplete Information.* Kluwer, 1995.

[6] P. Flener. Inductive Logic Program Synthesis with DIALOGS. In S. Muggleton (ed), *Proc. of ILP'96*, pp. 175–198. *LNAI* 1314, Springer-Verlag, 1997.

[7] P. Flener and Y. Deville. Logic program synthesis from incomplete specifications. *J. of Symbolic Computation* 15(5–6):775–805, May/June 1993.

[8] P. Flener and S. Yılmaz. Inductive synthesis of recursive logic programs: Achievements and prospects. Submitted.

[9] P. Flener, K.-K. Lau, and M. Ornaghi. On correct program schemas. In N.E. Fuchs (ed), *Proc. of LOPSTR'97*. *LNCS*, Springer-Verlag, forthcoming.

[10] M.R. Garey and D.S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W.H. Freeman and Company, 1979.

[11] A. Hamfelt and J. Fischer Nilsson. Inductive metalogic programming. In S. Wrobel (ed), *Proc. of ILP'94*, pp. 85–96. *GMD-Studien* Nr. 237, Sankt Augustin (Germany), 1994.

[12] Y. Kodratoff and J.-P. Jouannaud. Synthesizing LISP programs working on the list level of embedding. In A.W. Biermann, G. Guiho, and Y. Kodratoff (eds), *Automatic Program Construction Techniques*, pp. 325–374. Macmillan, 1984.

[13] L.'T. Kou, L.J. Stockmeyer, and C.K. Wong. Covering edges by cliques with regard to keyword conflicts and intersection graphs. *Comm. of the ACM* 21(2):135–139, Feb. 1978.

[14] S. Lapointe and S. Matwin. Sub-unification: A tool for efficient induction of recursive programs. In *Proc. of ICML'92*, pp. 273–281. Morgan Kaufmann, 1992.

[15] S. Lapointe, C.X. Ling, and S. Matwin. Constructive inductive logic programming. In S. Muggleton (ed), *Proc. of ILP'93*, pp. 255–264. Technical Report IJS-DP-6707, J. Stefan Institute, Ljubljana (Slovenia), 1993.

[16] G. Le Blanc. BMWk revisited: Generalization and formalization of an algorithm for detecting recursive relations in term sequences. In F. Bergadano and L. De Raedt (eds), *Proc. of ECML'94*, pp. 183–197. *LNAI* 784, Springer-Verlag, 1994.

[17] G.D. Plotkin. A note on inductive generalization. In B. Meltzer and D. Michie (eds), *Machine Intelligence* 5:153-163. Edinburgh University Press, Edinburgh (UK), 1970.

[18] I. Stahl. *Predicate Invention in ILP: An Overview*. Technical Report 1993/06, Fakultät Informatik, Universität Stuttgart (Germany), 1993.

[19] I. Stahl, B. Tausend, and R. Wirth. Two methods for improving inductive logic programming systems. In P. Brazdil (ed), *Proc. of ECML'93*, pp. 41–55. *LNAI* 667, Springer-Verlag, 1993.

[20] P.D. Summers. A methodology for LISP program construction from examples. *J. of the ACM* 24(1):161–175, Jan. 1977.