Induction of Logical Relations Based on Specific Generalization of Strings

Yasin Uzun
Dept. of Computer Engineering
Bilkent University
06800 Bilkent Ankara, TURKEY
Email: yasinu@cs.bilkent.edu.tr

Ilyas Cicekli
Dept. of Computer Engineering
Bilkent University
06800 Bilkent Ankara, TURKEY
Email: ilyas@cs.bilkent.edu.tr

***Abstract*—Learning logical relations from examples expressed as first order facts has been studied extensively by the Inductive Logic Programming research. Learning with positive-only data may cause over generalization of examples leading to inconsistent resulting hypotheses. A learning heuristic inferring specific generalization of strings based on unique match sequences is shown to be capable of learning predicates with string arguments. This paper describes an inductive learner based on the idea of specific generalization of strings, and the given clauses are generalized by considering the background knowledge.**

I. Introduction

Inductive Logic Programming, shortly ILP, is a relatively new research area that is between Machine Learning and Logic Programming, and inherits the techniques and theories from both disciplines. The aim of ILP research is to learn logic programs, given examples and background knowledge expressed in Horn clause logic, which correctly define a single concept or multiple related concepts. The learned logic programs are usually expressed in Prolog syntax and declarative property of logic programs is the main source of efficiency of ILP.

Common approach in state-of the art ILP paradigm is to produce general clauses from positive examples and restrict their coverage by the help of negative examples. In domains where there is positive-only data, the systems may not be able to learn the concepts correctly because of the absence of negative examples. The problem is so substantial and common that, Progol system [5] is designed to work in a different mode when there is only positive data.

One application area of ILP is learning predicates having string arguments, which can occur in many domains such as Grammar Learning and Machine Translation. The bottom-up method Least General Generalization proposed in [2] may cause overgeneralization in the clause generation in the absence of negative examples. In [1], a specific generalization (SG) of two strings is proposed to reduce overgeneralization. To compute SG, unique match sequence, which is a sequence of similarities and differences, is found in the initial step and followed by the generalization by replacing differences with variables.

Although [1] proposes a heuristic for generalization of strings, it is far from being an ILP system because of lack of background knowledge processing. The purpose of the research that is materialized in this paper was to develop an ILP system based on the specific generalization idea. We achieved this purpose by extending the specific generalization algorithm taking the background clauses into consideration. We showed that the system works effectively in several example domains.

The rest of this paper is organized as follows. Specific generalization of strings, proposed in [1] is discussed in Section 2. In Section 3, we explain the construction of an inductive learner, which we name InGen, based on the specific generalization heuristic outlined in Section 2. Experimental results are given in Section 4. Section 5 concludes the paper with future directions to study.

II. String Generalization

A. Motivation

Learning by positive-only data is a difficult task in ILP due to the possible overgeneralization caused by the lack of restriction induced by negative examples. But in real-life, we have many domains where we have only positive examples such as Grammar Learning and Machine Translation. There have been attempts [3], [4], [5] to propose a solution for learning from positive-only data such as statistical techniques using prior probabilities or closed world assumption. In closed world assumption approach, every possible ground clause not given in the positive example set is produced by the system and labeled as negative.

Predicates defined on string arguments occur in many domains such as Grammar Learning and Machine Translation. In [1], the authors propose a solution for learning predicates that have string arguments in domains having no negative examples. The proposed methodology is based on the notion of unique match sequence, which is based on similarities (subsequences occurring in both strings) and differences (subsequences differing among strings) of two strings. The unique match sequence is generalized using Plotkin's LGG schema.

Suppose we have two positive examples with predicate *endsWith* in Prolog notation, where lists represent strings:

```
endsWith([a,b], [x,y]).
endsWith([c,d,b], [w,z,y]).
```

Although these two predicates share the common property that first argument is a list ends with *b*, and second argument is a list ends with *y*, GOLEM [6], which also uses LGG schema, overgeneralizes this pair with result:

```
endsWith([A,B|C],[D,E|F]).
```

which accepts all *endsWith* predicates with list pair having length at least two as input. The output of Progol [7], which is based on similar principles with GOLEM is:

```
endsWith([a,b], [x,y]).
endsWith([c,d,b], [w,z,y]).
```

which overfits on the examples and covers nothing more. The string generalization technique proposed in [1] learns the following clause with the same example pair:

```
endsWith(L1,L2) :-
   append(X,[b],L1), append(Y,[y],L2).
```

which accepts clauses with predicate *endsWith*, and the last elements of the first and second arguments are *b* and *y*. respectively.

B. Preliminaries

The mentioned methodology makes generalizations by processing similarities and differences of strings. A match sequence is the sequence of similarities and differences between two strings. Informally, a similarity between two strings is common subsequence of symbols and a differences are the subsequences between similarities. For a string pair $(abcd, abe)$; ab is the similarity and (cd, e) represents the difference.

Although the string pair $(abcd, ecfg)$ has a single match sequence $(ab, e)c(d, f)$, the pair $(abc, dbebf)$ has two match sequences $(a, d)b(c, ebf)$ and $(a, dbe)b(c, f)$ since b appears twice in the second string.

In the article, a specific case of a match sequence, the notion of unique match sequence is defined with two additional restrictions on a match sequence:

- Symbols occurring in similarities and differences constitute two disjoint sets. This rule enforces that, a symbol occurring in one of the similarities can not occur in any difference.
- Symbols of first and second constituents of differences constitute two disjoint sets. This rule enforces that, common symbols can only occur in similarities.

These two restrictions together provide that only string pairs whose common symbols occur the same number of times in the same order can have a unique match sequence. Some examples that can clarify the notion of unique match sequence are:

- *UMS(abceb, fgbhb) = (a,fg)b(ce,h)b.*
- *UMS(ab, ab) = ab.*
- *UMS(abc, xyz) = (abc, xyz).*
- *UMS(abcb, dbebf) = (a,d)b(c,e)b(*ϵ*,f).*
- *UMS(abc, abdb) =* ϕ*.*
- *UMS(ab, ba) =* ϕ*.*

The authors introduce the notions of separable and separation differences are to provide further capturing of similar patterns. In short, difference (D_1, D_2) is said to be separable by difference (d_1, d_2) if d_1 and d_2 occur the same number of times and greater than zero in D_1 and D_2, respectively. We say that a difference (D_1, D_2) is divided by another difference

if ($ums(\alpha_1,\alpha_2)$ does not exist)
  There is no possible generalization
else
  $SIofUMS \leftarrow ums(\alpha_1, \alpha_2)$
  while (there is a *MUSD* that separates
      $SIofUMS$ with factor ≥ 2)
    $SIofUMS \leftarrow separation(SIofUMS, MUSD)$
  $SG \leftarrow InverseSubsitute(SIofUMS)$

Fig. 1. Finding Specific Generalization

(d_1, d_2) with separation factor *n* where *n* is the number of times d_1 occurs in D_1 and d_2 occurs in D_2.

For instance, the difference *(aba,cdc)* is separable by difference *(a,c)* with factor 2. However, the difference *(aba,cd)* is not separable by difference *(a,c)* since *a* occurs twice in the first constituent while *c* occurs in the second constituent only once.

Separation of a difference (D_1, D_2) with separation difference (d_1, d_2) is the sequence $(\alpha_1, \beta_1)(d_1, d_2)(\alpha_2, \beta_2)(d_1, d_2) \dots (d_1, d_2)(\alpha_n, \beta_n)$, where D_1 consists of the sequence $\alpha_1 d_1 \alpha_2 d_1 \dots d_1 \alpha_n$ and D_2 consists of the sequence $\beta_1 d_2 \beta_2 d_2 \dots d_2 \beta_n$, and empty differences are dropped. separation of a match sequence with a difference is the sequence of similarities and separation of all differences with that difference.

In the framework terminology, the separation differences that separate all the differences in that match sequence and increase the number of differences more than once after the separation of a difference are discriminated as useful. As an instance of this concept, while *(a,b)* is a useful separation difference for match sequence *(ac,bde)g(a,b)* since the total number of differences which occur more than once increases from 0 to 2 after the separation, *(ab,d)* it is not a useful separation difference for this difference since the same parameter does not increase after the separation.

For a match sequence to be separated, the authors describe the most useful separation difference as the one among useful separation differences that separates the match sequence with the greatest factor. If there are more than one useful separation differences separating with the greatest factor *n*, the separation of the match sequence with most useful separation difference should be still separable by the other differences with factor *n*.

There can be many useful separation differences for a match sequence but there is at most one most useful separation difference. For instance, the most useful separation difference for match sequence *(cac,bdb)g(cf,bg)* is *(c,b)* with separation factor 3. For match sequence *(ab,c) g(ab,c)*, there is no most useful separation difference, because neither of *(a,c)* and *(b,c)* has the superiority over the other.

C. Methodology

1) Finding Specific Generalization: : The specific generalization of strings α_1 and α_2 is computed (if exists) by the algorithm in Figure 1. Once unique match sequence of the string pair is found (if there is), the best (not always

most) specific instance of the sequence is computed first. In this algorithm, the specific instance of a match sequence is computed by dividing the match sequence iteratively by the most useful separation difference. The iterations continue until none of the useful separation differences can be favored among others. The inverse substitution step is the operation of replacing differences with variables, with the restriction that same differences correspond to same variables in the result.

As an instance that shows how specific generalization works, consider the generalization of a string pair *abcdfc* and *abghefg*. The common subsequences of these strings are *ab* and *f*. Therefore the unique match sequence of the pair is *ab(cd,ghc)f(c,g)*. For this match sequence, *(c,g)* is the most useful separation difference with separation factor 2. The separation of the sequence with this difference gives the new sequence: *ab(c,g)(d,he)f(c,g)*. Since there is no most useful separation difference for this new sequence, we conclude that *ab(c,g)(d,he)f(c,g)* is the most specific instance for the generalization of the string pair. Applying the inverse substitution process, we get the generalized string *abXYfX* as the result of the specific generalization procedure.

A generalized string is a sequence of characters and variables such as *abX*, which represents all strings starting with *ab*. The generalized set *GS* of a generalized string is all the possible strings that are represented by that string. For instance, *GS(abX) = All strings starting with ab*.

2) Generalizing Predicates: : The proposed method for generalizing predicates is a coverage procedure based on specific generalization of strings. Every generalization rule includes append predicate implicitly in their bodies. For instance, a predicate definition noted as *p(Xa)* corresponds to

```
p(L) :- append(X,[a],L)
```

in Prolog notation.

Two clauses having string arguments are generalized using specific generalization of their arguments if exists. The generalization of two strings α_1, α_2 is their specific generalization, if their specific generalization exists, and it is not a (most general) single variable *X*.

Assume that *S* is a set of ground strings $\alpha_1, \alpha_2, \ldots, \alpha_n$. EG($\alpha$) represents set of ground strings represented by α, where α is a ground or generalized string. To construct the generalized set GEN(*S*) for a set of strings *S*, generalizations of all string pairs are computed and put into GEN(*S*). In the second step, among the generalizations that cover the same examples, the more specific one is kept and the other is removed from the set. Next, the generalizations whose coverage sets are subset of coverage of another generalization are removed from the set. Lastly, if there are generalizations such that all the examples that it covers are also covered by another subset, they are removed from the generalization set. Then *S* is initialized to GEN(*S*) and the whole procedure is repeated until there is no possible generalization that can be computed.

To illustrate how the algorithm works, consider the example clause set $\{p(ba), p(cda), p(a), p(aa), p(faga)\}$. Firstly, GEN(*S*) is initialized to the set of arguments $S = \{ba, cda, a, aa, faga\}$ as in Table I.a. In first iteration, *Xa*, which is the specific generalization of *ba, cda, a*; and *XaYa*, which is the specific generalization of *aa, faga* are added to GEN(*S*) as in Table I.b. Since EG(*ba*), EG(*cda*), EG(*a*), EG(*aa*), EG(*faga*) and EG(*XaYa*) are all subsets of EG(*Xa*), they are removed from the generalization set and generalized clause set will consist of a single clause in the end, which is p(*Xa*) as in Table I.c.

GEN(S)	ba	cda	a	aa	faga
Ex. used	{1}	{2}	{3}	{4}	{5}
EG set	{1}	{2}	{3}	{4}	{5}

a. After Initialization Step

GEN(S)	*Xa*	*XaYa*	*ba*	*cda*	*a*	*aa*	*faga*
Ex. used	{1, 2, 3}	{4, 5}	{1}	{2}	{3}	{4}	{5}
EG set	{1, 2, 3, 4, 5}	{4, 5}	{1}	{2}	{3}	{4}	{5}

b. After First Iteration

GEN(S)	Xa
Exs	{1, 2, 3}
EG	{1, 2, 3, 4, 5}

c. Final Result

TABLE I
GENERALIZATION OF STRINGS $ba, cda, a, aa, faga$

The predicates with multiple string arguments can be generalized in the same way with a little modification. The argument sequence can be treated as a single string separated with a special symbol such as ':', which must not occur as any part of the input. For instance, two example clauses such as, *p(a,bac)* and *p(d,fde)* can be treated as *p(a:bac)* and *p(d:fde)* and the resulting generalization is *p(X:YXZ)*, which corresponds to *p(X,YXZ)*. Therefore the methodology also finds the interdependencies between arguments of a single predicate.

III. INTEGRATION OF BACKGROUND KNOWLEDGE

As pointed out in the initial section, the aim of the research documented in this paper is to develop an inductive learning system for domains with positive-only data, using the idea of string generalization proposed in [1]. For this purpose, initially, we define the concept language that our system will work with. The second point that has been worked is to extend the technique to consider arbitrary first-order background predicates.

A. Language

Studies on attribute-value learning paradigms suffer from the lack of a standard language and notation. Inductive learning systems take their power from the declarativeness of the language they use, and Prolog is accepted almost the standard for these systems. The methodology described in this section also takes the input in Prolog notation, but the language is restricted form of Prolog. The example set and background knowledge consist of function-free ground literals without bodies, which correspond to real-life facts. All the examples in the given set must have the same predicate as we aim to build an empirical single predicate learner, but background

knowledge may include several types of predicates. In this context, a sample example set may be:

$daughter(sibel, ahmet).$ $daughter(sibel, zehra).$

Background knowledge may be:

$parent(zehra, sibel).$ $parent(ahmet, sibel).$

The output hypotheses consist of function-free Horn clauses such as:

$daughter(sibel, X) : -parent(X, sibel).$

B. Generalization

We generalize input examples by considering all symbolic arguments as a singleton list, and argument boundaries are specified by the special symbol ':'. Therefore usage of this symbol as a separate token is not allowed in the input and background knowledge set. This rule does not restrict the language, since a midlevel input can be generated by another token that does not occur in the input and post-process the output to reverse the replacement. So, the impact of preprocess is as follows:

$p([a],[b],[a,c])$ is converted into $p([a,:,b,:,a,c])$.

$p([d],[b],[d,e])$ is converted into $p([d,:,b,:,d,e])$.

From this point on, we treat each token in our system as a single symbol. That is, tokens correspond to characters, and lists correspond to strings in string generalization framework proposed in [1].

Having the argument of each example converted into a single list, we investigate the existence of unique match sequence for each pair of these lists. For a pair, if there is not a unique match sequence, we say that there is not any possible generalization for this pair. Otherwise, we compute the unique match sequence and search for its most useful separation difference as defined in Section II-B. If there is not such a difference, the specific instance is the match sequence itself. If there is a most useful separation difference, the most specific instance is computed using the specific instance algorithm in Figure 1 in Section II-C.

The generalization algorithm can be summarized as Figure 2. This algorithm generalizes two examples with respect to background knowledge. The algorithm returns a set of generalized clauses, and each generalized clause in the set at least covers the example clauses used in the generalization process. The selection criteria to create a hypotheses set which cover all given examples is described in Section 3.3. The algorithm in Figure 2 first finds the specific instance of the unique match sequence of the arguments of the examples. Then, it creates a generalized clause template from this specific instance. The generalized clause templates are specialized by appending body literals with the help of background predicates. A background predicate is added as a body literal if a difference in a generalized clause template is covered by an argument position of that background predicate. A difference (d_l, d_r) is covered by a background predicate if d_l and d_r appear in a same argument position of that background predicate. Specialization process continues as long as there is a difference which is not covered by a body literal.

$Generalize(p(arg_1), p(arg_2))$
- **if (** $uniqueMatchSequence(arg_1, arg_2)$ does not exist **) then**
  - There is no possible generalization, and return the empty set as result.
- **else** {
  - $ums \leftarrow uniqueMatchSequence(arg_1, arg_2)$,
  - $SIofUMS \leftarrow specificInstance(ums)$
  - $genaralizedCls \leftarrow \{\ p(SIofUMS)\ \}$
  - **while (** there is $generalizedCl$ in $generalizedCls$ such that a difference of $generalizedCl$ can be covered by a background predicate **and** that difference has not been already covered by the background predicate in $generalizedCl$ **)** {
    - Drop $generalizedCl$ satisfying the loop condition from $generalizedCls$
    - Let assume that the difference (l, r) has not been previously covered in $generalizedCl$ by a background predicate q, and (l, r) can be covered by the i^{th} argument of q with two coverage results $q(l_1, ..., l_{i-1}, l, l_{i+1}, ..., l_n)$ and $q(r_1, ..., r_{i-1}, r, r_{i+1}, ..., r_n)$
    - Copy $generalizedCl$ into $specializedCl$
    - Add $q((l_1, r_1), ..., (l, r), ..., (l_n, r_n))$ into the body of $specializedCl$
    - Add $specializedCl$ into $generalizedCls$
  - }
  - Replace all differences in the generalized clauses in $generalizedCls$ with appropriate variables
  - Return $generalizedCls$
- }

Fig. 2. Generalization algorithm of InGen

The unique match sequence of the arguments of initial two examples given above is $([a],[d])\,[:,b,:]\,([a,c],[d,e])$. The most useful separation difference for this unique match sequence is $([a],[d])$ with separation factor 2. Therefore we separate the UMS with this separation difference and get $([a],[d])\ [:,b,:]\ ([a],[d])\ ([c],[e])$ as the result. Since there is not a useful separation difference for this match sequence, we conclude that it is the most specific instance and it can be used in the generalization for this example pair. Thus, the generalized clause template $p(([a],[d])\,[:,b,:]\,([a],[d])([c],[e]))$ is put into the set of generalized clauses.

Background predicates are handled at this point, after computing the specific instance. For each difference in the most specific instance, we search a background predicate that contains both left and right constituents in one of its arguments, by using a breadth-first algorithm. If there is such a predicate, we stop the search and reconstruct the initial example pair and extend both of them using the clause that provides coverage.

For instance, for the example clauses given above, assume the background clauses are also provided as follows:

$q([a],[f]).\ q([d],[g]).\ q([b],[a]).$ $r([f]).\ r([g]).\ r([h]).$

Examples	Background Clauses	
daughter(senay, mehmet). $\oplus$	parent(mehmet, senay).	female(senay).
daughter(senay, fatma). $\oplus$	parent(fatma, senay).	female(nese).
daughter(nese, halil). $\oplus$	parent(halil, nese).	female(fatma).
daughter(nese, aylin). $\oplus$	parent(aylin, nese).	female(aylin).

TABLE II
THE DAUGHTER EXAMPLE

Clause	Free variables	Background literals	Covered examples
c_1	0	1	2
c_2	0	1	2
c_3	0	2	4
c_4	0	3	2

TABLE III
INPUT CLAUSES FOR LEARNING DAUGHTER RELATION

In the background knowledge, we see that predicate *q* covers both constituents of the difference $([a],[d])$ in the generalized clause template $p(([a],[d])\ [:,b,:]\ ([a],[d])\ ([c],[e]))$, therefore we specialize this template as follows:

$p(([a],[d])\ [:,b,:]\ ([a],[d])([c],[e]))\ :-$
$q(([a],[d])\ [:]\ ([f],[g]))$.

When we repeat the same procedure for the new generalized clause, we discover that the difference $([f],[g])$ is covered by predicate *r*, and the new specialized clause is as follows:

$p(([a],[d])\ [:,b,:]\ ([a],[d])([c],[e]))\ :-$
$q(([a],[d])\ [:]\ ([f],[g]))\ \&\ q(([f],[g]))$.

Next, we see that there is no other difference that can be coverable by a background predicate, therefore we stop specialization process.

The next issue is the generalization of found generalized clause templates. In this step, similarities are kept as constants, as performed in the specific generalization algorithm listed in Figure 1 in Section II-C. Note that, if there are several arguments, all ':' symbols must occur as similarities, since they occur the same number of times in each input list. The last process in this step is to replace the differences with variables, respecting the rule that the same differences are replaced with the same variables. Therefore the generalization that will be extracted for the example pair given above is:

$p(X\ [:,b,:]\ X\ Y)\ :-\ q(X\ [:]\ Z)\ \&\ q(Z)$.

that corresponds to the following Prolog clause:

$p(X,b,L) :-\ append(X,Y,L),\ q(X,Z),\ r(Z)$.

C. Construction of the hypotheses

In our system, the hypotheses set consists of the general clauses computed by generalization algorithm in Figure 2 and the examples themselves. The generalization algorithm is executed for all example pairs, and all generated generalized clauses are put into the hypotheses set. From a set of generalized clauses, we select the best clause with respect to our selection criteria in order to put it into the hypotheses and remove the examples covered by this clause. In the next step, the clauses that do not cover any example are removed from the clause set and the best clause from the remaining clause is chosen. Disjunction of selected clauses build the hypotheses. The iteration procedure continues until all the examples are covered by the hypotheses.

To select the best clause to add into the hypothesis, various criteria are applied for selection of best clause in the following order:

1) The number of free variables, which are the variables that appear only once in the body of a generalized clause. The clauses having fewer free variables are preferred over the others, since we believe that greater number of free variables lead to irrelevant clauses.
2) The number of examples that are covered by the clause. The clauses that cover more examples are favored over the others.
3) The number of body literals. The clauses having greater number of background literals are favored over the others, since we believe that each background literal introduces further specialization and more specific generalizations.

To illustrate how our methodology works, consider the case of learning daughter example, with instances of background knowledge clauses, female and parent, as in Table II. The initial clauses generated by InGen are:

$c_1 = daughter(senay, X_0)\ :-\ parent(X_0, senay)$.
$c_2 = daughter(nese, X_0)\ :-\ parent(X_0, nese)$.
$c_3 = daughter(X_0, X_1)\ :-$
$parent(X_1, X_0), female(X_0)$.
$c_4 = daughter(X_0, X_1)\ :-$
$parent(X_1, X_0), female(X_0), female(X_1)$.

The value of the selection criteria for each of the clauses are in Table III

Among the generated clauses, we observe that each of c_1, c_2, c_3, c_4 does not include free variables. Since clause c_3 covers all of the examples, it is selected into the hypotheses set. Therefore the output hypothesis is:

$H = \{c_3\} =$
$\{daughter(X_0, X_1) :-\ parent(X_1, X_0), female(X_0).\}$

which is the correct description of the concept.

There may be cases where some examples are covered by none of the generalized clauses. In such cases, the examples which are not covered by any of the clauses are lastly added to the hypotheses set to make the hypotheses complete with respect to the given example set.

IV. EXPERIMENTATION

We evaluated the performance of our system with several example sets. In this paper, we showed the performance of ours system in two family relations, namely daughter and aunt, and compared the generated results with two concurrent ILP learners, Progol and FOIL. The results show that InGen is somewhat competitive with these state-of-art systems.

For the learning daughter relation problem given in previous section, in order to start learning, Progol system needs mode and type declarations. That is, in addition to presentation of inputs in Prolog notation, Progol requires:

```
modeh(1,daughter(+person,+person))?
modeb(*,parent(-person,+person))?
modeb(*,parent(+person,-person))?
```

Example set (All positive)	Background Clauses	
aunt(jane,henry).	father(sam,henry).	sister(jane,sam).
aunt(sally,jim).	parent(sam,henry).	sister(sally,sarah).
aunt(judy,jim).	mother(sarah,jim).	sister(judy,sarah).
	parent(sarah,jim).	

TABLE IV
INPUT CLAUSES FOR LEARNING AUNT RELATION

```
modeb(*,female(+person))?
```

as mode declaration and

```
person(nese).
person(ali).
person(senay). ...
```

as type declaration. However, even with this additional information, the most specific clause produced by Progol for this input set is:

```
daughter(A,B) :-
  parent(B,A), parent(C,A), female(A).
```

and the output hypotheses produced is a single clause:

```
daughter(A,B) :- parent(B,A).
```

discarding the condition that A must be female, and is inconsistent. For the same input examples presented in a similar way, FOIL gives the same result as output.

For the same example, InGen does not need any type or mode declaration. As mentioned in the previous chapter, the output hypotheses generated by InGen for the daughter example set is:

```
daughter(X0, X1) :-
  parent(X1, X0), female(X0).
```

which is the correct description of the concept and shows that InGen is able to find the correct concept description using only examples and background clauses.

The aunt relation is generated and used by Muggleton to test his system Progol. The example set consists of three positive examples of the concept a person being aunt of another one, and a background knowledge set that includes father, mother, sister and parent relations that are related with the example set. This example is more complex and has features that can mislead an ILP system because a new variable (standing for the parent) must be introduced to correctly describe the concept.

The example and background knowledge set are as in Table IV. For this input, Progol can learn the relation:

```
aunt(A,B) :- parent(C,B), sister(A,C).
```

successfully. Furthermore Progol is capable of learning the same fact using the non-ground background clauses:

```
parent(Parent,Child) :-
  father(Parent,Child).
parent(Parent,Child) :-
  mother(Parent,Child).
```

instead of:

```
parent(henry,sam).    parent(jim,sarah).
```

which InGen is incapable of. FOIL was unable to answer for this input. InGen produces the following clauses:

```
aunt(X0, X1) :-
  sister(X0, X2), parent(X2, X1).
aunt(X0, jim) :- sister(X0, sarah).
```

The first clause takes its root from the generalization of the first and second examples and the second clause takes its root from the generalization of the last two. The output is:

```
aunt(X0, X1) :-
  sister(X0, X2), parent(X2, X1).
```

which is complete and correctly describes the concept. The first clause is preferred over the second since it covers all three examples while the second clause covers the last two.

V. CONCLUSION

The research outlined in this paper is an initial attempt to build an ILP system based on the specific generalization of strings proposed in [1]. The main contribution was that, usage of background knowledge for string generalization is integrated. By considering the background predicates and differences that covered by background predicates, the generalized clauses are further specialized by adding body literals.

The experiments we performed demonstrate that the system is an effective learner. It could successfully learn the family relations such as daughter, aunt and granddaughter, a card game, and possible grammatical structure of English sentences.

Although our system is capable of learning these concepts successfully, it is not a perfect ILP system. There are several potential directions to study to make our system better:

- The system is capable of learning function-free ground clauses only. The concept description language may be enlarged to cover functional and non-ground clauses.
- Our system is not capable of inventing new predicates, which may be necessary for learning some concepts such as sorting.
- The system should be tested with dataset having several thousands of examples to test its effectiveness.
- Different clause selection criteria can be applied.

As a conclusion, this work is an initial attempt for building an ILP system using specific generalization of strings and it can serve as a basis for future research to construct an effective learner using the same notion.

REFERENCES

[1] Cicekli, I. and Cicekli, N.K. Generalizing predicates with string arguments. Applied Intelligence. (2005).

[2] Plotkin, G.D. Automatic methods of inductive inference. PhD. Thesis, Edinburgh University. (1971).

[3] Dzeroski, S. Cussens, J., Manandhar S. An Introduction to Inductive Logic Programming and Learning Language in Logic. Learning Language in Logic. (2000).

[4] Mooney, R.J. and Califf M.E. Induction of First-Order Decision Lists: Results on Learning the Past Tense of English Verbs. Journal Of Artificial Intelligence Research. (1995).

[5] Muggleton, S. Learning from Positive Data. Machine Learning. (2001).

[6] Muggleton, S., Feng, C. Efficient Induction of Logic Programs. Proceedings of the First Conference on Algorithmic Learning Theory. (1990)

[7] Muggleton, S. Inverse Entailment and Progol. New Generation Computing. (1995).

[8] Quinlan, J.R. Learning relations: comparison of a symbolic and a connectionist approach. University of Sydney. (1989).

[9] Lavrac, N., Dzeroski, S. Inductive Logic Programming. Ellis Horwood. (1994)

[10] Mitchell, T. M. Machine Learning. McGraw-Hill Science/Engineering/Math. (1997)

[11] Lloyd, J.W. Foundations of Logic Programmming. Springer-Verlag. (1984)