Appl Intell (2006) 25:23–36
DOI 10.1007/s10489-006-8864-1

Generalizing predicates with string arguments

Ilyas Cicekli · Nihan Kesim Cicekli

Abstract The least general generalization (LGG) of strings may cause an over-generalization in the generalization process of the clauses of predicates with string arguments. We propose a specific generalization (SG) for strings to reduce over-generalization. SGs of strings are used in the generalization of a set of strings representing the arguments of a set of positive examples of a predicate with string arguments. In order to create a SG of two strings, first, a unique match sequence between these strings is found. A unique match sequence of two strings consists of similarities and differences to represent similar parts and differing parts between those strings. The differences in the unique match sequence are replaced to create a SG of those strings. In the generalization process, a coverage algorithm based on SGs of strings or learning heuristics based on match sequences are used.

Keywords Inductive logic programming · Machine learning · String generalization

1. Introduction

One of the main issues in Inductive Logic Programming (ILP) is the induction of predicate definitions from only positive examples. However learning from only positive examples may cause over-generalization because there are no restrictions imposed by negative examples. Many researchers have worked on the ILP systems which can learn from positive examples [3, 5, 7, 9, 14]. Some of them use statistical techniques to overcome this over-generalization problem. Muggleton [7] showed that logic programs are learnable with low expected error from positive examples within a Bayesian framework.

Predicates with string arguments naturally occur in many problem domains. The ILP techniques presented in this paper can be used in the induction of predicates with string arguments from only positive examples. For example, translation rules between two natural languages can be a predicate with two string arguments, and some of ILP techniques discussed here are successfully used in the learning process of the translation rules from given translation examples [1, 2, 4].

The proposed learning process for the predicates with string arguments assumes that only positive examples are available, and the predicate *append* which is assumed to be in background knowledge can appear in the body of the induced predicates. We also present a learning heuristic based on match sequences, and this learning heuristic is used in the induction of a recursive predicate in a special form.

In the proposed framework, the generalization of two strings depends on the unique match sequence between those two strings. The unique match sequence represents similarities and differences between a pair of strings. A similarity is a common substring of two strings, and a difference represents differing parts between two strings.

From a given set of positive examples for a predicate with string arguments, some of the current ILP systems learn over-generalized rules or do not perform any generalization at all. Let us assume that, we want to learn predicate `p` from the following positive examples given as Prolog clauses. Here, we assume that lists represent string arguments.

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

I. Cicekli (✉)
Department of Computer Engineering, Bilkent University, Ankara, Turkey
e-mail: ilyas@cs.bilkent.edu.tr

N. K. Cicekli
Department of Computer Engineering, METU, Ankara, Turkey
e-mail: nihan@ceng.metu.edu.tr

Although these two clauses have a common property, this property will not be captured by most of the current ILP systems. This common property is that the first arguments end with atom `b`, and the second arguments end with atom `y`. For example, the GOLEM system [7], which is one of ILP systems and uses only Plotkin's RLGG schema [12], generalizes these clauses with the following clause

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

without capturing that common property. This clause is an over generalization, and it accepts any lists whose lengths are more than one for both arguments. For the same positive examples, the Progol system [9] does not perform any generalization, and it returns the given positive examples as result. On the other hand, our proposed mechanism generalizes these clauses with the following clause.

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

This generalized clause means that any list ending with atom `b` can be the first argument of the induced predicate `p`, and a list ending with atom `y` can be the second argument. Here, we assume that the predicate `append` is known as background knowledge, and it can be used in the body of the induced predicate.

The rest of the paper is organized as follows. In Section 2, we explain how a unique match sequence, which represents similarities and differences in a pair of strings, can be found. In Section 3, we propose a specific generalization (SG) for two strings, which is created from the unique match sequence of those strings. A generalization of predicates is explained in Section 4. In Section 5, we describe a learning heuristic based on SGs of strings to generalize positive examples of recursive predicates with string arguments in a certain form, and the usage of this learning heuristic in the induction of translation templates in an example-based machine translation system. Section 6 explains how extra background knowledge can be used in the learning process by giving an application in grammar learning domain. Finally, we conclude the paper with pointers for further research.

2. Unique match sequence

In this section, we give a formal definition of a match sequence and a unique match sequence between two strings. Let us assume that every string is a sequence of symbols in a finite alphabet. Before the definition of the match sequence, we need to define similarity and difference first.

Definition 1. - Similarity. A *similarity* between α_1 and α_2, where α_1 and α_2 are two strings, is a string β such that it satisfies the following conditions:

(i) $\alpha_1 = \alpha_{1,1}\beta\alpha_{1,2}$ and $\alpha_2 = \alpha_{2,1}\beta\alpha_{2,2}$.
(ii) If both of $\alpha_{1,1}$ and $\alpha_{2,1}$ are not empty, their last symbols cannot be the same symbol.
(iii) If both of $\alpha_{1,2}$ and $\alpha_{2,2}$ are not empty, their first symbols cannot be the same symbol.
(iv) The similarity β cannot be an empty string unless both α_1 and α_2 are empty.

Definition 2. - Difference. A *difference* between two strings α_1 and α_2 is a pair of two strings (β_1, β_2), and it satisfies the following conditions:

(i) $\alpha_1 = \alpha_{1,1}\beta_1\alpha_{1,2}$ and $\alpha_2 = \alpha_{2,1}\beta_2\alpha_{2,2}$.
(ii) Either both of $\alpha_{1,1}$ and $\alpha_{2,1}$ must be empty, or both of them must be non-empty. In the latter case, their last symbols must be same.
(iii) Either both of $\alpha_{1,2}$ and $\alpha_{2,2}$ must be empty, or both of them must be non-empty. In the latter case, their first symbols must be same.
(iv) The same symbol cannot occur in both β_1 and β_2, and at least one of them is not empty.

According to these definitions, a similarity represents a similar part between two strings, and a difference represents a pair of differing parts between two strings. For example, *cd* represents a similarity between the strings *abcd* and *fcd*, and *(ab, f)* represents a difference between them.

Definition 3. - Match sequence. A *match sequence* between two strings α_1 and α_2 is a sequence $P_1 \ldots P_n$, where each P_i is a similarity S_i or a difference $D_i = (D_{i,1}, D_{i,2})$ and $n \geq 1$, and this sequence satisfies the following conditions:

(i) If we define two constituent functions as follows:

$$C_{i,1} = \begin{cases} S_i & \text{if } P_i \text{ is a similarity } S_i \\ D_{i,1} & \text{if } P_i \text{ is a difference } (D_{i,1}, D_{i,2}) \end{cases}$$

$$C_{i,2} = \begin{cases} S_i & \text{if } P_i \text{ is a similarity } S_i \\ D_{i,2} & \text{if } P_i \text{ is a difference } (D_{i,1}, D_{i,2}) \end{cases}$$

then $\alpha_1 = C_{1,1} \ldots C_{n,1}$ and $\alpha_2 = C_{1,2} \ldots C_{n,2}$.
(ii) A similarity cannot follow another similarity, and a difference cannot follow another difference in a match sequence.

The conditions for the match sequence guarantee that there will be at least one match sequence for any given two strings. But they do not guarantee that there will be at most one

Table 1 Match sequence examples

α	β	$MSs(\alpha, \beta)$
ϵ	ϵ	$\{\epsilon\}$ The match sequence of two empty strings is a sequence of a single similarity which is an empty string
a	a	$\{a\}$ The match sequence of two identical strings is a sequence of a single similarity which is equal to that string
a	b	$\{(a, b)\}$ The match sequence of two completely different strings is a sequence of a single difference
abc	$dbef$	$\{(a, d)b(c, ef)\}$
ab	abc	$\{ab(\epsilon, c)\}$
abc	$dbebf$	$\{(a, d)b(c, ebf),\ (a, dbe)b(c, f)\}\}$
abc	bdb	$\{(a, \epsilon)b(c, db),\ (a, bd)b(c, \epsilon)\}$
ab	ba	$\{(a, \epsilon)b(\epsilon, a),\ (\epsilon, b)a(b, \epsilon)\}$
$abcbd$	$ebfbg$	$\{(a, e)b(c, f)b(d, g),\ (a, ebf)b(cbd, g),\ (abc, e)b(d, fbg)\}$

match sequence for any given two strings. This means that there can be more than one match sequence for any given two strings. For example, the strings abc and $dbef$ in Table 1 have only one match sequence $(a, d)b(c, ef)$ because both of those strings contain only one common substring. On the other hand, the strings abc and $dbebf$ have two match sequences, because common substring b occurs once in the first string, and it occurs twice in the second string. The number of match sequences between two strings depends on both the number of the common parts and the positions of the common parts in those strings. For illustration purposes, Table 1 gives the match sequences for some string pairs. The first two columns show the pair of the strings that are compared, and the third column is the set of all possible match sequences between them.

Definition 4. - Unique match sequence. A *unique match sequence* (UMS) between two strings α_1 and α_2 is a match sequence between α_1 and α_2 such that the following conditions must be satisfied:

(i) If a symbol occurs in a similarity, it cannot occur in any difference.
(ii) If a symbol occurs in the first constituent of a difference, it cannot occur in the second constituent of any difference.

Any given two strings will have either only one *unique match sequence* or they will not have a unique match sequence at all. The conditions (i) and (ii) above guarantee its uniqueness when a unique match sequence exists for a pair of strings. Although the same symbol can appear in more than one similarity according to the conditions above, the following facts about unique match sequences can be observed:

– If a symbol appears in both α_1 and α_2, it must appear n times, where $n \geq 1$, in both of those strings. Otherwise, they cannot have a unique match sequence. For example, the strings *bc* and *bd* have a unique match sequence *b(c,d)* because the symbol *b* occurs exactly once in both of the strings. On the other hand, the strings *bc* and *bdb* cannot have a unique match sequence because the symbol *b* occurs only once in the first string and it occurs twice in the second one. This means that the symbol *b* must end up in a similarity and a difference of a match sequence of those strings, but this violates the condition (i) of the unique match sequence.
– If a symbol appears n times in both α_1 and α_2 where $n \geq 1$, its ith occurrence in α_1 and its ith occurrence in α_2 must end up in the same similarity of their unique match sequence. For example, the strings *bbcb* and *bbdb* have a unique match sequence *bb (c,d)b* where the first and second occurrences of the symbol *b* in those strings end up in the first similarity, and its third occurrences end up in the second similarity.
– If two symbols a and b appear in both α_1 and α_2, and the ith occurrence of a appears before the jth occurrence of b in α_1, the ith occurrence of a must appear before the jth occurrence of b in α_2 too. Otherwise, those strings cannot have a unique match sequence. For example, the strings *bdc* and *bec* have a unique match sequence *b (d,e)c* because the symbol *b* occurs before the symbol *c* in both of those strings. On the other hand, the strings *bdc* and *ceb* cannot have a unique match sequence because the symbol *b* occurs before the symbol *c* in the first string, and it occurs after the symbol *c* in the second string.

Some more examples for unique match sequences:

1. The unique match sequence of two empty strings is a sequence of a single similarity which is an empty string.
2. The unique match sequence of two identical strings is a sequence of a single similarity which is equal to that string. For example, the unique match sequence of ab and ab is ab.

3. The unique match sequence of two totally different strings is a sequence of a single difference. For example, the unique match sequence of ab and c is (ab, c).
4. The unique match sequence of $abcb$ and $dbebf$ is $(a, d)\, b\, (c, e)\, b\, (\epsilon, f)$.
5. There is no unique match sequence for abc and bdb because b appears once in abc but it occurs twice in bdb.
6. There is no unique match sequence for ab and ba because a appears before b in ab but a appears after b in ba.
7. The unique match sequence of $abcadb$ and $eabfagbh$ is $(\epsilon, e)\, ab\, (c, f)\, a\, (d, g)\, b\, (\epsilon, h)$.

3. Specific generalization of strings

The specific generalization of two strings is a *generalized string* that is a string of symbols and variables. The variables in generalized strings represent possible ground strings, and same variables represent the same ground strings.

The definition of the match sequence (and the unique match sequence) will be extended for generalized strings by assuming each variable as a new symbol. Before the unique match sequence of two generalized strings is found, all variables in one of the strings are renamed so that the strings do not contain the same variables. Then each variable is treated as a new symbol in the creation of the unique match sequence. Because of this renaming operation, a variable cannot appear in the similarity of a unique match sequence of two generalized strings. For example, the unique match sequence of $aXbcYd$ and $efbcZ$ will be $(aX, ef)\, bc\, (Yd, Z)$.

Each string (ground or generalized string) represents a set of ground strings (strings without variables). For example, the generalized string Xa represents the set of all ground strings ending with the symbol a. We say that the ground string function $GS(\alpha)$ represents the set of all ground strings that are covered by the string α. If α is a ground string, $GS(\alpha)$ is $\{\alpha\}$. The GS function creates following relations among strings:

- The string β is *more general* than the string α if $GS(\alpha) \subset GS(\beta)$.
- The string α is *more specific* than the string β if $GS(\alpha) \subset GS(\beta)$.
- The string α is *equal* to the string β if $GS(\alpha) = GS(\beta)$

For example, $GS(bXa)$ is the set of all ground strings starting with b and ending with a, and $GS(Xa)$ is the set of all ground strings ending with a. Since $GS(bXa) \subset GS(Xa)$, the string bXa is more specific than the string Xa, and Xa is more general than bXa. The strings XY and Z are equal because $GS(XY) = GS(Z)$ where both $GS(XY)$ and $GS(Z)$ represent the set of all possible ground strings. On the other hand, there is no specificity relation between strings Xa and bY because $GS(Xa) \not\subset GS(bY)$, or $GS(bY) \not\subset GS(Xa)$, or $GS(Xa) \neq GS(bY)$.

A generalized string is obtained from an instance of a unique match sequence in which all differences are replaced with variables, and the same differences are replaced with the same variables. An *instance* of a unique match sequence is obtained by dividing differences in that unique match sequences by sequences of differences. For example, the instance *(b, d) (c, e) a (c, e)* is obtained from the unique match sequence *(bc, de) a (c,e)* by dividing the difference *(bc, de)*. Although a difference cannot follow another difference in a match sequence, a difference can follow another difference in an instance of that match sequence. When an instance of a unique match sequence is generalized instead of that unique match sequence we may get a more specific generalized string. When the unique match sequence *(bc, de) a (c, e)* is generalized, we get XaY as a generalized string. On the other hand, when its instance *(b,d) (c,e) a (c,e)* is generalized, we get $XYaY$ as a generalized string. The string $XYaY$ is more specific than the string XaY.

In order to find the specific generalization of two strings, first the most specific instance of their unique match sequence is found. The *most specific instance* of a unique match sequence is one of its instances such that the most specific string is obtained when that instance is generalized. A match sequence may not have a unique most specific instance. In that case, we will be conservative and we will find a specific instance but it may not be the most specific one. As a result, we may not find the most specific generalization but we will find a specific one in that case. For example, *(b, d) (c, e) a (c, e)* is the most specific instance of the match sequence *(bc, de) a (c, e)*. On the other hand, the match sequence *(cd, fe) a (c, e) a (d, f)* has two specific instances $(c, \epsilon)\, (d, f)\, (\epsilon, e)\, a\, (c, e)\, a\, (d, f)$ and $(\epsilon, f)\, (c, e)\, (d, \epsilon)\, a\, (c, e)\, a\, (d, f)$, and none of these instances is more specific than the other one. To avoid this kind of ambiguity, we select an instance that is more general than both of the specific instances. In this case, the match sequence *(cd, fe) a (c, e) a (d, f)* will be the specific instance of itself.

In order to find a specific instance of a unique match sequence, the differences in that match sequence are replaced by sequences of differences. The replacement of a difference should not lead to an ambiguity, and that replacement should be the most useful one. In Section 3.1, we discuss how to handle this ambiguity problem and how to select the best difference replacement. We describe the algorithm that finds a specific instance of a unique match sequence in Section 3.2.

3.1. Separable differences

In order to avoid the ambiguity, a difference is broken up into a sequence of differences by another difference, and this break-up operation should satisfy the conditions given in the following definition.

Definition 5. - Separable difference. A difference (A, B) is *separable* by a difference (α, β) iff the following conditions are satisfied:

(i) α occurs n times in A where $n \geq 0$, and any symbol of α does not occur in other parts of A. In other words, $A = a_1\alpha a_2 \ldots \alpha a_{n+1}$, and each a_i does not contain any symbol of α.
(ii) β occurs n times in B where $n \geq 0$, and any symbol of β does not occur in other parts of B. In other words, $B = b_1\beta b_2 \ldots \beta b_{n+1}$, and each b_i does not contain any symbol of β.
(iii) If α is empty, each b_i cannot be empty unless a_i is empty.
(iv) If β is empty, each a_i cannot be empty unless b_i is empty.

The difference (A, B) is separated into a sequence of differences in the form $(a_1, b_1)(\alpha, \beta)(a_2, b_2) \ldots (\alpha, \beta)(a_{n+1}, b_{n+1})$ where we drop (a_i, b_i) from the sequence if both a_i and b_i are empty. We say that the difference (A, B) is separable by the difference (α, β) with factor n.

The purpose of the conditions (iii) and (iv) above is to eliminate a possible ambiguity. If we do not impose the restriction (iii), we could get a difference sequence $(\epsilon, \beta)(a_i, \epsilon)$ as a part of a match sequence instance. But, since this difference sequence can also be rewritten as $(a_i, \epsilon)(\epsilon, \beta)$, this will cause an ambiguity. A similar discussion also applies to the condition (iv).

For example, the difference (cac, dbd) is separable by the difference (c, d) into the difference sequence $(c, d)\,(a, b)\,(c, d)$. In this case, the separation factor is 2. On the other hand, the difference (cac, db) is not separable by the difference (c, d) because c appears twice in cac but d appears only once in db. The difference (ab, fg) is separable by the difference (c, d) into itself with factor 0 because c does not occur in ab, and d does not occur in fg. The difference (a, b) cannot be separable by the difference (a, ϵ) because the condition (iv) will be violated. If we try to separate (a, b) with (a, ϵ), we will cause an ambiguity by getting two difference sequences $(a, \epsilon)(\epsilon, b)$ and $(\epsilon, b)(a, \epsilon)$.

A match sequence (or an instance of a match sequence) may contain more than one difference. To be able to separate a difference in the match sequence by the difference D, all of the differences in that match sequence must be separable by that difference D. If the differences in a match sequence are $(A_1, B_1), \ldots, (A_m, B_m)$, they are separable by a non-empty difference (α, β) iff each (A_i, B_i) is separable by (α, β) with the factor n_i where $n_i \geq 0$. In this case, we say that $(A_1, B_1), \ldots, (A_m, B_m)$ is separable by (α, β) with the factor n where $n = \sum_{i=0}^{m} n_i$. If all the differences in a match sequence are separable by a difference, we create an instance of that match sequence by separating all the differences by that difference. For example, the differences in the match sequence $(a, b)\,g\,(ad, bf)$ are separable by the difference (a, b) with the factor 2 into the differences in the instance $(a, b)\,g\,(a, b)\,(d, f)$.

A match sequence can be separable by more than one difference. We want to find the separation difference which lead to the most specific instance of that match sequence. Among all separation differences for a match sequence, the one that leads to the most specific instance without ambiguity is selected, and it is called the *most useful separation difference*.

We say that a difference D is a *useful separation difference* for a match sequence (or an instance of match sequence) if all the differences in that match sequence are separable by D, and the total number of differences which occur more than once is increased after the separation. For example, the difference (a, b) is a *useful separation difference* for the match sequence $(ac, bde)g(a, b)$ because the instance $(a, b)\,(c, de)\,g\,(a, b)$ is the result of the separation of this match sequence by the difference (a, b), and the total number of differences which occur more than once is increased from 0 to 2 as a result of this separation. But, the difference (a, bd) is not a useful separation difference for this match sequence, because the separation by that difference does not lead to any increase in the total number of differences which occur more than once.

Definition 6. - Most useful separation difference. We say that a useful separation difference D for a match sequence is the *most useful separation difference* for that match sequence iff the following conditions hold:

(i) The differences of this match sequence are separable by the useful separation difference D with the factor n.
(ii) There is no other useful separation difference D_2 that can separate the differences of this match sequence with the factor m such that $m > n$.
(iii) If there is another useful separation difference D_2 that can separate the differences of this match sequence with the factor n, the differences in the resulting instance after the separation of the differences in the match sequence by D must still be separable by D_2 with factor n.

It is possible that there can be many useful separation differences for a match sequence, but there might not be the most useful separation difference for that match sequence. The last condition above is used to avoid ambiguous separations of the

- $SIofUMS(\alpha_1, \alpha_2) \leftarrow UMS(\alpha_1, \alpha_2)$
- **while** *(there is a most useful separation difference D for* $SIofUMS(\alpha_1, \alpha_2)$ *with factor n where* $n \geq 2$ *)* **do**
  - *Separate all the differences in* $SIofUMS(\alpha_1, \alpha_2)$ *by the most useful separation difference D to get a new instance, and put the result into* $SIofUMS(\alpha_1, \alpha_2)$.

Fig. 1 Specific instance algorithm

differences. That condition also prefers the longest one when there are two useful differences with the same separation factor. For example, the most useful separation difference for the match sequence $(cac, bdb)g(cf, bg)$ is (c, b) with factor 3. The most useful separation difference for $(abab, cdc)$ is (ab, c) with factor 2 because two other useful separation differences (a, c) and (b, c) with factor 2 do not satisfy the last condition above. On the other hand, there is no most useful separation difference for $(ab, c)g(ab, c)$ because neither (a, c) nor (b, c) with factor 2 satisfy the last condition.

3.2. Finding specific instance and specific generalization

A specific instance *SI of UMS*(α_1, α_2) of a unique match sequence $UMS(\alpha_1, \alpha_2)$ is found by the algorithm in Fig. 1. If a unique most specific instance for a match sequence is available, the algorithm in Fig. 1 will find it. If there is no unique most specific instance, we do not favor one instance over another instance because we do not use any statistical technique in this process. The algorithm stops when it detects an ambiguity among useful separation differences with a maximum separation factor. In those cases, we accept a less specific generalization to avoid ambiguity.

At each iteration of the specific instance algorithm in Fig. 1, a most useful separation difference with a maximum separation factor is used to separate the differences in the current specific instance. For example, the match sequence $(abc, de)\, g\, (a, d)\, g\, (afc, de)$ has two useful separation differences. The first one is (a, d) with separation factor 3, and the second one is (c, e) with separation factor 2. Since the first one is the most useful one, the match sequence is separated by the first separation difference, and we get the instance $(a, d)\, (bc, e)\, g\, (a, d)\, g\, (a, d)\, (fc, e)$. At the next iteration, this instance is separated with the difference (c, e), and the instance $(a, d)\, (b, \epsilon)\, (c, e)\, g\, (a, d)\, g\, (a, d)\, (f, \epsilon)\, (c, e)$ is found. Since there is no more useful separation difference for this instance, it will be the specific instance of the match sequence $(abc, de)\, g\, (a, d)\, g\, (afc, de)$.

After a specific instance of the unique match sequence for α_1 and α_2 is found, all differences are replaced with variables in order to create the specific generalization $SG(\alpha_1, \alpha_2)$ for those strings. The algorithm in Fig. 2 finds a specific generalization of two strings α_1 and α_2.

- *Find the unique match sequence* $UMS(\alpha_1, \alpha_2)$ *of* α_1 *and* α_2. *If* $UMS(\alpha_1, \alpha_2)$ *does not exist, these strings are not generalized, and we say that* $SG(\alpha_1, \alpha_2)$ *does not exist for these strings.*
- *Find a specific instance* $SIofUMS(\alpha_1, \alpha_2)$ *of* $UMS(\alpha_1, \alpha_2)$.
- *Replace all differences in this specific instance* $SIofUMS(\alpha_1, \alpha_2)$ *with new variables, and replace the same differences with the same variables in order to create* $SG(\alpha_1, \alpha_2)$.

Fig. 2 Specific generalization algorithm

Some examples of SGs of strings are:

1. $UMS(abcd, ecfg)$ is $(ab, e)\, c\, (d, fg)$. Since a specific instance of this match sequence is itself, two differences in this unique match sequence are replaced with two new variables to create the SG of those strings. Thus, $SG(abcd, ecfg)$ is XcY.
2. $UMS(abcdeaf, gbchegf)$ is $(a, g)\, bc\, (d, h)\, e\, (a, g)\, f$. Since a specific instance of this match sequence is itself, $SG(abcdeaf, gbchegf)$ is $XbcYeXf$. In this example, the same differences are replaced with the same variable.
3. $UMS(cac, fbadf)$ is $(c, fb)\, a\, (c, df)$. Since this match sequence has two separable differences, we find a specific instance of this match sequence, and this instance is $(c, f)(\epsilon, b)\, a\, (\epsilon, d)(c, f)$, and $SG(cac, fbadf)$ is $XYaZX$.
4. The SG of two strings without any common symbols will be a single variable.

4. Generalization of predicates

In this section, we present a generalization algorithm for the predicates with string arguments. This generalization algorithm induces the predicate from its given positive examples, and it is a coverage algorithm based on the specific generalization of strings. The coverage algorithm induces a set of predicate definitions, which covers all given positive examples. Each predicate definition in the induced set covers some of the given positive examples, and the predicate *append*, which is assumed to be in the background knowledge, can appear in the body of the definition of that predicate. In our notation, although we represent the predicate definitions using generalized strings, the usage of a generalized string means that the predicate *append* is used in the body of the definition of the induced predicate. For example, a learned predicate definition `p(Xa)` in our notation can be represented as `p(L) :- append(X,[a],L)` in Prolog notation. In the rest of this section, the generalization of single-arity predicates is discussed and then the discussion is extended for multiple-arity predicates.

4.1. Generalization of single-arity predicates

Two clauses of a single arity predicate p are generalized by using the SG of their arguments. The SG of two strings exists only if they have a unique match sequence. If they do not have a unique match sequence, they do not have a SG, and we do not generalize those clauses. Let us assume that $p(\alpha_1)$ and $p(\alpha_2)$ are two clauses of the single-arity predicate p where each α_i can be a ground string or a generalized string. $p(\alpha_i)$ is a positive example of p if α_i is a ground string. $p(\alpha_i)$ is a generalization of the other clauses of p if α_i is a generalized string. The strings α_1 and α_2 cannot contain the same variable if both of them are generalized strings.

The generalization $GEN(\alpha_1, \alpha_2)$ of the arguments of two clauses $p(\alpha_1)$ and $p(\alpha_2)$ of a single-arity predicate p is defined as follows:

$$GEN(\alpha_1, \alpha_2) = \begin{cases} SG(\alpha_1, \alpha_2) & \text{If } SG(\alpha_1, \alpha_2) \text{ exists, and} \\ & \text{it is not a single variable} \\ \text{none} & \text{Otherwise} \end{cases} \quad (1)$$

According to the definition of GEN, two clauses are generalized only if their arguments have a SG and that SG is not a single variable. When the SG of their arguments is a single variable, two clauses are not generalized to avoid the over-generalization of the predicate.

Let us assume that $S = \{\alpha_1, \alpha_2, \ldots, \alpha_n\}$ is a set of the arguments of the positive examples of a single-arity predicate p. The generalization algorithm that finds the generalization set *GEN(S)* for a given set of strings S is given in Fig. 3. The generalized strings in the found set *GEN(S)* are the arguments of the generalized clauses of the single-arity predicate p. Initially, *GEN(S)* is assumed to be S, and at each iteration of the algorithm a new generalization set is created from the old generalization set by finding the generalization of all string pairs in the old set. The algorithm uses a coverage function EG. The *coverage function EG* for a string represents the set of the positive examples whose arguments are covered by that string. $EG(\alpha)$ is $\{i\}$ if α is the argument of the positive example E_i. If α is a generalized string, $EG(\alpha)$ is the set of the numbers of the positive examples whose arguments in $GS(\alpha)$. If there are two different generalized strings which cover the same positive examples, the most specific one is kept and the other one is deleted from the set. If all positives examples covered by a string are also covered by some other strings in the generalization set, that string is also deleted from the set. The following two examples in this section demonstrate the details of the generalization algorithm.

- $GEN(S) \leftarrow S$.
- **while** *(there are two strings α_1 and α_2 in $GEN(S)$ such that $GEN(\alpha_1,\alpha_2)$ exists)* **do**
  - $OLDGEN(S) \leftarrow GEN(S)$
  - *For every pair of two distinct strings α_1 and α_2 in $OLDGEN(S)$ such that $GEN(\alpha_1, \alpha_2)$ exists, add $GEN(\alpha_1, \alpha_2)$ into $GEN(S)$. (where $\alpha_1 \neq \alpha_2$).*
  - *Drop each string G_i from $GEN(S)$ if there exists another string G_j in $GEN(S)$ such that $EG(G_i) = EG(G_j)$, and G_i is a most general form of G_j (where $G_i \neq G_j$). So, if two strings cover the same positive examples, we keep only the most specific one.*
  - *Drop each string G_i from $GEN(S)$ if there exists another string G_j in $GEN(S)$ such that $EG(G_i) \subset EG(G_j)$ (where $G_i \neq G_j$).*
  - *Drop each G_i from $GEN(S)$ if for all k in $EG(G_i)$ there exists another G_j such that k is also in $EG(G_j)$ (where $G_i \neq G_j$). This means that every positive example covered by the generalized string G_i is also covered by another generalized string.*

Fig. 3 Generalization algorithm for a set of strings

Example 1. - Postfix a: Let us assume that the following clauses are given as positive examples of the single-arity predicate p that represents strings ending with substring a.

```
1. p(ba).
2. p(cda).
3. p(a).
4. p(aa).
5. p(faga).
```

The set of the arguments of the positive examples is $S = \{ba, cda, a, aa, faga\}$, *GEN(S)* is computed as follows by the algorithm in Fig. 3:

– Initially, $GEN(S) = S = \{ba, cda, a, aa, faga\}$. The following table shows the current content of *GEN(S)* together with EG function values for each string in *GEN(S)* and the examples used in the generalization of that string.

$GEN(S)$	ba	cda	a	aa	$faga$
EG	{1}	{2}	{3}	{4}	{5}
$ExUsed$	{1}	{2}	{3}	{4}	{5}

– For every pair of strings with a generalization, we add their generalization into *GEN(S)*. The SG of all pairs of ba, cda, and a is Xa, and the SG of aa and $faga$ is $XaYa$. Now, *GEN(S)* will be as follows.

$GEN(S)$	Xa	$XaYa$	ba	cda	a	aa	$faga$
EG	{1, 2, 3, 4, 5}	{4, 5}	{1}	{2}	{3}	{4}	{5}
$ExUsed$	{1, 2, 3}	{4, 5}	{1}	{2}	{3}	{4}	{5}

– Since $EG(ba)$, $EG(cda)$, $EG(a)$, $EG(aa)$, $EG(faga)$, and $EG(XaYa)$ are subsets of $EG(Xa)$, we drop ba, cda, a, aa, $faga$, and $XaYa$ from *GEN(S)*. So, *GEN(S)* will be:

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

– Since there is no pair of strings in *GEN(S)* with a generalization, we are done. Thus, the generalized clause of this predicate will be.

```
p(Xa).
```

Example 2. - Postfix a or b: Let us assume that the following clauses are given as positive examples of the single-arity predicate p that represents strings ending with substring a or b.

```
1. p(ca).
2. p(aa).
3. p(da).
4. p(fa).
5. p(gb).
6. p(bb).
7. p(cb).
```

– Initially, $GEN(S) = \{ca, aa, da, fa, gb, bb, cb\}$. The following table shows the current content of *GEN(S)* together with *EG* function values for each string in *GEN(S)*.

GEN(S)	ca	aa	da	fa	gb	bb	cb
EG	{1}	{2}	{3}	{4}	{5}	{6}	{7}
ExUsed	{1}	{2}	{3}	{4}	{5}	{6}	{7}

– For every pair of strings with a generalization, we add their generalization into *GEN(S)*. Since the SG of all pairs of ca, da, and fa is Xa, the SG of ca and cb is cX, and the SG of gb and cb is Xb, *GEN(S)* will be as follows.

GEN(S)	Xa	cX	Xb	ca	aa	da	fa	gb	bb	cb
EG	{1, 2, 3, 4}	{1, 7}	{5, 6, 7}	{1}	{2}	{3}	{4}	{5}	{6}	{7}
ExUsed	{1, 3, 4}	{1, 7}	{5, 7}	{1}	{2}	{3}	{4}			

– Since $EG(ca)$, $EG(aa)$, $EG(da)$, $EG(fa)$, $EG(gb)$, $EG(bb)$, and $EG(cb)$ are subsets of other sets of *EG* function in the table above, we drop ca, da, aa, fa, ga, bb, and cb from *GEN(S)*. So, *GEN(S)* will be:

GEN(S)	Xa	cX	Xb
EG	{1, 2, 3, 4}	{1, 7}	{5, 6, 7}
ExUsed	{1, 3, 4}	{1, 7}	{5, 7}

– Since all members of $EG(cX)$ are covered by other sets of *EG* function in the table above, we drop cX from *GEN(S)*. Thus *GEN(S)* will be:

GEN(S)	Xa	Xb
EG	{1, 2, 3, 4}	{5, 6, 7}
ExUsed	{1, 3, 4}	{5, 7}

– Since there is no pair of strings in *GEN(S)* with a generalization, we are done. Thus, the generalized clauses of this predicate will be.

```
p(Xa).
p(Xb).
```

4.2. Generalization of multiple-arity predicates

The multiple-arity predicates are generalized in a similar fashion as single-arity predicates. Although the unique match sequences for the argument pairs are found separately, the generalization is performed for all arguments at the same time after the unique match sequences are combined as a single unique match sequence. With small changes in the definition of *GEN* function, the generalization algorithm given in Fig. 3 for the single arity predicates is also used for the multiple-arity predicates.

The generalization $GEN((\alpha_1, \ldots, \alpha_n), (\beta_1, \ldots, \beta_n))$ of the arguments of two clauses $p(\alpha_1, \ldots, \alpha_n)$ and $p(\beta_1, \ldots, \beta_n)$ of an n-arity predicate p is found as follows:

– First, for each pair of α_i and β_i, the unique match sequence $UMS(\alpha_i, \beta_i)$ is found. If a unique match sequence exists for each pair, the unique match sequence $UMS((\alpha_1, \ldots, \alpha_n), (\beta_1, \ldots, \beta_n))$ for all arguments is defined as $UMS(\alpha_1, \beta_1) : \ldots : UMS(\alpha_n, \beta_n)$ assuming that the symbol ':' is a new symbol. The new symbol ':' is treated as a similarity in the match sequence to mark argument boundaries. If $UMS(\alpha_i, \beta_i)$ does not exist for the ith pair of the arguments, we say that the unique match sequence between $(\alpha_1, \ldots, \alpha_n)$ and $(\beta_1, \ldots, \beta_n)$

does not exist. In this case, these two clauses are not generalized.
- To find $SG((\alpha_1, ..., \alpha_n), (\beta_1, \ldots, \beta_n))$, we find a specific instance $SI of UMS((\alpha_1, \ldots, \alpha_n), (\beta_1, \ldots, \beta_n))$ of $UMS((\alpha_1, \ldots, \alpha_n), (\beta_1, \ldots, \beta_n))$. Then we replace all variables in this specific instance to create SG for these arguments.
- Thus, the generalization $GEN((\alpha_1, \ldots, \alpha_n), (\beta_1, \ldots, \beta_n))$ for the arguments $(\alpha_1, \ldots, \alpha_n)$, and $(\beta_1, \ldots, \beta_n)$ is SG $((\alpha_1, \ldots, \alpha_n), (\beta_1, \ldots, \beta_n))$ if SG $((\alpha_1, \ldots, \alpha_n), (\beta_1, \ldots, \beta_n))$ exists, and it is not in the most general string $X_1 : \ldots : X_n$ where each X_i is a different variable. Otherwise, the clauses do not have a generalization.

Let us assume that we have a set of positive examples of an n-arity predicate p, and $S = \{\alpha_{1,1} : \ldots : \alpha_{1,n}, \ldots, \alpha_{m,1} : \ldots : \alpha_{m,n}\}$ is the set of the arguments of these positive examples. The generalization set *GEN(S)* for S is found by using the generalization algorithm in Fig. 3.

Example 3. - Substring: Let us assume that the following clauses are given as positive examples of the binary predicate p that represents the substring relation.

```
1. p(a,bac).
2. p(d,fde).
```

- Initially, $GEN(S) = \{a : bac, d : fde\}$. The following table shows the current content of *GEN(S)* together with *EG* function values for each string in *GEN(S)*, and the examples used in the generalization of that string.

$GEN(S)$	$a : bac$	$d : fde$
EG	{1}	{2}
ExUsed	{1}	{2}

- For every pair of strings with a generalization, we add their generalization into *GEN(S)*. Since the SG of $a : bac$ and $d : fde$ is $X : YXZ$, *GEN(S)* will be as follows.

$GEN(S)$	$X : YXZ$	$a : bac$	$d : fde$
EG	{1, 2}	{1}	{2}
ExUsed	{1, 2}	{1}	{2}

- Since $EG(a : bac)$ and $EG(d : fde)$ are subsets of $EG(X : YXZ)$ in the table above, we drop $a : bac$, and $d : fde$ from *GEN(S)*. So, *GEN(S)* will be:

$GEN(S)$	$X : YXZ$
EG	{1, 2}
ExUsed	{1, 2}

- Since there is no pair of strings in *GEN(S)* with a generalization, we are done. Thus the generalized clauses of this predicate will be.

```
p(X,YXZ).
```

5. A learning heuristic based on SGs of strings

In this section, we describe a learning heuristic that is used in the induction of a 2-arity recursive predicate whose structure is known before the learning phase. It is also assumed that the alphabet of strings in the first argument position is different from the alphabet of strings in the second argument position. During the generalization process from the given set of positive examples, some additional heuristics are used in addition to the generalization methods used in the generalization process described in the previous section. The learned predicate is a recursive procedure. That is, the body of the learned predicate may contain recursive calls to itself and calls to the predicate *append*. In addition to the generalized clauses whose bodies contain recursive calls to this predicate, the ground unit clauses are also learned during the generalization process.

In Section 5.1, this learning heuristic is described. The structure of the 2-arity recursive predicate induced by this learning heuristic is very similar to the structure of translation templates used in an example-based machine translation system. In Section 5.2, we describe how to use this learning heuristic in a real-life example based on a machine translation system.

5.1. Learning heuristic

The predicate which is learned by the learning heuristic described in this section is a 2-arity recursive predicate and it is assumed that its structure is known before the learning phase. A given positive example of this 2-arity recursive predicate is $p(\alpha, \beta)$ where α is a string of an alphabet A and β is a string of an alphabet B. A learned predicate definition can be in a unit clause, or an if-then rule in the following form:

$$p(T^a, T^b) \textbf{ if } p(X_1, Y_1) \textbf{ and } \ldots \textbf{ and } p(X_n, Y_n)$$

where $n \geq 1$, T^a is a string of symbols in the alphabet A and variables $X_1, \ldots, X_n$; T^b is a string of symbols in the alphabet B and variables $Y_1, ..., Y_n$; and both T^a

and T^b must contain at least one symbol. For example, if the alphabet $A = \{a, b, c, d, e, f, g, h\}$ and the alphabet $B = \{t, u, v, w, x, y, z\}$, the following rules can be definitions of the learned predicate.

- $p(abc, utv)$
- $p(abX_1c, uY_1)$ **if** $p(X_1, Y_1)$
- $p(aX_1bX_2c, Y_2uvY_1)$ **if** $p(X_1, Y_1)$ **and** $p(X_2, Y_2)$
- $p(aX_1X_2b, Y_2vY_1)$ **if** $p(X_1, Y_1)$ **and** $p(X_2, Y_2)$

A generalized clause is a generalization of a set of positive examples, where certain components are generalized by replacing them with variables and establishing bindings between these variables. For example, in the second example above, abX_1c represents all strings starting with ab and ending with c where X_1 represents a non-empty string on the alphabet A, and uY_1 represents all strings starting with u where Y_1 represents a non-empty string on the alphabet B. The generalized clause says that a string of the alphabet A in the form of abX_1c corresponds to a string of the alphabet B in the form of uY_1 given that X_1 corresponds to Y_1. If we know that the correspondence $p(de, vyz)$ exists, the correspondence $p\,(abdec, uvyz)$ can be inferred from the generalized clause.

A *unique match sequence* between the arguments of two examples $p(\alpha_1, \beta_1)$ and $p(\alpha_2, \beta_2)$ is a pair of two unique match sequences $M^a : M^b$ where M^a is the unique match sequence of α_1 and α_2, and M^b is the unique match sequence of β_1 and β_2. After the unique match sequence $M^a : M^b$ is found, the learning heuristic is applied to this unique match sequence in order to find a generalized clause for the examples by replacing differences with variables in an instance of this unique match sequence, and establishing bindings between the variables. In addition to a generalized clause, the learning heuristic can also infer unit clauses. Of course, if there is no unique match sequence for the arguments of the examples, the learning heuristic cannot be applied to them. The learning heuristic also requires extra conditions on the instance of the unique match sequence which is used in the learning process. The learning heuristic can infer new clauses from an instance $MI^a : MI^b$ of the unique match sequence $M^a : M^b$ of the examples $p(\alpha_1, \beta_2)$ and $p(\alpha_2, \beta_2)$, if this instance satisfies the following conditions:

1. Both MI^a and MI^b must contain at least one similarity and one difference.
2. Both MI^a and MI^b cannot contain a difference with an empty constituent.
3. Both MI^a and MI^b must contain n differences where $n \geq 1$. In other words, they must contain equal number of differences.
4. Each difference in MI^a must correspond to a difference in MI^b, and a difference cannot correspond to more than one difference on the other side. Thus, we will have n corresponding differences.

If there is just one difference on both sides, they should correspond to each other (i.e. the fourth condition is trivially satisfied). But, if there is more than one difference on both sides, we need to look at previously learned unit clauses to determine the corresponding differences. For example, if there are two differences D_1^a and D_2^a in M^a, and two differences D_1^b and D_2^b in M^b; we cannot determine whether D_1^a corresponds to D_1^b or D_2^b without using prior knowledge. Now, let us assume that the correspondence between the differences D_1^a and D_1^b has been learned earlier. In this case D_2^a must correspond to D_2^b. In general, if the $n-1$ corresponding differences have been learned earlier, the last two differences must correspond to each other.

We say that the corresponding difference between the differences $D^a = (D_1^a, D_2^a)$ and $D^b = (D_1^b, D_2^b)$ has been learned, if the following two unit clauses have been learned earlier.

$$p\left(D_1^a, D_1^b\right)$$
$$p\left(D_2^a, D_2^b\right)$$

Now, let us assume that the differences in M^a are D_1^a,....,D_n^a and the differences in M^b are D_1^b,...,D_n^b where D_i^a corresponds to D_i^b. In this case, the first $n-1$ corresponding differences have been learned earlier, and the correspondence between the differences D_n^a and D_n^b is being inferred now. The learning heuristic replaces each D_i^a with the variable X_i to create a specific generalization SG^a from M^a, and each D_i^b with the variable Y_i to create a specific generalization SG^b from M^b. Then, the following generalized clause is induced by the learning heuristic.

$$p(SG^a, SG^b) \textbf{ if } p(X_1, Y_1) \textbf{ and } \ldots \textbf{and } p(X_n, Y_n)$$

In addition, the following two unit clauses are learned from the inferred correspondence between the difference $D_n^a = (D_{n,1}^a, D_{n,2}^a)$ and $D_n^b = (D_{n,1}^b, D_{n,2}^b)$.

$$\mathrm{p}\left(D_{n,1}^a, D_{n,1}^b\right)$$
$$\mathrm{p}\left(D_{n,2}^a, D_{n,2}^b\right)$$

Example 4. - Learning from unique match sequences with single differences:

Let us assume that

$$p(abc, vwxyz)$$
$$p(abef, tuxyz)$$

are two positive examples. The unique match sequence for the arguments of these examples will be

$$ab\,(c, ef)\ :\ (vw, tu)\,xyz.$$

Since the unique match sequence is an instance that satisfies the four conditions, the following clauses can be learned from that unique match sequence.

$p(ab\,X_1, Y_1\,xyz)$ **if** $p(X_1, Y_1)$
$p(c, vw)$
$p(ef, tu)$

where abX_1 is the SG of abc and $abef$, and Y_1xyz is the SG of $vwxyz$ and $tuxyz$.

Example 5. - Learning from unique match sequences with multiple differences:

Let us assume that

$p(bac, vwxy)$
$p(daef, tuxz)$

are two positive examples. The unique match sequence for the arguments of these examples will be

$$(b, d)\,a\,(c, ef)\ :\ (vw, tu)\,x\,(y, z).$$

This unique match sequence satisfies the first three conditions because it has two differences on both sides. But we do not know whether it satisfies the fourth condition. We cannot know whether the difference (b, d) on the left hand side corresponds to the difference (vw, tu) or to the difference (y, z) on the right hand side without using prior knowledge. Let us assume that the clauses in Example 4 have been learned earlier. Since we have learned that the difference (c, ef) corresponds to the difference (vw, tu) in Example 4, the difference (b, d) must correspond to the difference (y, z). Thus, all difference correspondings are found in our unique match sequence. The learning heuristic infers the following generalized clause by generalizing the given examples, and the next two unit clauses from the corresponding difference between (b, d) and (y, z).

$p(X_1\,a\,X_2, Y_2\,x\,Y_1)$ **if** $p(X_1, Y_1)$ **and** $p(X_2, Y_2)$
$p(b, y)$
$p(d, z)$

where X_1aX_2 is the SG of bac and $daef$, and Y_2xY_1 is the SG of $vwxy$ and $tuxz$.

5.2. Application to example-based machine translation

The learning heuristic described in Section 5 can be used in the learning of translation templates from a given bilingual corpus for two natural languages. In fact, it is successfully used as a part of the learning module of an Example-Based Machine Translation System (EBMT) between English and Turkish, and the details of this EBMT system can be found in [1, 2, 4]. In the case of EBMT, the positive examples are the given translation examples, and the generalized clauses are the induced translation templates.

In order to learn translation templates, the learning heuristic should be applied to every pair of translation examples in the system. The translation examples are treated as atomic translation templates. In fact, the learning procedure starts from these examples. Learning should continue until no more new templates can be learned from the atomic translation templates. The learned translation templates can be used in the translation of other sentences in both directions.

The learning heuristic can work on the surface level representation of sentences. However, in order to generate useful templates, it is helpful to use the lexical representation. In this case, the set of all root words, prefixes, and suffixes in a natural language are treated as the alphabet of that language for our purposes. Thus, a natural language is treated as the set of all meaningful strings on that alphabet. Normally, the given translation examples should be sentences of two natural languages, but they can also be phrases in those languages. Of course, morphological analyzers will be needed for both languages to compose the lexical forms of sentences.

An example-based machine translation system using this learning heuristic has two major parts: the learning module and the translation module. The learning module infers the translation templates from a given set of translation examples using the learning heuristic and the generalization algorithm described in this paper. A confidence factor can also be assigned to each translation template to indicate how good that translation template is. In order to assign these confidence factors [11], statistical techniques based on the information available in the sets of translation examples are used.

The translation module takes a sentence in the source language and produces a set of translation results in the target language. In order to translate a sentence from one language to another, first the lexical representation of the sentence is created using a morphological analyzer of the source language. Using the learned translation templates, possible translations of this sentence are found. The translation results are sorted with respect to the computed confidence factors of the results. At the end, we hope that the top results contain good translations and the correct translation is among them. After solutions are converted into surface level representations by using the morphological analyzer of the target language, a human expert can choose the correct solution by just looking at the top results, or the solution with the highest confidence factor can be given as the result of the translation.

Example 6. Learning between English and Turkish sentences

In order to explain the behavior of our learning heuristic on the actual natural language sentences, we give a simple learning example for translating sentences between English and Turkish. Assume that we have the translation examples `tt(I will drink water, su içeceğim)` and `tt(I will drink tea, çay içeceğim)` between English and Turkish. Their lexical representations are `tt(I will drink water, su iç+FUT+1SG)` and `tt(I will drink tea, çay iç+FUT+1SG)` where +FUT and +1SG denote future tense and first singular agreement morphemes in Turkish, respectively. For these two examples, the unique match sequence will be '`I will drink (water,tea) : (su,çay) iç+FUT+1SG`'. From this match sequence the learning heuristic learns the following three templates by creating SGs of the given sentences.

`tt(I will drink` X_1`,` Y_1 `iç+FUT+1SG)` **if** `tt(`X_1`,`Y_1`)`
`tt(water,su)`
`tt(tea,çay)`

In this example, we not only learn the general pattern in the first clause between English and Turkish, but also learn that `water` corresponds to `su` in Turkish, and `tea` corresponds to `çay`.

The learned translation templates can be used in translations in both directions. For example, if the correspondence `tt(orange juice, portakal suyu)` has been learned earlier, the sentence `I will drink orange juice` can be translated into the Turkish sentence `Portakal suyu içeceğim` using the learned translation templates.

6. Learning with background knowledge

In this section, we present an extension to our learning algorithm to demonstrate how the background knowledge can be used during learning. Here, we assume that we have single-arity predicates *b1*,*b2*,...,*bn* as background knowledge in addition to the predicate *append*. These predicates may appear in the bodies of the induced clauses.

In the last step of the specific generalization algorithm in Fig. 2, the differences are replaced with variables. In this extension, we replace the differences with typed variables. The type of a variable is a background predicate. A difference (D_1, D_2) is replaced with a typed variable X^{bi} if the goals $bi(D_1)$ and $bi(D_2)$ are finitely provable with respect to the given definition of the background predicate bi.

Let us assume that the following clauses are given as background knowledge.

```
b1(a).
b1(c).
```

The difference (a, c) in the match sequence $f(a, c)g$ is replaced with the typed variable X^{b1}, and the specific generalization $fX^{b1}g$ is found for this match sequence. The generalized string $fX^{b1}g$ with a typed variable X^{b1} is more specific than the generalized string fXg with an untyped variable X because the set of ground strings represented by the first one is a subset the set of ground strings represented by the second one.

Since we can have typed variables in the generalized strings in addition to untyped variables, the matching algorithm should also deal with these typed variables. The match algorithm treats the variables with same types as same tokens. For example, the match sequence of the generalized strings aX^{b1} and bY^{b1} will be $(a, c)Z^{b1}$ where X^{b1} and Y^{b1} are treated as the same token (a similarity) and they are represented by a new typed variable Z^{b1} in the match sequence. Thus, typed variables can be part of similarities in the match sequence. As a result, we can get a generalized string that may only contain variables and at least one of these variables is a typed variable.

Example 7. Grammar learning. Now, we will use this extension in an example that learns a simple grammar from the given English sentences. A similar example is also used by Muggleton in his Progol system [9]. In fact, our background predicates may correspond to his background single-arity predicates with positive mode declarations.

Let us assume that the following four predicates are given as background knowledge.

```
tverb(hits).    np (a man).       np (a cat).
tverb(walks).   np (the man).     np (the cat).
tverb(takes).   np (a dog).       np (a boy).
                np (the dog).     np (the boy).
iverb(sleeps).  np (a house).     np (a room).
iverb(walks).   np (the house).   np (the room)
                np (a ball).      np (a picnic).
prep(at).       np (the ball).
prep(to).
prep(in).
```

In this example, we use a finite set of clauses to represent the predicate `np` for simplicity purposes, but it could be defined using some auxiliary predicates. Now let us also assume that we have the following clauses representing simple English sentences as positive examples.

```
1.  s(a man sleeps).
2.  s(the boy sleeps).
3.  s(the dog walks).
4.  s(a boy walks).
5.  s(a man walks a dog).
6.  s(the boy walks the cat).
7.  s(the man hits the ball).
8.  s(a boy hits a dog).
9.  s(the man hits the ball at the house).
10. s(a boy hits a dog at a picnic).
```

```
11. s(the man takes the ball to the house).
12. s(a boy takes a dog to a room).
```

Initially, the generalization set for the arguments will contain the arguments of all positive examples of the predicate `s`. After the first pass of the learning algorithm, the following generalized strings will be in the generalization set.

a. X^{np} `sleeps`	from examples 1 & 2
b. X^{np} `walks`	from examples 3 & 4
c. X^{np} `walks` Y^{np}	from examples 5 & 6
d. X^{np} `hits` Y^{np}	from examples 7 & 8
e. X^{np} `hits` Y^{np} `at` Z^{np}	from examples 9 & 10
f. X^{np} `takes` Y^{np} `to` Z^{np}	from examples 11 & 12

The second pass of the learning algorithm will induce the following generalized strings from the generalized strings above.

$X^{np}\ Y^{iverb}$	from generalized strings a & b
$X^{np}\ Y^{tverb}\ Z^{np}$	from generalized strings c & d
$X^{np}\ Y^{tverb}\ Z^{np}\ V^{prep}\ W^{np}$	from generalized strings e & f

Thus, the learned clauses will contain the following three clauses.

$s(XY)$
  if $np(X)$ **and** $iverb(Y)$
$s(XYZ)$
  if $np(X)$ **and** $tverb(Y)$ **and** $np(Z)$
$s(XYZVW)$
  if $np(X)$ **and** $tverb(Y)$ **and** $np(Z)$ **and** $prep(V)$
  and $np(W)$

Since these three clauses cover all 12 examples, the learned clause set can only contain these three clauses.

7. Conclusion

In this paper, we introduced an ILP technique which is based on SGs of strings to reduce over-generalization problem in the learning process of predicates with string arguments. The over-generalization can be a serious problem when the learning is done from only positive examples. For example, just using Plotkin's RLGGs [13] for the predicates with string arguments will not be acceptable because of this over-generalization problem. The learning technique described in this paper does not cause over-generalization and still performs good generalizations from the given positive examples.

We believe that humans learn general sentence patterns using similarities and differences between many different example sentences that they are exposed to. This observation led us to the idea that general sentence patterns can be taught to a computer using learning heuristics based on similarities and differences in sentence pairs. In this sense, our learning technique is close to how humans learn languages from examples. In this paper, we tried to extend the usage of similarities and differences between strings in the generalization process of strings.

The ILP technique described in this paper can be used for the induction of predicates whose bodies may contain calls to predicate *append* which is the only predicate in background knowledge. Later, we described an extension to our learning process so that single-arity background predicates can be used in the learning process. We are also investigating other learning techniques, so that the bodies of the induced predicates may refer to multiple-arity predicates in the background knowledge. Here, we also described a learning heuristic to be used in the induction of a recursive predicate in a certain form. In general, if the pattern of a predicate is known, the special learning heuristics based on the match sequences can be developed to be used in the induction process of that predicate. We believe that the learning heuristics based on match sequences are very useful techniques in the generalization of predicates with string arguments. We are investigating other learning techniques in which the user tells the system which predicates are given as background knowledge, and the learning process can use the given predicates in the body of the induced predicates.

References

1. Cicekli I, Güvenir HA (2003) Learning translation templates from bilingual translation examples. In: Carl M, Way A (eds), Recent advances in example-based machine translation. The Kluwer Academic Publishers, Boston, pp 255–286
2. Cicekli I, Güvenir HA (2001) Learning translation templates from bilingual translation examples. Appl. Intell. 15(1)
3. Dzeroski S, Cussens J, Manandhar S (2000) An Introduction to inductive logic programming and learning language in logic, Lecture notes in artificial intelligence 1925, Springer-Verlag, pp 3–35
4. Güvenir HA, Cicekli I (1998) Learning translation templates from examples. Inform. Syst 23(6):353–363
5. Mooney RJ, Califf ME (1995) Induction of first-order decision lists: Results on learning the past tense of english verbs. J Artif Intell Res 3:1–24
6. Mooney RJ (1997) Inductive logic programming for natural language processing. In: Muggleton S. (ed), Proceedings of the 6th International Workshop on Inductive Logic Programming, Springer-Verlag, Berlin, pp 3–21
7. Muggleton S (2001) Learning from positive data. Machine Learning
8. Muggleton S (1999) Inductive logic programming: issues, results and the challenge of learning language in logic. Artif Intell 114(1–2):283–296
9. Muggleton S (1995) Inverse entailment and progol. New Gener Comp 13:245–286
10. Muggleton S, Feng C (1992) Efficient induction of logic programs. In: Muggleton S. (ed), Inductive Logic Programming, Academic Press, London, pp 281–298
11. Öz Z, Cicekli I (1998) Ordering translation templates by assigning confidence factors. In: Lecture notes in computer science 1529, Springer Verlag, pp 51–61
12. Plotkin GD (1970) A note on inductive generalisation. In: Meltzer M, Michie D. (eds), Machine Intelligence 5. Elsevier North-Holland, New York, pp 153–163

13. Plotkin GD (1971) Automatic Methods of Inductive Inference, Ph.D. Thesis, Edinburgh University, Edinburgh, 1971.
14. Quinlan JR, Cameron RM (1995) Induction of logic programs: Foild and related systems. New Gener Comp 13:287–12.

Ilyas Cicekli received a Ph.D. in computer science from Syracuse University in 1991. He is currently a professor of the Department of Computer Engineering at Bilkent University. From 2001 till 2003, he was a visiting faculty at University of Central Florida. His current research interests include example-based machine translation, machine learning, natural language processing, and inductive logic programming.

Nihan Kesim Cicekli is an Associate Professor of the Department of Computer Engineering at the Middle East Technical University (METU). She graduated in computer engineering at the Middle East Technical University in 1986. She received the MS degree in computer engineering at Bilkent University in 1988; and the PhD degree in computer science at Imperial College in 1993. She was a visiting faculty at University of Central Florida from 2001 till 2003. Her current research interests include multimedia databases, semantic web, web services, data mining, and machine learning.