GENERALIZATION OF PREDICATES WITH STRING ARGUMENTS

A THESIS
SUBMITTED TO THE DEPARTMENT OF COMPUTER ENGINEERING
AND THE INSTITUTE OF ENGINEERING AND SCIENCE
OF BILKENT UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
MASTER OF SCIENCE

By
Göker Canıtezer
January, 2002

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.

Advisor: Prof. H. Altay Güvenir

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.

Co-Advisor: Asst. Prof. İlyas Çiçekli

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.

Assoc. Prof. Tuğrul Dayar

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.

Asst. Prof. Uğur Güdükbay

Approved for the Institute of Engineering and Science:

Prof. Mehmet Baray
Director of the Institute of Engineering and Science

ABSTRACT

GENERALIZATION OF PREDICATES WITH STRING ARGUMENTS

Göker Canıtezer
M.S in Computer Engineering
Supervisors: Prof. H. Altay Güvenir,
Asst. Prof. İlyas Çiçekli
January, 2002

String/sequence generalization is used in many different areas such as machine learning, example-based machine translation and DNA sequence alignment. In this thesis, a method is proposed to find the generalizations of the predicates with string arguments from the given examples. Trying to learn from examples is a very hard problem in machine learning, since finding the global optimal point to stop generalization is a difficult and time consuming process. All the work done until now is about employing a heuristic to find the best solution. This work is one of them. In this study, some restrictions applied by the SLGG (Specific Least General Generalization) algorithm, which is developed to be used in an example-based machine translation system, are relaxed to find the all possible alignments of two strings. Moreover, a Euclidian distance like scoring mechanism is used to find the most specific generalizations. Some of the generated templates are eliminated by four different selection/filtering approaches to get a good solution set. Finally, the result set is presented as a decision list, which provides the handling of exceptional cases.

Keywords: generalization, slgg, sequence alignment

ÖZET

KARAKTER DİZİSİ ARGÜMANLI ÖNERGELERİN GENELLEŞTİRİLMESİ

Göker Canıtezer
Yüksek Lisans, Bilgisayar Mühendisliği
Tez Yöneticileri: Prof. Dr. H. Altay Güvenir,
Yrd. Doç. Dr. İlyas Çiçekli
Ocak, 2002

Karakter dizilerinin genelleştirilmesi, makine öğrenimi, örnek tabanlı otomatik çeviri, DNA sırası hizalama gibi pek çok alanda kullanılmaktadır. Bu tezde, verilen örneklerden oluşan karakter dizisi parametreli yüklemlerin genelleştirilmiş hallerini bulan bir yöntem sunulmaktadır. Verilen örneklerden öğrenmeye çalışmak gerçekten zor bir problemdir, çünkü genelleştirmeyi durdurmak için global optimum noktayı bulmak zor ve zaman alıcı bir işlemdir. Şu ana kadar yapılan bütün işler, en iyi çözümü bulmak için kullanılan deneme-yanılma yöntemleridir. Bu çalışma da onlardan birisidir. Bu projede, iki karakter dizisinin mümkün olan bütün hizalanmalarını bulmak için ÖEGG (Özel Enaz Genel Genelleştirme) algoritmasında uygulanan bazı kısıtlamalar kaldırılmıştır. Ek olarak, en özel genelleştirmeleri bulmak için Euclid uzaklığına benzer bir puanlandırma mekanizması kullanılmıştır. Üretilen kalıplarlardan bazıları dört farklı seçim/eleme yöntemi ile elenmiştir. Son olarak, sonuç kümesi karar listesi halinde sunularak, istisnai durumların yakalanması sağlanmıştır.

Anahtar Kelimeler: genelleştirme, öegg, sıra hizalama

Acknowledgement

I would like to express my deepest gratitude to Asst. Prof. İlyas Çiçekli for his supervision, guidance, suggestions and invaluable encouragement throughout the development of this thesis.

I would like to thank to committee members for reading this thesis and their comments.

I would like to thank to all my friends for their encouragement and logistic support.

I have to thank my boss Dr. Semih Çetin, head of cyberSoft and Dr. Mesut Göktepe, project manager for their continuous support during my MS studies.

I would like to thank to my parents, my grandmother and grandfather and all other relatives who believe in me and support me.

To My Family

Contents

List of Figures

List of Tables

Chapter 1

INTRODUCTION

The string generalization problem is a subtopic of machine learning (ML), and inductive logic programming (ILP). Like many other real world problems there are many examples and learning from these examples is going between *specialization* (memorizing examples) and *total generalization* (learning nothing).

Most of the approaches in ILP try to find the optimal solution, which means covering all the positive examples and not covering the negative examples. If there is noisy data, this becomes more difficult. There are two methods to overcome this problem. The first one is trying to generate negative examples and using them to specialize. The second one is beginning from the most specialized condition try to generalize up to some point, where all the positive examples are covered.

Inductive logic programming is an important subtopic of machine learning, which is used for the induction of Prolog programs from examples in the presence of background knowledge [1, 2]. Since first-order logic is very expressive, relational and recursive concepts that cannot be represented in the attribute/value representations assumed by most machine learning algorithms can be learned by ILP methods. ILP methods have been successfully used in important applications such as predicting protein secondary structure [3], automating the construction of natural language parsers [4] and in small programs for sorting and list manipulation.

In order to explain the related topics easily in the following chapters, some background information will be helpful about sequence alignment, decision lists, translation templates and string generalization. Thus, the first section is about sequence alignment, its types and used algorithms. The second section explains the history and the advantages of decision lists. Information about translation templates is given in the third section and finally in the fourth section introductory information about string generalization used in this work is given.

1.1 Sequence alignment

Sequence alignment is one of the most important tools in molecular biology. It has been used extensively in discovering and understanding the functional and evolutionary relationships among genes and proteins [5, 6]. There are two classes of alignment algorithms: algorithms without allowing gaps in alignments, e.g., BLAST and FASTA [6, 7], and algorithms with gaps, e.g., the Needleman-Wunsh algorithm [6], and the Smith-Waterman algorithm [8]. The simpler gapless alignment as it was implemented in the original BLAST [7, 9] is very fast and is widely used in large-scale database searches, since the results depend only weakly on the choice of the scoring systems [10], and the statistical significance of the results is well-characterized [1, 2, 3]. However, in order to detect weakly homologous sequences, gaps have to be allowed in an alignment [4] which leads to the more sophisticated Smith-Waterman algorithm [5].

Main difficulty for any alignment is the selection of scoring schemes/parameters. In a generic sequence matching problem, a score is assigned to each alignment of given sequences, based on the total number of matches, mismatches, gaps, etc. Maximization of this score defines an optimal alignment [6].

In addition to alignment methods between two sequences, multiple sequence alignment is another fundamental and most challenging problem in computational molecular biology [6]. It plays an essential role in the solution of many problems such as searching for highly conserved subregions among a set of biological sequences and finding the evolutionary history of a family of species from their molecular sequences [11].

An important approach to multiple sequence alignment is the tree alignment method. The biological interpretation of the model is that the given tree represents the evolutionary history which has created the molecular (DNA, RNA or amino acid) sequence written at the leaves of the tree. The leaf sequences represent the existing organisms today, and the internal nodes of the tree are the ancestral organisms that may have existed [11].

The tree alignment problem is known to be NP-HARD [12]. Many heuristic algorithms have been proposed in the literature [13, 14] and some approximation algorithms with guaranteed relative error bounds have been reported. Thus, the more accurate the algorithm is, the more time it consumes [11].

As it can be guessed, there are many different approaches to find the alignments of molecular sequences. Some of them use local similarity matrices, e.g., PAM and BLOSUM [15]. Some others use dynamic programming to find the highest scoring global alignment in the presence of gaps [5]. Some kind of shortest path algorithms and sequence graphs are used for some heuristics to find tree alignments [15].

1.2 Decision Lists

Decision lists are first introduced by Rivest in 1987 [16] as a new technique for representing concepts. In [16], decision lists are used for strict generalization of concept representation techniques, e.g., k-CNF, k-DNF, kDT. A decision list may be thought of as an extended "if-then-elseif-…-else" rule. In other words, a decision list is defining the general pattern with exceptions. The exceptions correspond to the early items in the decision list, whereas the more general patterns correspond to the later items [16]. Rivest used decision lists for learning boolean functions. First usage of decision lists for inductive logic programming is done by Mooney and Califf [17]. In this paper, it is expressed that some ILP techniques make some important assumptions that restricts their application, such as:

1. Background knowledge is provided in extensional form as a set of ground literals.
2. Explicit negative examples of the target predicate are available.
3. The target program is expressed in "pure" prolog where clause-order is irrelevant and procedural operators such as cut (!) are disallowed.

However, each of these assumptions brings significant limitations. One of the limitations that is relevant to us is [18]:

"- Concise representation of many concepts requires the use of clause-ordering and/or cuts."

Mooney finds solution to these problems by introducing FOIDL (First Order Inductive Decision List). In FOIDL, a learned program can be represented as a first-order decision list, an ordered set of clauses each ending with a cut. This representation is very useful for problems that are best represented as general rules with specific exceptions [17].

When answering an output query, the cuts simply eliminate all but the first answer produced when trying the clauses in order. In the original algorithm of [16], rules are learned in the order they appear in the final decision list, e.g., new rules are appended to the end of the list as they are learned. However, [19] argues for learning decision lists in the reverse order since most preference functions tend to learn more general rules first. These are best positioned as default cases towards the end. FOIDL learns an ordered sequence of clauses in reverse order, resulting in a program which produces only the first output generated by the first satisfied clause. In our work, order of learning is not important since the learned clauses are sorted with respect to their specialization (fragmentation) score.

1.3 Translation Templates

In the translation process, providing the correspondences between the source and the target language is a very difficult task in exemplar-based machine translation. Although, manual encoding of the translation rules has been achieved by Kitano [20], when the corpus is large; it becomes a complicated and error-prone task. Therefore, [21, 22] offer a technique in which the problem is taken as a machine learning task. Exemplars are stored in the form of templates that are generalized exemplars. The templates are learned by using translation examples and finding the correspondences between the patterns in the source and target languages. The heuristic of the translation template learning (TTL) [23] algorithm can be summarized as follows: given to translation pairs, if there are some similarities (differences) in the source language, then the corresponding sentences in the target language must have similar (different) parts, and they must be translations of the similar (different) parts of the sentences in the source language. Certain parts are replaced with variables to get a template, which is a generalized exemplar, by this method.

There are two types of translation templates: similarity translation template and difference translation template. In similarity translation templates, differences are replaced with variables, and in difference translation templates vice versa. TTL algorithm cannot learn anything if the number of similarities or differences in the match sequences are not equal [21].

In the first implementation, templates produced by STTL and DTTL are ordered according to the number of terminals in the source language [21, 22, 23]. The translation is a bi-directional process, so templates are ordered according to both languages. Since this criterion is not sufficient for large systems, [23] added confidence factor assignment in which each rule and some rule combinations are assigned weights.

In our work templates are assigned fragmentation and coverage scores. Coverage score may be thought of as a confidence factor.

1.4 String Generalization

String generalization is an important topic since it can be used in pattern matching, natural language processing, especially in Example-Based Machine Translation (EBMT) and genetics. By using string generalization, we aim to find rules about the orders and structures of sub-strings or character sequences of that language (natural or not, the alphabet of the language may include any symbol).

There are some generalization techniques. One of them is Plotkin's [24, 25] relative least general generalization (RLGG) technique, which is used by many ILP systems [26]. In [27] a new generalization technique, specific least general generalization, is introduced. SLGG is more powerful for finding the optimum generalized template [27]. For example, the GOLEM system uses RLGG schema and generalizes two clauses:

```
p([b,a]).
p([c,d,a]).
```

by creating p([A,B|C]) as the generalized clause. The generated clause covers the two given clauses but it can be noticed that it is an over generalization, since there are common parts, which is [a] in this example, that should have been captured by the generalization algorithm. Moreover, this common part is at the end of these lists. This should be captured too. In [30, 31, 36], to generalize two clause examples of a single-arity predicate with string arguments, SLGG of two strings is used. For the example above, SLGG technique generalizes as the following:

```
p(L) :- append(L1,[a],L).
```

by assuming that append predicate is in the background knowledge.

If the system only learns the given examples, which means memorizes the examples, it is the most specialized point. If it accepts all examples, it is the total generalization point, which means learning nothing. Thus, our algorithm should find the optimal stopping point between the total generalization point and the most specialized point. For example we have two strings such as:

I will come home later
He will come later

After the generalization of these strings, we should learn the template [28], generalized form of the strings:

X will come Y later

This template means that our language has a structure that has two variables X, Y and a constant string "will come __ later". A similar work was done by Cicekli [29], but it has restrictions called minimal match sequence. For example, the minimal match sequence of the strings *abcbd* and *ebfbg* will be *(a, e)b(c, f)b(d, g)*. But, strings *abcbd* and *ebf* cannot have a minimal match sequence because *b* occurs twice in the first string and b occurs only once in the second string [29, 30].

So our new algorithm should not omit these two strings, *abcbd* and e*bf*. Since there are two *b*'s in the first string which match to the *b* in the second string we can learn two different templates that are *(a,e)b(cbd,f)* and *(abc,e)b(d,f)*. Reader will notice that the structures of templates are same XbY, which can be combined into one template. If the strings *abbcd* and *abc* were used, *ab(b,ε)c(d, ε)* and *a(b, ε)bc(d, ε)* templates would be generated. Since the structures of these two are different we cannot combine them.

When we generate more than one template another problem arises. Which one is more valuable/correct? At this point our heuristic comes in and gives more points to the least fragmented template. a*baabcd* and abcd are strings and generated templates in the order of value are *(aba, ε)abcd, a(baa,ε)bcd, (ab,ε)a(a,ε)bcd, ab(aab,ε)cd.*

After generating templates, they are sorted in the order of most specialized to most generalized. This is similar to the decision list of FOIDL.

The remaining chapters are organized as follows. Chapter 2 is about related work, such as, FOIDL, sequence alignment and confidence factor assignment. Chapter 3 provides information on string generalization algorithm, scoring/sorting, selection sets. Applications in different domains can be found in Chapter 4, architecture and the implementation in Chapter 5 and finally conclusion and future work in Chapter 6.

Chapter 2

Related Work

String generalization process is related with many different areas of machine learning, since each level of generalization process deals with different algorithms and approaches. In this chapter, related work about these different levels and justification of our method can be found.

Generalization of predicates with string arguments has three main sub-processes. These are alignment, scoring and decision list generation. From the point of performance, alignment process is the bottleneck of the problem, since alignment problem is known to be NP-HARD [12]. Because of this, in this work we did not tried to hardly optimize the performance of the program. If it works in a reasonable time with reasonable amount of data it is enough for us, because the main goal of this project is finding an approach that generalize predicates with string arguments in an optimal level. Some approaches about optimization of aligning strings and/or character sequences can be found in Section 2.1. Scoring of generated templates is very important, since it affects the result set and the performance of the final work. Information about previously used heuristics for scoring is in Section 2.2. As it is stated in Chapter 1, a first-order decision list is very useful for problems that are best represented as general rules with specific exceptions [17]. Section 2.3 is about the decision lists and FOIDL. Finally, the last section is about the methods we used in this work.

2.1 Sequence Alignment

In Chapter 1, it is stated that there are many different approaches to find the alignments of the molecular sequences. Some of them use local similarity matrices, e.g., PAM and BLOSUM [31]. Some others use dynamic programming to find the highest scoring global alignment in the presence of gaps [5]. Some kind of shortest path algorithms and sequence graphs are used for some heuristics to find tree alignments [15].

In computational biology there are two types of alignment problem, i.e., gapless and gapped. Gapless alignment looks for similarities between two sequences $\alpha=a_1a_2\ldots a_N$ and $\beta=b_1b_2\ldots b_M$ of length N and M respectively. M and N are nearly equal. The letters a_i and b_j are taken from an alphabet of size c. A local gapless alignment, ***A***, of these two sequences consists two substrings; first substring $a_{i-l+1}\ldots a_{i-1}a_i$ of length l of sequence α and the second substring $b_{i-l+1}\ldots b_{j-1}b_j$ of sequence β of the same length of the first substring, l. Each such alignment is assigned a score. And the global optimal score is calculated by using dynamic programming [5, 32]. Although this approach is fast enough to find the alignments of sequences, alignment and scoring concepts in this approach do not meet our requirements.

In gapped alignment, a possible alignment *A* still consists of two substrings of the original sequences α and β. But now, these subsequences GATGC and GCTC may be aligned as GATGC and GCT-C using one gap. In Smith-Waterman local alignment, each such alignment *A* is assigned a score according to $S[A] = \Sigma\, s_{a,b} - \delta N_g$ where the sum is taken over all pairs of aligned letters, N_g is the total number of gaps in the alignment, and δ is an additional scoring parameter, the “gap cost”.

Example 2.1: We can see the differences of gapless and gapped alignments in this example. Let us assume that our sequences are GATGC and GCTC. Gapless alignment algorithm aligns as

```
GATGC
GCTC
*  *
```

G and T is found as the similar part. Gapped alignment algorithm aligns as

```
GATGC
GCT-C
*  * *
```

Note that gapped alignment finds three similar points (G, T, C), although gapless alignment finds two similar points. On the other hand, our algorithm finds all possible alignments, but the most valuable one for us is:

```
GATGC--
```

```
---GCTC
   **
```

Since it is less fragmented than the others.

2.2 Confidence Factor Assignment

Although, in most of the machine learning applications, we can find a kind of scoring scheme, in this chapter assigning confidence factor to the learned templates by the TTL algorithm is the main topic.

In [21] says that the algorithm orders the templates according to their specificities. Specificity is defined as: “Given two templates, the one that has a higher number of terminals is more specific than the other.” Note that, the specificity is defined according to the source language. For two-way translation, the templates are ordered once for each language as source.

Oz and Cicekli in [32] says that ordering according to the number of terminals of the templates is not sufficient for large systems. So they added a confidence factor assignment process in which each rule and some rule combinations are assigned weights. This process has three parts: Confidence factor assignment to facts, rules and rule combinations. Again in this approach confidence factors are assigned for left to right translation and right to left translation separately.

Ratio of the number of correctly covered source and target examples over total number of sources covered by source template gives the confidence factor of a fact or rule. For rules this is the partial confidence factor and during translation confidence factors of these rules are multiplied to find the real confidence factor. To find the confidence factor of the rule combinations a kind of Euclidian distance is used. Length of differences and similarities are used as dimensions [23].

2.3 GENERALIZATION

2.3.1 FOIL

In a nutshell, FOIL is a system for learning function-free Horn clause definitions of a relation in terms of itself and other relations. The program is actually slightly more flexible since it can learn several relations in sequence, allows negated literals in the definitions (using standard Prolog semantics), and can employ certain constants in the definitions it produces. FOIL's input consists of information about the relations, one of which (the target relation) is to be defined by a Horn clause program. For each relation, it is given a set of tuples of constants that belong to the relation. For the target relation, it might also be given tuples that are known not to belong to the relation; alternatively, the closed world assumption may be invoked to state that no tuples, other than those specified, belong to the target relation. Tuples known to be in the target relation will be referred to as positive tuples and those not in the relation as negative tuples. The learning task is then to find a set of clauses for the target relation that accounts for all the positive tuples while not covering any of the negative tuples [33].

The basic approach used by FOIL is an AQ-like covering algorithm [34]. It starts with a training set containing all positive and negative tuples, constructs a function-free Horn clause to "explain" some of the positive tuples, removes the covered positive tuples from the training set, and continues with the search for the next clause. When clauses covering all the positive tuples have been found, they are reviewed to eliminate any redundant clauses and reordered so that any recursive clauses come after the non-recursive base cases [33].

Perfect definitions that exactly match the data are not always possible, particularly in real-world situations where incorrect values and missing tuples are to be expected. To get around this problem, FOIL uses encoding-length heuristics to limit the complexity of clauses and programs. The final clauses may cover most (rather than none) of the negative tuples [33, 35].

2.3.2 GOLEM

Top-down methods such as Shapiro's MIS and Quinlan's FOIL [35], search the hypothesis space of clauses from the most general towards the most specific. MIS employs a breadth-first search through successive levels of a "clause refinement" lattice, considering progressively more complex clauses. To achieve greater efficiency Quinlan's FOIL greedily searches the same space guided by an information measure similar to that used in ID3. This gains efficiency at the expense of completeness [36].

Bottom-up algorithms based on inverting resolution [37] also have problems related to search strategies. In the framework of inverse resolution clauses are constructed by progressively generalizing examples with respect to given background knowledge. Search problems are incurred firstly since there may be many inverse resolvents at any stage, and secondly because several inverse resolution steps may be necessary to construct the required clause. Thus problems related to search hamper both top-down and bottom-up methods. In search based methods efficiency is gained only at the expense of effectiveness [36].

Plotkin's [38, 39] notion of relative least general generalization (rlgg) replaces search by the process of constructing a unique clause which covers a given set of examples. GOLEM is not interested in constructing a single clause which is the rlgg of positive examples, but rather a set of hypothesized clauses of positive examples. This set of hypothesized clauses cover all the positive examples and do not cover any negative examples.

As it is stated in Chapter 1, GOLEM system uses RLGG schema and generalizes two clauses:

```
p([b,a]).
p([c,d,a]).
```

by creating p([A,B|C]) as the generalized clause. Generated clause covers the two given clauses but it can be noticed that it is an over generalization. Since there are common parts, which is [a] in this example, should have been captured by the generalization algorithm. Moreover, this common part is at the end of these lists, and this should be captured too.

2.3.3 FOIDL

In [17] Mooney and Califf states that development of FOIDL was motivated by a failure they observed when applying existing ILP methods to a particular problem, that of learning the past tense of English verbs. They were unable to get reasonable results from FOIL or GOLEM since they make important assumptions that restrict their application, which are explained in Section 1.2. These assumptions bring significant limitations since:

1. An adequate extensional representation of background knowledge is frequently infinite or intractably large.
2. Explicit negative examples are frequently unavailable and an adequate set of negative examples computed using a closed-world assumption is infinite or intractably large.
3. Concise representation of many concepts requires the use of clause-ordering and/or cuts.

In FOIDL these limitations are overcame by the following properties:

1. Background knowledge is represented intentionally as a logic program.
2. No explicit negative examples are need to be supplied or constructed.
3. A learned program can be represented as a first-order decision list; an ordered set of clauses each ending with a cut. This representation is very useful for problems that are best represented as general rules with specific exceptions.

Chapter 3

Generalization of Predicates with String Arguments

In this chapter, a different approach for finding the generalized forms of the character sequences/strings is proposed. Although, there are tools to generalize the given positive data, we meet with the over generalization problem.

The main point of motivation of this work is extracting maximum information from a bilingual corpus to use it in an EBMT system [21, 22]. Many methods have been used to increase the performance of the translation system, and this work is one of them [23, 28, 32].

Following sections are about the algorithmic process to find the templates [22, 28]. We start by finding the optimal match sequences of two strings and go on with generalization process and converting optimal match sequences to SLGGs. Scoring and sorting of the single-arity and n-arity templates is another important topic that is described, and finally finding selection sets with different scoring mechanisms will be explained in the following lines.

3.1 Optimal Match Sequence

This part includes information about background information about similarity-difference concept and match sequence string for generating templates.

Cicekli, describes similarity and difference in [29] as follows:

A *similarity* between α_1 and α_2, where α_1 and α_2 are two non-empty strings of atoms, is a non-empty string β such that $\alpha_1 = \alpha_{1,1}\beta\alpha_{1,2}$ and $\alpha_2 = \alpha_{2,1}\beta\alpha_{2,2}$. A similarity represents a similar part between two strings.

A *difference* between α_1 and α_2 , where α_1 and α_2 are two non-empty strings of atoms, is a pair of two strings (β_1, β_2) where β_1 is a substring of α_1 and β_2 is a substring of α_2, the same atom cannot occur in both β_1 and β_2, and at least one of them is not empty. A difference represents a pair of differing parts between two strings.

In [27] minimal match sequence is used to generate templates, but in this project optimal match sequence is used. An *optimal match sequence* between two strings α_1 and α_2 is a sequence of similarities and differences between α_1 and α_2 such that the following conditions are satisfied by this match sequence:

1. Concatenation of similarities and the first constituents of differences must be equal to α_1.
2. Concatenation of similarities and the second constituents of differences must be equal to α_2.
3. An optimal match sequence should contain at least one similarity or one difference.
4. A similarity cannot follow another similarity, and a difference cannot follow another difference.

Reader may notice that 1st, 2nd and 4th conditions are same with the minimal match sequence. Moreover, every minimal match sequence is an optimal match sequence but every optimal match sequence is not a minimal match sequence.

To make clear; a few examples can be given:

Example 3.1:

α_1 = abcd

α_2 = acd

OMS = a(b,ε)cd

“a” and “cd” parts of the two strings are the same but α_1 includes “b”, but α_2 does not include any characters in the same position. Thus, difference part shows “b” and “ε” (empty string).

Example 3.2: What happens if same character occurs more than once?

α_1 = abcda

α_2 = acd

For these two strings we cannot represent them in one similarity difference string, since α_1 includes two “a”s. Both “a”s can match to the “a” in the α_2. Thus, we need two optimal match sequence:

OMS $_1$ = a(b, ε)cd(a, ε)

OMS $_2$ = (abcd, ε)a(ε, cd)

Example 3.3: Is the sequence of the characters important?

α_1 = abcd

α_2 = adc

Orders of the character sequences are really very important, since this process is an *alignment like* process. Differences in the order changes the alignment points, which causes different match sequences. Although, α_1 and α_2 includes same characters with the example 3.1, changing the order of “c” and “d” in α_2 causes different match sequences.

OMS $_1$ = a(bc, ε)d(ε, c)

OMS $_2$ = a(b, d)c(d, ε)

Example 3.4: Is “a(bc,cd)e” a valid optimal match sequence?

As explained in the beginning, “the same atom cannot occur in both β_1 and β_2”. Since “c” occurs both in β_1 and β_2 this is not a valid match sequence. The meaning of this OMS is:

α_1 = abce

α_2 = acde

Thus, there is only one generatable optimal match sequence, which is

OMS = a(b, ε)c(ε, d)e

Many examples could be given about optimal match sequences, but these four examples explain the most important characteristics of this concept. At this point, similar and different parts between sequence alignment and the optimal match sequence can be explained.

In sequence alignment process, two or more strings tried to be aligned. If the examples above are used for sequence alignment, their results would be similar to the following lines.

For Ex 1:

S_1 = `abcd`,

S_2 = `a-cd`

```
     * **
```

For Ex 2:

```
S1 = abcda
S2 = a-cd-
     * **
```

or

```
S1 = abcda--
S2 = ----acd
         *
```

For Ex 3:

```
S1 = abcd
S2 = adc-
     * *
```

or

```
S1 = abcd-
S2 = a--dc
     *  *
```

(Examples with long sequences can be examined in Appendix B)

Generated sequence alignment results changes with the used algorithm and its parameters [9]. Some algorithms do not allow gap generation between sequences, and some algorithms do [5, 6, 9]. Algorithms that allow gap generation has two main parameters called, gap creation penalty and gap extension penalty. These parameters are used for selecting the most wanted results, and this topic will be covered in the scoring part of the algorithm.

Stars under the aligned sequences show the similar/aligned parts. If these marked parts are taken with their different parts between them, then we can generate the minimal match sequences of these strings. This means that sequence alignment algorithms could be used to generate optimal match sequences. But, as stated above sequence alignment algorithms with gap generation uses some parameters for not generating all possible match sequences. It causes not generating all optimal match sequences of two strings.

In addition to this, there is a lot of work done on sequence alignment since 1970s [32]; as sequence alignment is one of the most commonly used computational tools of molecular biology. Thus, some of these algorithms could be adapted to find optimal match sequence in a fast way [32, 40].

3.2 Generalization Process/Generating Templates

Generalization is another important part of this thesis. After finding the match sequences, generalized templates should be generated. There are some generalization techniques. One of them is Plotkin's [24, 25] relative least general generalization (RLGG) technique, which is used by many ILP systems [26]. In [27] a new generalization technique, specific least general generalization, is introduced. SLGG is more powerful for finding the optimum generalized template [27]. For example, the GOLEM system uses RLGG schema and generalizes two clauses:

```
p([b,a]).
p([c,d,a]).
```

by creating p([A,B|C]) as the generalized clause. Generated clause covers the two given clauses but it can be noticed that it is an over generalization. Since there are common parts, which is [a] in this example, should have been captured by the generalization algorithm. Moreover, this common part is at the end of these lists, and this should be captured too. In [21, 22, 27], to generalize two clause examples of a single-arity predicate with string arguments, SLGG of two strings is used. For the example above, SLGG technique generalizes as the following:

```
p(L) :- append(L1,[a],L).
```

by assuming that append predicate is in the background knowledge.

In this work, SLGG is used with a slight modification. Generalization process can be defined as following:

If there is an optimal match sequence originated from similarity and difference sequences such as $(D_0)S_1D_1S_2D_2\ldots S_n(D_n)$ then generated template would be (V_0) $S_1V_1S_2V_2\ldots S_n(V_n)$, where V is a variable such as X, Y, Z, etc. There are some conditions the generated template must satisfy:

1.Same differences cannot be replaced with the same variables
2.V_0, V_1, V_2,…V_n are all different variables
3.There should be at least one similarity or variable
4.There should be a similarity between two variables

Reader may notice that only difference with the SLGG is the first conditions, which provides a little bit more generalization. In the original SLGG, same differences are replaced with the same variables. To find the SLGG of two strings :

- Firstly the optimal match sequences are found
- Secondly all differences are replaced with variables to create the SLGG.

If the strings are *abc* and *dbef*, their optimal match sequence will be *(a,d)b(c,ef)*, and the SLGG of these strings will be *XbY*. For the strings *abcd* and *abdc*, there will be two optimal match sequences *ab(c, ε)d(ε, c)* and *ab(ε, d)c(d, ε)*, and their SLGGs will be *abXdY* and *abXcY* respectively.

In order to show the whole process for the generalization of single arity predicates some examples can be given.

Example 3.5: In this example, the conditions, which there are more than two strings, will be examined. Let us assume that the following clauses are given as positive examples.[27]

1. p(ba).
2. p(cda).
3. p(a).

These clauses will be represented in Prolog as follows.

1. p([b,a]).
2. p([c,d,a]).
3. p([a]).

To generalize all of the predicates, we will find optimal match sequences for all the predicate pairs, 1 and 2, 1 and 3, 2 and 3. For clauses 1 and 2, SLGG of the ba and cda will be Xa. For 1 and 3, it will be Xa too. And for 2 and 3, SLGG of cda and a will be Xa again. Thus the result set for generated SLGGs will only have one member, Xa. This SLGG can be represented in Prolog as follows:

```
p(L) :- append(L1, [a], L).
```

Example 3.6: In this example, positive examples, which produce more than one SLGG, will be examined. Let us assume that the following clauses are given as positive examples.

1. p(ca).
2. p(dea).
3. p(b).
4. p(fgb).

The generalization of clauses 1 and 2 is Xa, 1 and 3 is X (since there is no similar part), 1 and 4 is X too, 2 and 3 is X, 2 and 4 is X and finally 3 and 4 is Xb. Thus, the result set is { Xa, X, Xb}. Since there is more than one solution, we should order them as in decision lists [17]. Scoring and sorting algorithm will be explained in Section 3.3. The results can be represented in Prolog as follows

```
p(L) :- append(L1, [a], L).
p(L) :- append(L1, [b], L).
p(L).
```

As it is seen from the result, first two predicates capture the fact that these predicates should end with a or b. The third clause is the over-generalized one and can be eliminated by the scoring algorithm.

A question may come to mind that "What happens if we generate SLGGs from these SLGGs?" This means that trying to generalize the learned templates. If you need more generalization in a specific domain this can be tried but generally it does not improve the performance much. Say, *Xabcd* and *XbYcd* are generated templates; generalization of these templates produces *XbYcd* again. Nothing has been learned from these templates. If *abXcd* and *efXab* are used then *XabY* template can be learned, which means that there is an *ab* structure that is independent from *ef* and *cd*. Thus, we can say that our algorithm does not work incrementally, since for the generation of the templates we need all the examples.

3.2.1 Generalization with n-arity predicates

Generalization with n-arity predicates is important for different domains, such as exemplar-based machine translation systems [27, 29]. Some EBMT systems use 2-arity predicates for learning translation rules. In this section, generalization process for n-arity predicates using single arity generalization will be looked through.

In the generalization of single-arity predicates, string pairs are used to find the optimal match sequences and the SLGGs of these strings. In n-arity predicate generalization, again string pairs are used, but these pairs are the first parameter of a predicate and the first parameter of another predicate and second parameters, third parameters, … and nth parameters. After the generation of the SLGGs, they are combined with respect to their scores. This process can be defined as follows

Let us assume that p1(s1, s2, …, sn) and p2(α1, α2, …, αn) are two predicates with the same arity. The alphabets of these arguments can be different and these alphabets may not be the known character based alphabets. Optimal match sequences for s1…sn and α1…αn is O1…On and their SLGG sets are S1…Sn. The cartesian product of these sets gives the generalized templates of these predicates.
There are some conditions that are satisfied because of the definition, these are:

- Number of elements of each SLGG set might be different from each other.
- Number of elements of SLGG sets are depends on the generated SLGGs from sx and αx.
- If n(S) gives the number of elements in S. Cartesian product of these sets produces a result set with n(S1)*n(S2)* … *n(Sn) elements.

Notice that result set might be very big and it may include nonsense or useless templates. Using the scoring and sorting algorithm can prevent this. Scoring reduces the elements of S1, S2, …, Sn which causes a decrease in the size of results set. Scoring will be explained in Section 3.3.

Definition might be a little bit blur, but a few examples will be enough to make the scene clear.

Example 3.7: In this example, we will see the basic process to find the generalized templates for 2-arity predicates. Let us assume that the following positive predicates are given

```
p(abc, dbe).
p(klc, dmv).
```

These clauses will be represented in Prolog as follows

```
p([a, b, c], [d, b, e]).
p([k, l, c], [d, m, v]).
```

First of all, we should find the optimal match sequences between abc and klc, and then OMS between dbe and dmv. Optimal match sequence of abc and klc is (ab,kl)c. OMS of dbe and dmv is d(be,mv). SLGGs of these match sequences are Xc and dX, respectively. Cartesian product of gives only one solution

```
p(Xc, dY).
```

Which means that, first parameter must end with c, and the second parameter must begin with d. It can be represented in Prolog as

```
p(List1, List2) :- append(L1, [c], List1),
                   append([d], L2, List2).
```

Example 3.8: This example shows the multi-result generation process with the following positive examples.

```
p(abc, dbe).
p(acb, ebd).
```

Optimal match sequences of abc and acb are a(ε, c)b(c, ε) and a(b, ε)c(ε, b).

Optimal match sequences of dbe and ebd are (ε, eb)d(be, ε), (d,e)b(e,d) and (db, ε)e(ε, bd).

SLGGs of abc and acb are aXbY and aXcY.

SLGGs of dbe and ebd are XdY, XbY, XeY.

Result set will include 2x3 = 6 elements; these are

```
p(aXbY, LdM).
p(aXbY, LbM).
p(aXbY, LeM).
p(aXcY, LdM).
p(aXcY, LbM).
p(aXcY, LeM).
```

Example 3.9: This example examines the conditions, which there are more than 2 positive examples. Following positive examples can be used for this example.

1. p(abc, dbe).
2. p(klc, dmv).
3. p(alc, dme).

The result set for 1 and 2 has been generated in Example 3.7. We need to generate SLGGs of 1 and 3, and 2 and 3.
Optimal match sequence of abc and alc is a(b, l)c.
Optimal match sequence of klc and alc is (k,a)lc.
Optimal match sequence of dbe and dme is d(b, m)e.
Optimal match sequence of dmv and dme is dm(v, e).
SLGGs are aXc, Xlc, dXe, dmX respectively.

We have generalized templates

p(Xc, dY) from 1 and 2.
p(aXc, dYe) from 1 and 3
p(Xlc, dmY) from 2 and 3.

as the result set. Notice that some of the generated templates are more specialized, while the others are more generalized. Ordering of these generated templates is another problem and will be explained in Section 3.3.

If the alphabets of the arguments are same, finding similar parts and giving the same variables to those part could be a good feature, but it does not supported in the current version of the program. This feature can be added to the program easily, since our current algorithm has already finds the similar parts of given to strings. If we give the two arguments of the example, we can find the similar parts easily. (This is true for only the predicates with two arguments).

3.3 Scoring and Sorting

Scoring the generated templates is one of the most important parts of this work. Although, generalization algorithm finds all the optimal match sequences and their SLGGs, it is not enough for practical usage of the result set. There should be an order between these result, which we can say which ones are more specialized and which ones are more generalized. Since order of applying rules is very important in many ILP systems [17, 21, 32], order of the rules should be declared by our algorithm too.

If we can define which result is the most specialized one for us then, it will be easy to find an algorithm for ordering the templates. Let us examine the following positive examples and their result set.

Positive examples are:

1. p(abc).
2. p(klc).
3. p(alc).

Generated templates for these examples are:

1. p(Xc).
2. p(aXc).
3. p(Xlc).

Notice that Xc covers all the examples, but aXc and Xlc covers 2/3 of the examples. From this point of view it can be said that Xc is the most general one and its score should be less than the others. For the present, let us assume that it is correct. Then, how will we decide about the order of aXc and Xlc? Although both of them cover the 2/3 of the examples, we can make a preference that more compact, less fragmented results are better and less specialized. Thus, our algorithm can be based on the coverage and compactness/fragmentation.

3.3.1 Fragmentation score for single-arity predicates

Fragmentation of a template means that the fragmentation of terminal symbols in a template. Say, a template occurs from (V)$T_1VT_2 \ldots T_n$(V). V stands for variables and T symbolizes the terminal groups. Number of fragments for this template is n, since there are n terminal groups. If n(T) gives the length of the terminal groups, then fragmentation score of a template is

$$FS = n(T_1)^2 + n(T_2)^2 + \ldots + n(T_n)^2$$

Example 3.10: This example shows the calculation of the fragmentation score for a simple template. Let us assume that generated templates are the following ones:

```
p(Xc).
p(aXc).
p(Xlc).
```

Fragmentation scores for these templates are

$FS(Xc) = 1^2 = 1$
$FS(aXc) = 1^2 + 1^2 = 2$
$FS(Xlc) = 2^2 = 4$

Using only the fragmentation score scheme, templates can be sorted with respect to their specificity. The prolog output will be as follows

```
p(List) :- append(L1, [l, c], List).
p(List) :- append([a], L1, L2), append(L2, [c], List).
p(List) :- append(L1, [c], List).
```

assuming, "append" as the background knowledge.

3.3.1.1 Fragmentation score for n-arity predicates

Fragmentation score for n-arity template is the sum of the fragmentation scores of individual arities. If fragmentation score of each arity is λ, total fragmentation score, θ, will be:

$$\theta = \lambda_1 + \lambda_2 + \ldots\lambda_n.$$

Example 3.11: This example shows the calculation of n-arity predicates in a detailed manner. Let us assume that we have given following positive examples.

1. p(abc, dbe).
2. p(klc, dmv).
3. p(alc, dme).

Generated templates for these examples are

p(Xc, dY) from 1 and 2.
p(aXc, dYe) from 1 and 3
p(Xlc, dmY) from 2 and 3.

Total fragmentation scores for these templates are

- Xc = 1, dY = 1 and $\theta = 1 + 1 = 2$
- aXc = 2, dYe = 2 and $\theta = 2 + 2 = 4$

- Xlc = 4, dmY = 4 and $\theta = 4 + 4 = 8$

As it is seen from the scores, generated templates should be sorted as

```
p(Xlc, dmY).
p(aXc, dYe).
p(Xc, dY).
```

Since the fragmentation score is a kind of indicator of the coverage of all the possible strings with the given alphabet, not the coverage of the example set, we may need to change the order of or remove some of the generated templates with respect to our example set and domain. Thus, fragmentation score is used with the coverage score for sorting and eliminating the generalized templates.

It can be noticed that, scoring algorithm omits some conditions. For example, aXc and XaYc are the generated templates. Scoring algorithm calculates the scores of aXc and XaYc as 2 for both of them, although XaYc covers the superset of aXc's coverage. If this kind of accuracy is needed then the number of variables can be used as a parameter for the calculation. Moreover, there are other methods that deal with gap creation and gap extension in sequence alignment [5, 9]. These methods can be adapted for this purpose.

3.3.2 Confidence factor/ Coverage score

Confidence factor assignment to the learned rules is very common in statistical machine learning algorithms [23]. By the help of the confidence factor, very rare or very specialized rules can be eliminated or vice versa. Both of them can be used in different domains. If generalized templates are more useful instead of the specialized ones, or if you want to cover all the examples with a few templates, then templates with small coverage score can be eliminated easily or vice versa.

Confidence factor of a template, δ, can be calculated as

$$\delta = \gamma/\eta$$

where γ is the number of covered examples, and η is the total number of examples. With single-arity predicates it can be calculated as following

1. p(abc).

2. p(klc).
3. p(alc).

are the positive examples and the generated templates are

```
p(Xc).
p(aXc).
p(Xlc).
```

The coverage scores of these templates are

- Xc covers 3/3 of the examples (abc, klc and alc) and δ is 1.
- aXc covers 2/3 of the examples (abc and alc) and δ is 0.66.
- Xlc covers 2/3 of the examples (klc and alc) and δ is 0.66.

3.3.2.1 Coverage score for n-arity predicates

For n-arity predicates calculation of coverage score is similar to the single-arity predicates. Definition is same with the singe-arity predicates, but finding coverage a little bit different. If the given positive examples are as following

1. p(abc, dbe).
2. p(klc, dmv).
3. p(alc, dme).

And the generated templates are

1. p(Xlc, dmY).
2. p(aXc, dYe).
3. p(Xc, dY).

For the first template, Xlc covers 2nd and 3rd examples; dmY covers 2nd and 3rd examples too, intersection set is 2nd and 3rd examples. So the coverage of p(Xlc, dmY) is 2/3 (0.66).
For the second one, aXc covers 1st and 3rd examples; dYe covers 1st and 3rd examples and the intersection set is 1st and 3rd examples. The coverage of p(aXc, dYe) is 2/3 (0.66).
For the last one, Xc covers all the examples, and dY covers all the examples too. So the coverage score for the p(Xc, dY) is 3/3 (1.0).

Example 3.12: In this example, parameters of generated templates covers in a synchronized manner, but this may not be come true for every example set. If a new example, p(plc, dce), is added to our predicates, we can observe the difference. Our predicates will be

1. p(abc, dbe).
2. p(klc, dmv).
3. p(alc, dme).
4. p(plc, dce).

And the generated templates are

```
p(Xlc, dmY).
p(Xlc, dYe).
p(Xlc, dY).
p(aXc, dYe).
p(Xc, dYe).
p(Xc, dY).
```

First parameters of the 1st, 2nd and 3rd templates are same but the second parameters are different. This will cause different coverage sets for these templates.

- Xlc covers 2nd, 3rd, 4th examples.
- dmX covers 2nd and 3rd examples.
- dXe covers 1st, 3rd, 4th examples.
- dX covers all the examples.

The intersection sets for these templates are

- For p(Xlc, dmY), 2nd and 3rd , the coverage is 2/4 (0.5).
- For p(Xlc, dYe), 3rd , the coverage is 1/4 (0.25).
- For p(Xlc, dY), 2nd, 3rd, 4th, the coverage is 3/4 (0.75).

As it can be seen from the example, first and second parameters could cover different examples. We should be careful about this fact during coverage score calculations.

Moreover, the last example shows an important point that, generated templates are in the fragmentation score order, but when we calculated their coverage scores, we saw that their order change with respect to their coverage scores. This shows that fragmentation score and the coverage score should be used in a combined manner. And the weights of these scores on the total score could be changed with a parameter.

3.3.3 Total Score

Total score calculation is needed because of different domains and different requirements of the applications. By the total score calculation we can give different weights to the fragmentation score and the coverage score. If the weights are equal then we want results with high fragmentation score and high coverage. In fact, adjusting the weights of the fragmentation and coverage could be a little bit painful.

Total score, Φ, is the sum of the weighted δ, coverage score, and θ, fragmentation score, by given weight factors. If fragmentation factor is α, and coverage factor is β, then total score is

$$\Phi = \alpha\theta + \beta\delta$$

Changing fragmentation and/or coverage factor affects the ordering of the generated templates. If the templates that have more coverage score are more exceptional, then we should increase the coverage factor or vice versa. Weight parameters can be defined in the input file as follows

```
parameter('align_factor', 0.15).
parameter('cover_factor', 0.50).
```

First predicate defines the weight of the fragmentation score, α, as 0.15, and the second predicate defines the weight of the coverage score, β, as 0.50. If we want to see all the scores for example 3.12:

	Fragmentation	Coverage	Total Score
p(Xlc, dmY).	16	0.50	2.650
p(Xlc, dYe).	8	0.50	1.450
p(Xlc, dY).	4	0.75	0.975
p(aXc, dYe).	4	0.50	0.850
p(Xc, dYe).	2	0.75	0.675
p(Xc, dY).	1	1.00	0.650

Table 3.1: Calculated scores for example 3.12

3.3.4 Cut-point level

Cut point level is an important facility that may speed up the whole process. In this work, cut point is used for only selection of first n high scored solution, but this might be broadened to different types of cut-point applications. Some of them are:

- Detecting score gaps between consecutive templates to find the cut point
- Taking the average or mean of the scores and getting the templates, which are around the mean or average.
- Use different scores for the selection, such as fragmentation, coverage, etc.

This list can grow easily by appending statistical methods. Selecting first n top scored template is enough for our work. We can define the cut-point level in the input file as follows

```
parameter (’constraint_level’,5).
```

This predicate says that get the first five high scored templates for the final template set. Usage of the cut point can be understood with an example easily.

Example 3.13: In order to show the usage of the cut point, pairs of the examples should produce more than one optimal match sequence. Since cut point is applied to generated templates of two strings. Assume that following positive examples are given

```
p(aabcc).
p(abc).
```

Optimal match sequences of these strings are

- a(a,)bc(c,)
- a(a,)b(,c)c
- (a,)abc(c,)
- (a,)ab(,c)c

And the SLGGs of these match sequences are

- aXbcY with fragmentation score of 5.
- aXbYc with fragmentation score of 3.
- XabcY with fragmentation score of 9.

- XabYc with fragmentation score of 5.

If the cut-point is defined as 1, we get the most compact solution XabcY, although other solutions are might show meaningful generalizations, such as aXbYc, which means every string will begin with *a*, end with *c*, and it must include a *b* in the middle somewhere. If the cut-point is 2, we will get XabcY and then there are two solutions aXbcY and XabYc. Which one should we get? Or should we get both of them? In this work, we preferred to get the one that we meet first, since there might be many more solutions with the same score. Getting all the solutions with the same score might be a little overwhelming for processing the final result set. This condition can be examined by adding a new positive example to our input set.

Example 3.14: In this example, a new predicate will be added to the input set and the effects of the cut-point on the final set will be looked through. If our domain is the strings which include *abc*. Our examples will be

1. p(aabcc).
2. p(abc).
3. p(cabca).

Optimal match sequences of 1 and 2 are already calculated in the previous example. For 1-3 and 2-3, optimal match sequences and their SLGGs with their score are

From 1-3:

(ε, c)a(a, ε)bc(c, a)	➔ XaYbcZ with score of 5.
(a, c)abc(c, a)	➔ XabcY with score of 9.
(a, c)ab(c, ε)c(ε, a)	➔ XabYcZ with score of 5.
(ε, c)a(a, ε)b(c, ε)c(ε, a)	➔ XaYbZcM with score of 3.
(ε, cabc)a(abcc, ε)	➔ XaY with score of 1.
(ε, c)a(ε, bc)a(bcc, ε)	➔ XaYaZ with score of 2.
(aab, ε)c(ε, ab)c(ε, a)	➔ XcYcZ with score of 2.

From 2-3:

(ε, c)abc(ε, a)	➔ XabcY with score of 9.
(ab, ε)c(ε, abca)	➔ XcY with score of 1.
(ε, cabc)a(bc, ε)	➔ XaY with score of 1.

Notice that there are many useless generated templates; by the help of the cut-point, we get XabcY from 1-2, XabcY again from 1-3 and XabcY from 2-3 too. Thus, final result set for these inputs will include only XabcY. This is the perfect solution that we want. On the other hand, this approach may prevent the occurrence of the interesting but less compact templates. Although cut-point mechanism is used to reduce the generated output, with big datasets this might not be an enough solution. Selecting the useful subset(s) within these templates is another problem and it will be handled by the selection sets.

3.4 Selection Sets

Selection sets are used to examine the practicality/usability of used scoring schemes. General algorithm during the calculation of these sets is

1. Order the templates by its fragmentation/coverage/total score.
2. Select templates one by one beginning from the most specific.
3. Omit the ones with coverage score 1.00.
4. Check that selected template covers new/uncovered examples.
5. If all the examples are covered, stop to select templates.
6. Remove redundant templates that are covered by a more general template in the selection set.

Differentiating from this whole coverage selection set only includes the ones with the coverage score of 1.00.

There are four kinds of selection sets used in this work. These four different approaches are

- By fragmentation score
- By coverage score
- By total score
- By whole coverage

In order to see the differences between these methods we need a positive example set that we can use in four selection algorithm to see the difference. Let us assume the following past tenses of some verbs have been given as the positive examples.

```
pr(moved).
pr(removed).
pr(killed).
pr(spied).
pr(fried).
pr(married).
pr(written).
pr(engineered).
pr(stopped).
pr(connected).
pr(clipped).
```

If the fragmentation score weight and coverage score weight are 0.5. The generated templates and their scores will be as in Table 3.2.

	Fragmentation	**Coverage**	**Total**
Xmoved	25	0.181818	3.840909
Xried	16	0.181818	2.490909
Xpped	16	0.181818	2.490909
XrYied	10	0.181818	1.590909
Xied	9	0.272727	1.486364
XnYneZed	9	0.181818	1.440909
XnYneZeKd	7	0.181818	1.140909
XnYnZeKed	7	0.181818	1.140909
sXpYed	6	0.181818	0.990909
XnYeZed	6	0.181818	0.990909
mXed	5	0.181818	0.840909
XoYed	5	0.363636	0.931818
XrYed	5	0.363636	0.931818
XmYed	5	0.272727	0.886364
XeYed	5	0.272727	0.886364
XreYd	5	0.181818	0.840909
XiYed	5	0.545455	1.022727
XlYed	5	0.181818	0.840909
XpYed	5	0.272727	0.886364
XriYeZ	5	0.272727	0.886364
XtYed	5	0.181818	0.840909
cXed	5	0.181818	0.840909
XcYed	5	0.181818	0.840909
Xed	4	0.909091	1.054546
XenY	4	0.181818	0.690909
XteY	4	0.181818	0.690909
XnYnZeKd	4	0.181818	0.690909
XnYeZeKd	4	0.181818	0.690909
XoYeZd	3	0.363636	0.631818

XrYeZd	3	0.363636	0.631818
XeYeZd	3	0.272727	0.586364
XiYeZd	3	0.545455	0.722727
XrYiZeK	3	0.272727	0.586364
XnYeZd	3	0.181818	0.540909
cXeYd	3	0.181818	0.540909
XeYd	2	0.909091	0.754545
XrYeZ	2	0.454545	0.527273
XiYeZ	2	0.636364	0.618182
XiYnZ	2	0.181818	0.390909
XeYnZ	2	0.181818	0.390909
XtYeZ	2	0.272727	0.436364
XeY	1	1.00	0.65
XnY	1	0.272727	0.286364

Table 3.2: Generated templates for some past tense examples

There are 43 generated templates in the result set and many of them are uninteresting and useless. From these results, we get any information about the examples, but decreasing the number of templates and increasing the percentage of usefulness would be better. To achieve this goal 4 ways have been tried. Now, we can examine these four different approaches with the same data.

3.4.1 Selection with fragmentation score

Generated selection set without removing the redundant templates with respect to fragmentation score will be as follows

```
p(Xmoved).
p(Xried).
p(Xpped).
p(Xied).
p(XnYneZed).
p(XiYed).
p(XriYeZ).
```

Reader might notice that, these generated templates are not the first seven templates in Table 3.2. This is because we do not select the templates that do not cover any new/uncovered example, which is declared as the 4th step of the algorithm. If we step over the algorithm:

Xmoved is selected that covers *moved* and *removed*.

Xried covers *fried* and *married*.

Xpped covers *stopped* and *clipped*.

XrYied covers *fried* and *married* already covered by Xried.(omitted)

Xied covers *spied*, *fried* and *married*.

XnYneZed covers *connected* and *engineered*.

XnYneZeKd does not covered any new example, not selected.

XnYnZeKed does not covered any new example, not selected.

…

Until XiYed, templates cover the examples already covered by previous templates. In other words, templates between XnYneZed and XiYed do not cover any new example, so we do not include them in our selection set, but XiYed covers *killed*, which is not covered before.

And the algorithm goes on like this, until all the examples are covered. Since only *written* has left as uncovered, when we meet with XriYeZ which covers *written*, algorithm stops. In the end we have a compact and very informative result set with 8 elements, instead of 43. Moreover, it covers all the given examples as the other one. But a careful one, may notice that Xied covers the superset of Xried and XiYed covers the superset of Xied. Then why do we use Xried and Xied? In fact, Xried and Xied might be needed in different domains and some applications, our algorithm provides the redundant template removal for the ones who need more compact results. On the other hand, going towards more compact result, means that loosing information about the examples. Thus, the requirements of the domain should be defined well about the needed information.

Final result set with removal of the redundant templates would be

```
p(Xmoved).
p(Xpped).
p(XnYneZed).
p(XiYed).
p(XriYeZ).
```

with 5 templates.

3.4.2 Selection with coverage score

To be able to see the execution of the algorithm easily, we need the generated templates sorted by coverage score in descending order.

	Fragmentation	Coverage	Total
XeY	1	1	0.65
Xed	4	0.909091	1.054546
XeYd	2	0.909091	0.754545
XiYeZ	2	0.636364	0.618182
XiYed	5	0.545455	1.022727
XiYeZd	3	0.545455	0.722727
XrYeZ	2	0.454545	0.527273
XoYed	5	0.363636	0.931818
XrYed	5	0.363636	0.931818
XoYeZd	3	0.363636	0.631818
XrYeZd	3	0.363636	0.631818
Xied	9	0.272727	1.486364
XmYed	5	0.272727	0.886364
XeYed	5	0.272727	0.886364
XpYed	5	0.272727	0.886364
XriYeZ	5	0.272727	0.886364
XeYeZd	3	0.272727	0.586364
XrYiZeK	3	0.272727	0.586364
XtYeZ	2	0.272727	0.436364
XnY	1	0.272727	0.286364
Xmoved	25	0.181818	3.840909
Xried	16	0.181818	2.490909
Xpped	16	0.181818	2.490909
XrYied	10	0.181818	1.590909
XnYneZed	9	0.181818	1.440909
XnYneZeKd	7	0.181818	1.140909
XnYnZeKed	7	0.181818	1.140909
sXpYed	6	0.181818	0.990909
XnYeZed	6	0.181818	0.990909
mXed	5	0.181818	0.840909
XreYd	5	0.181818	0.840909
XlYed	5	0.181818	0.840909
XtYed	5	0.181818	0.840909
cXed	5	0.181818	0.840909
XcYed	5	0.181818	0.840909
XenY	4	0.181818	0.690909
XteY	4	0.181818	0.690909
XnYnZeKd	4	0.181818	0.690909
XnYeZeKd	4	0.181818	0.690909
XnYeZd	3	0.181818	0.540909
cXeYd	3	0.181818	0.540909

XiYnZ	2	0.181818	0.390909
XeYnZ	2	0.181818	0.390909

Table 3.3: Generated templates sorted by coverage score

Table 3.3 says that scoring of the templates with fragmentation and coverage are in the reverse direction generally as it is expected. So the one that covers all the examples is at the top. Fortunately, our selection algorithm omits the ones that cover all the examples, since they will block the selection of specialized templates. Thus, the generated selection set will be

```
p(Xed).
p(XiYeZ).
```

Xed covers all the regular verbs and XiYeZ covers the *written*. As it is seen this selection set shows only the most common properties of the examples. With this approach, it might not be very useful, but the algorithm for selection could be changed that templates, which have coverage score above average or some cut point (like 0.20 for this example) or some gap, can be selected and from this group, there could be another selection.

3.4.3 Selection with total score

Selection with total score might be the most promising selection set generation approach. In order to see this, we need the sorted solution by total score.

	Fragmentation	**Coverage**	**Total**
Xmoved	25	0.181818	3.840909
Xried	16	0.181818	2.490909
Xpped	16	0.181818	2.490909
XrYied	10	0.181818	1.590909
Xied	9	0.272727	1.486364
XnYneZed	9	0.181818	1.440909
XnYneZeKd	7	0.181818	1.140909
XnYnZeKed	7	0.181818	1.140909
Xed	4	0.909091	1.054546
XiYed	5	0.545455	1.022727
sXpYed	6	0.181818	0.990909
XnYeZed	6	0.181818	0.990909
XoYed	5	0.363636	0.931818
XrYed	5	0.363636	0.931818

XmYed	5	0.272727	0.886364
XeYed	5	0.272727	0.886364
XpYed	5	0.272727	0.886364
XriYeZ	5	0.272727	0.886364
mXed	5	0.181818	0.840909
XreYd	5	0.181818	0.840909
XlYed	5	0.181818	0.840909
XtYed	5	0.181818	0.840909
cXed	5	0.181818	0.840909
XcYed	5	0.181818	0.840909
XeYd	2	0.909091	0.754545
XiYeZd	3	0.545455	0.722727
XenY	4	0.181818	0.690909
XteY	4	0.181818	0.690909
XnYnZeKd	4	0.181818	0.690909
XnYeZeKd	4	0.181818	0.690909
XeY	1	1	0.65
XoYeZd	3	0.363636	0.631818
XrYeZd	3	0.363636	0.631818
XiYeZ	2	0.636364	0.618182
XeYeZd	3	0.272727	0.586364
XrYiZeK	3	0.272727	0.586364
XnYeZd	3	0.181818	0.540909
cXeYd	3	0.181818	0.540909
XrYeZ	2	0.454545	0.527273
XtYeZ	2	0.272727	0.436364
XiYnZ	2	0.181818	0.390909
XeYnZ	2	0.181818	0.390909
XnY	1	0.272727	0.286364

Table 3.4: Generated templates sorted by total score

If Table 3.2 and Table 3.4 is compared, it is seen that Xed takes its position between the templates with high fragmentation score. The reflection of this change can be observed in the generated selection set.

```
p(Xmoved).
p(Xried).
p(Xpped).
p(Xied).
p(XnYneZed).
p(Xed).
p(XriYeZ).
```

Again we have 7 templates, but this time it includes Xed, instead of XiYed. Destiny of the XiYed is directed by a small number (1.054546-1.022727 = 0.031819). Notice that weights of the fragmentation and coverage score can change the order of the templates, which may cause the changing of the selection set. In addition to this, the order of the Xed and XriYeZ is in a conflict with sorting from most specialized to most generalized. In fact, this is our choice in this selection model, but this behavior may need to be questioned in different domains.

Moreover, if we want to use the removal of redundant templates, the output will be

```
p(Xed).
p(XriYeZ).
```

This output is the same with the coverage score selection set. It covers all the regular verbs and the *written*.

3.4.4 Selection set with coverage score 1.0

In this example, there is only one template that covers all the examples. So the result set is

```
p(XeY).
```

If our input had included a past tense of a verb, which does not include any *e* character, then the selection set would have been empty, since the only possible template that covers all the examples would be p(X).

Although this selection set might seem useless, there might be some domains with long sequences that seeing common points is difficult. By this selection set, these kind of common points that are not to be noticed, might be discovered easily.

Chapter 4

Implementation

Our system consists of three fundamental parts:

- Alignment of two sequences
- Assigning score to individual templates
- Constructing decision list

Alignment module generates all possible templates for two given strings obeying the constraints about maximum template number. During this generation process, scoring module assigns score to these match sequences/templates. After generation of all possible templates for all sequence pairs, decision list construction module sorts and eliminates some of these templates with respect to given selection criteria and finally produces the selection set/decision list. General architecture is given in Figure 4.1. The components will be explained in details in the following sections.

4.1 Alignment Module

The aligner component of the alignment module takes only two sequences/strings for the alignment process. All possible alignments of these two sequences are generated, but only the ones that obey the constraint level parameter are stored in the buffer. Manager part of the alignment module arranges the template generation. Since only two sequences can be given at the same time to the aligner, for n-arity predicates aligner is fed in the argument order, and the Cartesian product of the generated templates is taken to get the final templates. Again, constraint level checker eliminates the excessively generated templates. After generation of templates for two sequences, alignment manager feeds the aligner with another combination of given examples. It goes until all the combinations of examples are fed into the aligner. The most important component of the alignment module is the aligner, which finds the optimal match sequences of given two strings. The algorithm is given in the following lines.

4.1.1 Algorithm to find optimal match sequences

In this part, a simple recursive algorithm will be given to find optimal match sequences. Since the main goal of this project is not finding the optimal match sequences in a faster way, an easy implementable, recursive algorithm was chosen to implement. A simplified pseudo-code has been given to give an idea about the recursive solution.

In fact, alignment operation is a depth-first search. Recursive algorithm finds all possible alignments with depth-first search and generated match sequences are put in a buffer, and then they are converted to templates for the generalization process. Generalization process takes generated match sequences and changes difference parts of the match sequences with variables.

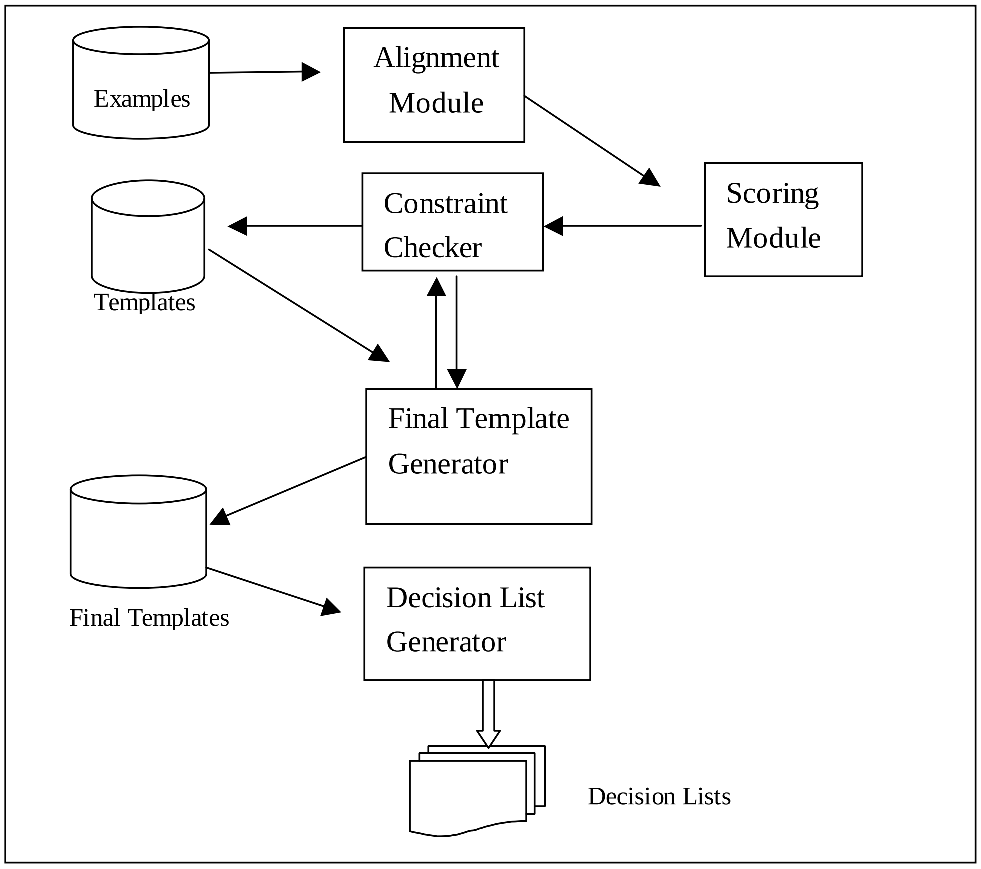

Figure 4.1: General architecture

```
FindAlignment(String1, String2, oms, Dif1,  Dif2)

  If String1 or String2 is empty then
    concat similarity and difference parts with already generated oms
    call scoring module with generated oms to put it to db
    return
  do {
    if String1 does not include String2’s first character or
        their first characters are same then
    {
       append String2’s first character to Dif2
       FindAlignment(String1, String2.substring(1,last), oms, Dif1,  Dif2)
    }
    else if it includes first character in different position
    {
       append new diffs to oms and empty Dif1 and Dif2
       FindAlignment(String1.substring(position), String2.substring(1, last), oms,
                      Dif1,  Dif2)

       If there are suitable parts in String1 for matching with the rest of String2
        then
          FindAlignment(String1, String2.substring(1, last), oms, “”,  Dif2)

    }
  }until String1  does not include String2’s first character
```

Figure 4.2: Alignment algorithm

Figure 4.2 shows the recursive alignment algorithm. This function is called by the alignment manager such as:

```
FindAlignment(“abcd”, “ad”, “”, “”, “”);
```

First two arguments are strings which will be aligned. Third parameter is the already generated oms of these strings and the last two arguments are the already found differences that are not finished. In the first call, since the string1 and string2 are not empty, do-until loop is executed. Since the first characters are same “a”. FindAlignment is called as

```
FindAlignment(“bcd”, “d”, “a”, “”, “”);
```

“b” is not found in string2 so “b” is added to the Dif1 and

```
FindAlignment(“cd”, “d”, “a”, “b”, “”);
```

is called. Again string2 does not include “c”, so “c” is added to the Dif1 and

```
    FindAlignment(“d”, “d”, “a”, “bc”, “”);
```

is called. The called function finds “d” as similar part and closes the differences by appending difference part to the oms. Oms becomes “a(bc, E)”. Then similar part is appended to the oms to call

```
    FindAlignment(“”, “”, “a(bc,E)d”, “”, “”);
```

Since the string1 and string2 are empty, we understood that oms is the final product. Thus, this oms can be send to the scoring module. When the function returns from the scoring module, all the called functions returns, since there are no more producible oms. The alignment manager calls the findAlignment with other strings.

If our strings are “abcda” and “ad”, our recursion will be rolled back until the first called function. This function finds the second occurrence of “a” and calls itself as follows

```
    FindAlignment(“”, “d”, “a”, “abcd”, “”);
```

Since the first string is empty, function realize that it is the stop point and it concatenates the rest of the remaining string to produce the oms. The resultant oms is “(abcd,E)a(E,d)”. Again this oms is sent to the scoring function and when it returns, recursive functions returns to the caller function.

The data structures and the class definitions can be found in Appendix A.

4.2 Assigning Score to Templates

Scoring module has two parts. One of them calculates the fragmentation score of the generated templates just after the generation of the template. Other part calculates the coverage score. Calculation of fragmentation score is fast enough since it is calculated from the generated template. On the other hand, calculation of the coverage score depends on the size of the examples. It checks for every example that if it is covered by the generated template. Using a kind of caching mechanism might accelerate this process. Nevertheless, this has not been implemented in this project.

Fragmentation score calculation part takes the generated template as input such as:

abcXdef

and gives the fragmentation score. Fragmentation score is calculated similar to the Euclidian distance calculation without the square root. Each similar part (terminals between variables) taken as a different dimension and the sum of the squares of their lengths gives the fragmentation score. For n-arity predicates, total fragmentation score is the sum of the fragmentation scores of each argument. The details of the calculation process can be found in Chapter 3. At the end, the template and its fragmentation score information are send to the constraint checker.

Calculation of coverage score is done during the generation of final templates. Since the individual coverage of templates of n-arity predicates does not mean anything, their coverage score is calculated for all the argument templates. Thus for the calculation of the coverage score, we need to wait until the final templates begin to be generated. Calculation details of coverage score are in Chapter 3.

4.2.1 Constraint checker

The constraint checker is the guardian of the databases (arrays in the current implementation). Every template structure is checked before being added to the DB. Constraint checker gets template structure as input and if it is an acceptable template, adds it to the DB.

In fact, there are two different constraint checkers, one of them checks the templates for single-arity predicates and the temporary templates for multi-arity predicates. “constaint_level” parameter is important for this checker. The other one, checks the constraints of multi-arity predicates for the final template DB. For this checker “unbalanced_variable” and “constraint_level” parameters are the most important ones.

4.3 Decision List Construction

Selection set generation and producing decision lists is the final part of the whole process. There are four kinds of selection sets used in this work. These four different approaches are

- By fragmentation score
- By coverage score
- By total score
- By whole coverage

Details of these approaches can be found in Chapter 3, but in this section information about the relations between the selection set generation module and the other modules can be found.

Selection set generation module uses the final templates generated by alignment and scoring modules. All the final templates have fragmentation score, coverage score and total score. This module selects from these templates with respect to four different constraints and put them in different sets. These sets are sorted with respect to their related score. At this point decision list generation part produces decision lists for these sorted selection sets. Figure 5.2 explains this process.

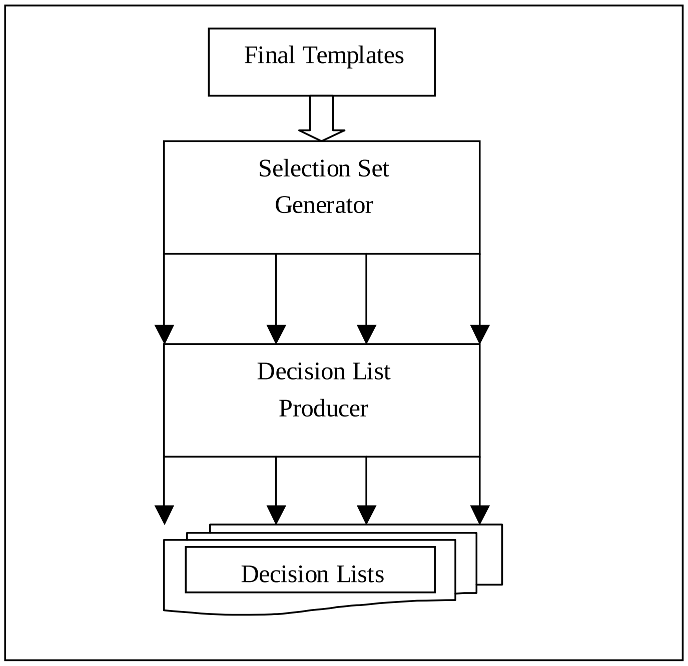

Figure 4.3: Decision list generation

4.4 Working of The Program

In this section, we will begin with the example set and generate one of the decision lists. Let us assume the following example predicates are given:

```
        p(abcd).
```

```
p(ad).
p(aed).
```

Alignment manager takes two of these predicates (firstly 1 and 2, then 2 and 3) for alignment and calls the aligner such as:

```
align(“abcd”, “ad”)
```

After this call, the given algorithm in Figure 4.2 begins to work. It finds “aXd” and sends it to the scoring module. Scoring module calculates fragmentation score as “2” and sends it to the constraint checker, constraint checker puts it to the template array in the first position. After this point, since there are no more templates which can be generated from these two strings, alignment manager sends these templates to the final template generator. Final template generator appends the coverage and total score to every template structure. In this example, it is “1” and “1.5” respectively for “aXd” (since the default ratio for fragmentation and coverage scores are 0.5). And, the templates are added to the final template array in the control of the constraint checker. Alignment manager calls aligner with

```
align(“ad”, “aed”)
```

Aligner finds “aXd” again and the same process is repeated, but in the addition to the final template phase, it is found that this template has already been added to the array and it does not added again. Thus in the end we have a final template array with one record, which is {“aXd”, 2, 1, 1.5}. In fact, there are more values such as variable count, fragment count in the real structure but they are used internally.

Since all the final templates and their required scores are generated, we selection set generation can be called. In the current version of the program, the default selection set is the one which uses the fragmentation score for sorting. Final templates are sorted by their fragmentation score or the one defined in input file. In addition, the first “N” of the templates are selected and the rest are cropped. Decision list generator generates the following lines:

```
p(X):-
append([‘a’],L1,L2),
append(L2,[‘d’],X),!.
```

This is all the work done to find the generalization of given examples.

4.5 Time Complexity Analysis

In this section, time complexities of all the fundamental procedure, i.e. alignment, scoring, selection set generation will be derived. Assume that we have N examples, total number of alignment operations is denoted by TA.

$$TA = \binom{N}{2} = \frac{N(N-1)}{2} = \frac{N^2 - N}{2} = O(N^2)$$

Calculation of the coverage score depends on the number of examples and the number of generated templates. Calculation of coverage score for only one template is O(N), since there are N comparisons. Total number of comparisons for NT (number of templates) will be $N \times NT$. Thus, complexity of scoring will be $O(N \times NT)$.

In order to generate selection sets, all the generated templates are compared with the examples in the worst case. So the complexity of selection generation will be $O(N \times NT)$.

All calculated time complexities given above depend on the number of examples. Maximum complexity is the $O(N \times NT)$, but we have not calculated anything about the generation of templates from two string pairs that directly affects the NT. Assume that we have two strings with lengths *m* and *n*, such that $m \geq n$. Difference between *m* and *n* is defined as *d (d=m - n)*. In the worst case, total number of comparisons between two strings is

$$2m\left(\frac{n(n+1)}{2} - \frac{d(d+1)}{2}\right) = O(m \times n^2)$$

Maximum number of generatable template from two strings is *n*. In the worst case, we can assume that all the templates, generated from the examples, are different. Thus, the total number of the length of the strings which are half of the example strings that have minimum length gives the number of templates. Assume that all the strings have length *l*. Total number of templates will be

$$NT = l \times \frac{N(N-1)}{2}$$

As a result, the complexity of scoring and selection set generation will be $O(l \times N^3)$.

Table 4.1 shows timings and number of generated templates for past tense learning. “FindAlignment calls” column of the table is about the total number of recursive calls during the whole process. Generated templates are the total number of templates which are generatable/generated from all the examples. “Time without templates” is timing for alignment process without generating templates, since the templates are not generated, scoring/sorting and selection set generation do not affect the results.

Number of examples	**FindAlignment calls**	**Generated templates**	**Time without templates (ms)**	**Total Time (ms)**
10	2868	426	440	770
20	10095	1769	1810	4340
30	23964	4145	4230	15100
40	39369	7036	7250	29440
50	66468	11255	11760	54480
60	96962	15938	16860	85080
70	133291	21962	22680	126220
80	182076	30307	30480	186800
90	224454	37379	37570	249970

Table 4.1: Statistics for past tense learning

Chapter 5

Applications

String generalization can be used in many different domains. Some of them are DNA sequence alignment, past tense learning, translation template learning, etc. In this chapter, application of TDL[*] to these domains will be examined in detail.

Following sections are about the domains with single arity predicates, which includes learning member predicate, etc. Then domains with n-arity (especially 2-arity) will be examined.

5.1 Applications with Single-Arity

Although there are many applications with single-arity we chose the biological sequence alignment domain to explain the weak and strong sides of our program.

5.1.1 DNA sequence alignment

The main goal of the alignment is establishing homology in nucleotide positions. There are four types of sequences. These sequences are amino acid, protein-coding DNA, ribosomal DNA, non-coding DNA.

Sequence alignment has problems with amino acids and protein-coding DNAs, if they are less conserved, they can get insertions and deletions of nucleotides. Moreover, for non protein-coding DNAs, greater occurrence of insertions and deletions may be observed, since their sequences are not constrained by a translation.

[*] In this chapter, our application will be called as TDL, stands for "Template Decision List".

There are different alignment programs: e.g., CLUSTAL W, Divide and Conquer, Malign, Pileup, TreeAlign [3]. All programs use a set of parameters to calculate alignment; some can be set by the user. Common to all are: mismatch penalty, match or identity score, gap creation penalty, gap extension penalty. Individual programs have additional unique parameters. Matches increase overall score; mismatches, gaps decrease it.

Two protein sequences in FASTA format are as follows:

```
>oryza
MTKAIPKIGS---RRKVRIGLRRNARFSLRKSARRITKGVIHVQASFNNT
IITVTDPQGRVVFWSSAGTCGFKSSRKASPYAGQRTAVDAIRTV---GLQ
RAEVMVKGAGSGRDAALRAIAKSGVRLSCIRDVTPMPHNGCRPPKKRRL
>nicotiana
MAKAIPKISS---RRNGRIGSR--------KGARRIPKGVIHVQASFNNT
IVTVTDVRGRVVSWSSAGTSGFKGTRRGTPFAAQTAAANAIRTVVDQGMQ
RAEVMIKGPGLGRDAALRAIRRSGILLTFVRDVTPMPHNGCRPPKKRRV
```

The alignment of these two sequences found by our program is:

```
mXkaipkiXs---rrXrigXrXkXarriXkgvihvqasfnnt
iXtvtdXgrvvXwssagtXgfkXrXpXaXqXaXairtvXgXq
raevmXkgXgXgrdaalraiXsgXlXrdvtpmphngcrppkkrrX
```

All the variables shown as **X**, represent different sequences. Current version of our program finds this template in more than an hour. This is because current algorithm to find the alignments is a recursive one and the depth of the recursion increases with the length of the examples. Thus, without changing the algorithm of the aligner module of the program, it is not practical to use it for long sequences.

From the point of correctness, our algorithm finds the most specific generalized template. This is the best optimal solution with respect to our heuristic which says that the optimal solution is the one which has minimum number of variables and maximum fragmentation score, sum of the squares of the lengths of the similar parts. DNA/Protein sequence alignment algorithms have some parameters such as gap creation penalty and gap cost. Gap creation penalty affects the number of variables, in other words, the difference sequences in our algorithm. Gap cost is not taken into consider by our algorithm, since the length of the difference part is not important with respect to our algorithm. Finding the global optimal or finding the maximum similar points between two sequences is important, so restrictions on the length of the differences affects to solution. Moreover, some of the DNA/Protein sequence alignment programs employs parameters about DNA/Protein structures, such as the protein boundaries, replication begin-end points, etc. In fact most of the parameters employed are used to speed up the alignment process, since finding the best solution is NP-HARD as stated in Section 2.1, all the algorithms try to minimize the search space with these constraints. Our algorithm is very slow, but it finds the best optimal solution. Moreover, since our algorithm finds all possible alignments, our solution set covers the solutions of other programs. As a result, we can say that there is a trade-off between finding the global optimal solution and the time.

5.2 Experiments with 2-Arity

5.2.1 Past tense learning

Learning the past tense of English verbs is an important topic in machine learning since 1986. There is a lot of work done on this topic. Rumelhart and McClelland began with a classic perceptron algorithm, Ling and Marinov continued with slightly modified version of C4.5. Califf, tried to apply FOIL and GOLEM, but he got disappointing results [41]. Mooney achieved better results with FOIDL.

Some of these applications worked with phonetic encoding of verbs, some of them used alphabetic data. We will use the alphabetic dataset in our examples [42].

All the previous methods try to find the similar and changing parts between the present participle and past participle of the words. On the other hand, our method finds similar and changing parts between present participles, and then between past participles of verbs. Moreover, other methods use some kind of negative example, explicitly or implicitly and some of them use closed-world assumption. Using assumptions about the negative examples might be problematic. In real world, a new language learner cannot say anything negative about the past participles of words. He/she cannot say that this verb cannot be regular/irregular, until she/he learns its correct form.

From this point of view, TDL tries to learn every positive example like a human being without teacher. TDL compares each verb with each other, their past participles with each other and tries to generate templates about what it learns. Moreover, it uses a simple heuristic to define their order.

Example 5.1: In this example, learning past tenses of regular verbs will be examined. A few examples will be enough to learn “add ‘ed’” rule.

```
past(look, looked).
past(accept, accepted).
```

From two example, our method generates two templates:

```
past(X, Yed).
past(X, YeZd).
```

First rule is enough to show the learning of “add ‘ed’” rule with TDL. Second rule shows an important point in our work. There is one variable “X” in left side, but there are two variables “Y” and “Z” in right side. Note that our algorithm does not ensure that same variables correspond to the same meaning. Even for *past(X, Yed)*, X and Y does not represent the same thing, because X and Y might be in different alphabets. This condition might occur in translation examples. This will be examined in the following sections.

For this kind of domains, we use a parameter to provide the balance of variables in all the arguments of the rules:

parameter(‘unbalancedvariable’, false).

If we continue with the generated templates, fragmentation score selection and total score selection will produce only one template, which is:

```
p(X, Yed).
```

In FOILD, for the examples above, following line is generated [17]:

```
past(A,B) :- split(B, A, [e,d]).
```

Example 5.2: This example is about “exceptions to the exception to the rule” concept declared in [17] by Mooney and Califf. The example of this concept is about learning the changing of “y” to “ied” and incorrectly covering a few examples that are correctly covered by the previously learned “add ‘ed’” rule (e.g., bay → bayed; delay → delayed). FOIDL can overcome this problem. What about the TDL? Let us our examples are:

```
past(bay,bayed).
past(delay,delayed).
past(try,tried).
past(fry,fried).
```

There are seven generated templates for these examples. Note that some of these templates affected from the problem explained in the previous example.

```
past(Xay,Yayed).
past(Xry,Yried).
past(Xay,YeZd).
past(Xay,YdZ).
past(Xy,Yed).
past(Xy,YeZd).
past(Xy,YdZ).
```

Although, there are seven generated templates, selections sets for fragmentation and total score produce same output with two clauses:

```
past(Xay,Yayed).
past(Xry,Yried).
```

Since TDL, tries to find and use the most specific templates, we do not meet this kind of incorrect coverage. Reader may notice that result set does not have

```
past(Xy,Yied).
```

template, which is the source of the problem in Mooney's approach. Since TDL uses the decision lists as in FOIDL, the list that is ordered with our heuristic will be:

```
past(Xay,Yayed).
past(Xry,Yried).
past(Xy,Yied).
```

Again, TDL covers the examples correctly.

Example 5.3: Is everything excellent with TDL? In this example, the condition that TDL cannot learn anything will be shown. Let the examples are do not have any common character.

```
past(act, acted).
past(know, known).
```

Generated template is the most general template that accepts everything:

```
    past(X,Y).
```

Although, this is very disappointing, in the real dataset there are many words and these words have common parts with each other. If this was not the case, nobody would learn to speak or everyone would have a super memory. This problem can be recovered easily, if there are examples that cannot be covered by the generated templates except the most general template. These examples can be added directly to the result set, but in real life, main problem is these kind of exceptional verbs have similar characters with other words. Thus, our program may generate a template that can cover these examples (for know and known in this case) such as:

```
    p(XnY, MnoN).
```

Although, this is not a useful template from our point of view, it is an acceptable template for the program. Because of this kind of exceptions, current version of the program cannot learn wanted templates (in the meaning of past tense learning). In order to eliminate this, the pattern of the template to generate can be given. For the past tense learning, the templates that begin with a variable followed by terminals will be most suitable one. This causes omitting the information about the similar parts in the beginning and the middle of the strings, since only the end parts of the verbs are affected from this change. If in English there are verbs with past tenses that the first characters of the two tenses are different, this solution cannot be used too.

5.2.2 Learning translation templates

In 1996, Guvenir and Cicekli proposed a new exemplar based machine translation system [21]. This system is based on the translation templates. The heuristic of the translation template learning (TTL) algorithm can be summarized as follows: given to translation pairs, if there are some similarities (differences) in the source language, then the corresponding sentences in the target language must have similar (different) parts, and they must be translations of the similar (different) parts of the sentences in the source language. Similar parts are replaced with variables to get a template, which is a generalized exemplar, by this method. In [21], translation templates are ordered according to the number of terminal symbols of the templates. Since this criterion is not sufficient for large systems, [23] added confidence factor assignment that each rule and some rule combinations are assigned weights.

In this section, notation of [21, 22] will be used for the source and target languages. Let us try to use TDL to learn translation templates.

Example 5.4: This example is about the main motivation point of this work. In [30], it is stated that TTL algorithm cannot learn any template from the following translation examples between American and British English:

```
1. The other day, the president analyzed the state of the union ↔
      The other day, the president analysed the state of the union
2. Recently, the president analyzed the state of the union ↔
      Recently, the president analysed the state of the union
3. Recently, the president analyzed the union ↔
      Recently, the president analysed the union
```

The reason for this is that the lexical item *the* will end up in both a similarity and a difference in a match sequence of any two of these examples [30]. Such as :

```
      Recently, the president analyzed the (state of the, ε) union
      Recently, the president analyzed (the state of, ε) the union
```

for the sentences 2 and 3. In [30], it cannot choose one of them as the correct template, but this is not a problem for our algorithm, since it generates and accepts all possible templates (both of the templates above, in this example). In TDL, we can use constraint level and scoring for restrictions on the learned templates. If we define the constraint level as 1, TDL learns three different templates from these examples, which are:

```
1. X the president analyzed the state of the union ↔
      Y the president analysed the state of the union
2. X the president analyzed the Y union ↔
      M the president analysed the N union
3. Recently, the president analyzed the X union↔
      Recently, the president analysed the Y union
```

These templates are learned since their fragmentation scores are greater than other templates. TDL learned the third template:

```
      Recently, the president analyzed the X union↔ ...
```

instead of

```
      Recently, the president analyzed Y the union↔ ...
```

because the fragmentation score of the first one is 26, and 20 for the second one, if we count the words as the terminals. If the constraint level is increased to 2, then a few more templates are added to list. One of them is:

```
X the president analyzed Y the union ↔
      M the president analysed N the union
```

This template is important since it shows "the union" as one structure. Thus, we can say that TDL can find the "wanted" similarities between the examples. On the other hand, TDL might learn another template such as

```
X the president analyzed Y the union ↔
      M the president analysed the N union
```

Fragmentation score of this template will be greater than the previous template. So this will be higher priority to be selected.

Example 5.5: This example shows the powerful and weak sides of the TDL. Following example translations "I saw you at the garden" ⇔ "Seni bahçede gördüm" and "I saw you at the party" ⇔ "Seni partide gördüm" are given with their lexical level representations [21]:

```
    i see+p you at the garden ↔ sen+yH bahçe+DA gör+DH+m
      i see+p you at the party ↔ sen+yH parti+DA gor+DH+m
```

TDL, generates the following output:

```
      ttl(i see+p you at the X, sen+yh Y+da gor+dh+m).
```

From these examples with one pair of differences in both sides, following translation templates are learned by the TTL algorithm:

$$
\begin{array}{c}
\texttt{[i see+p you at the X}^1\texttt{]} \leftrightarrow \texttt{[sen+yH X}^2\texttt{+DA gör+DH+m]} \\
\texttt{if [X}^1\texttt{]} \leftrightarrow \texttt{[X}^2\texttt{]} \\
\texttt{[garden]} \leftrightarrow \texttt{[bahçe]} \\
\texttt{[party]} \leftrightarrow \texttt{[parti].}
\end{array}
$$

TTL learns information from the differences. “garden” ↔ “bahçe” pair and “party” ↔ “parti” pair are learned from the differences. But, TDL algorithm can only learn the similar part as translation rule. If, on the other hand, the number of differences are equal on both sides, but more than one, i.e., $1 < n = m$, without prior knowledge, it is impossible to determine which difference pairs in one side corresponds to which difference pairs on the other side [21]. Equal number of variables/differences on both sides is really needed? The next example will be about this.

Example 5.6: In this example, we will try to make a translation with the templates with different number of variables/differences on both sides. Let us assume that following translation pairs are given:

```
  my party was good ↔ benim partim güzeldi
your party was good ↔ senin partin güzeldi
          my school ↔ benim okulum
```

To show the process as simple as possible, following templates are generated by hand. Since TDL does not consider the word boundaries, with TDL generated templates it would be difficult to explain the translation in a few lines.

ttl(X^{s1} party was good, X^{t1} partiY^{t1} güzeldi).
ttl(my X^{s2}, benim X^{t2}m Y^{t2}).

With these two templates, although we can only translate “my party was good” ↔ “benim partim güzeldi”, this translation is very valuable. Since it shows that we can make a translation with the templates that have unequal variables on both sides.

If “my party was good” is given as the source, 2nd template is found, since “my” parts match, then we search for “party was good”; template should begin with a variable and we find the 1st template. So we can say that “benim” corresponds to X^{t1}, “parti” matches with X^{t2}, Y^{t1} with “m”, and “güzeldi” with Y^{t2}. Thus, “benim partim güzeldi” can be generated.

Moreover, if append one more example (“your school” ↔ “senin okulun”) to our training set, we can learn two more templates for translation:

ttl(your X^{s3}, senin X^{t3}n Y^{t4}).
ttl(X^{s4} school, X^{t4} okulY^{t4}).

By the help of these templates, we can translate “your school”, “my school” and “your party was good” too. This means that we can generate all the given positive examples. Besides, this example shows that if we can provide enough examples, TDL generate templates that can translate at least the given example set.

Chapter 6

Conclusion and Future Work

In this thesis, a different approach to the generalization of predicates with string arguments has been presented.

The main goal of this project was to be able to generalize the examples which have more than one same similar parts. This goal has been achieved as it is showed in Section 5.2.2.

In the previous versions of the algorithm, finding match sequences has strict rules. These rules are relaxed in this project. This caused learning similarities, in an effective way, but during learning of similarities, learning of differences is omitted. Unfortunately, we see that learning differences is very important for translation template learning.

Moreover, we see that for multi-arity predicates dependencies between the arguments with same alphabet are very important for past tense learning. Although, our algorithm can find the similar parts of the present and past forms of verbs perfectly, finding the past form of a given verb is not easy with the current version of our work.

For the single arity predicate examples, our algorithm can learn functions such as membership, but for applications such as DNA sequence alignment, our current alignment strategy is too slow for practical usage, since our search algorithm tries to find all possible alignments.

From the paragraphs above, future work can be extracted. These can be:

- For 2-arity predicates that have the same alphabet in both arguments, similar parts should be detected and variables should be the same for the same parts of the arguments.

- Instead of using breadth-first search to find all possible alignments, dynamic programming can be employed to find the alignments that have scores above a value.
- For translate template learning, the algorithm should learn new templates from the difference parts of the examples.

In conclusion, we see that our greedy approach to find the best possible similarity works; but without adding other features, usage of this tool is rather difficult.

References

[1] S. Karlin and S.F. Altschul, "Methods for assessing the statistical significance of molecular sequence features by using general scoring schemes", Proc. Natl. Acad. Sci. U.S.A. 87, pp 2264-2268, 1990.

[2] S. Karlin and S.F. Altschul, "Applications and statistics for multiple high scoring segments in molecular sequences", Proc. Natl. Acad. Sci. U.S.A. 90, pp 5873-5877, 1993.

[3] S. Muggleton, R. King and M. Sternberg, "Protein secondary structure prediction using logic-based machine learning", Protein Engineering, v 5, pp 647-657, 1992.

[4] W.R. Pearson, "Searching protein sequence libraries: comparison of the sensitivity and selectivity of the Smith-Waterman and FASTA algorithms", Genomics 11(3), pp 635-650, 1991.

[5] R. Bundschuh, "Rapid significance estimation in local sequence alignment with gaps", RECOMB 2001, pp 77-85, Montreal, Canada.

[6] T. Hwa and M. Lassig, "Optimal detection of sequence similarity by local alignment", RECOMB 98, pp 109-116, NY, USA.

[7] S.F. Altschul, W. Gish, W. Miller, W.W. Myers, and D.J Lipman, "Basic local alignment search tool", J. Mol. Biol. 215, pp 403-410, 1990.

[8] T.F. Smith and M.S. Waterman, "Identification of common molecular subsequences", J. Mol. Biol. 147, pp 195-197, 1981.

[9] R. Bundschuh, "An analytical approach to significance assessment in local sequence alignment with gaps", RECOMB 2000, pp 86-95, Tokyo, Japan.

[10] S.F. Altschul, "A protein alignment scoring system sensitive at all evolutionary distances", J. Mol. Evol. 36, pp 290-300, 1993.

[11] L. Wang, T. Jiang and D. Gusfield, "A more efficient scheme for tree alignment", RECOMB 97, pp 310-318, Santa Fe New Mexico, USA.

[12] L. Wang and T. Jiang, "On the complexity of multiple sequence alignment", Journal of Computational Biology 1, pp 337-348, 1994.

[13] S. Altschul and D. Lipman, "Trees, stars, and multiple sequence alignment", SIAM Journal on Applied Math. 49, pp 197-209, 1989.

[14] R. Ravi and J. Kececioglu, "Approximation algorithms for multiple sequence alignment under a fixed evolutionary tree", Proc. 6th Combinatorial Pattern Matching Conf., pp 330-339, 1995.

[15] B. Schwikowski and M. Vingron, "The deferred path heuristic for the generalized tree alignment problem", RECOMB 97, pp 257-266, New Mexico, 1997.

[16] R.L. Rivest, Learning decision lists. Machine Learning 2(3), pp 229-246, 1987.

[17] R.J. Mooney and M.E. Califf, "Induction of first-order decision lists: results on learning the past tense of english verbs", Journal of Artificial Intelligence Research 3, 1995.

[18] F. Bergadano, D. Gunetti and U. Trinchero, "The difficulties of learning logic programs with cut", Journal of Artificial Intelligence Research 1, pp 91-107, 1993.

[19] G.I. Webb and N. Brkie, "Learning decision lists by prepending inferred rules", Proceedings of the Australian Workshop on the Machine Learning and Hybrid Systems, pp 6-10, 1993.

[20] H. Kitano, "A comprehensive and practical model of memory-based machine translation", Proc. of the Thirteenth Int. Joint Conf. on Artificial Intelligence, Morgan Kaufmann v 2, pp 1276-1282, 1993.

[21] H.A. Guvenir and I. Cicekli, "Learning translation templates from examples", Proc. of the 6th Annual Workshop on Information Technologies and Systems (WITS'96), pp 112-123, 1996.

[22] I. Cicekli and H.A. Guvenir, "Learning translation rules from a bilingual corpus", Proc. of the 2nd Int. Conf. on New Methods in Language Proc. (NeMLaP-2), pp 90-97, 1996.

[23] Z. Oz, "Confidence factor assignment to translation templates", Msc. Thesis, Bilkent University, 1998.

[24] G.D. Plotkin, "A note on inductive generalization", Machine Intelligence 5, pp 153-163, 1971.

[25] G.D. Plotkin, "Automatic methods of inductive inference", Ph.D. Thesis, Edinburg University, 1971.

[26] S. Muggleton and C. Feng, "Efficient induction of logic programs", Inductive Logic Programming, pp 281-198, 1992.

[27] I. Cicekli, "A specific least general generalization of strings and its application to example-based machine translation", Proceedings of the 10th Turkish Symposium on Artificial Intelligence and Neural Networks (TAINN2001), North Cyprus, 2001.

[28] H.A. Guvenir and I. Cicekli, "Learning translation templates from examples", Information Systems, v 23, no 6, 1998.

[29] I. Cicekli, "Similarities and differences", Proceedings of SCI2000, Orlando, FL, USA, pp 331-337, July 2000.

[30] I. Cicekli and H.A. Guvenir, "Learning translation templates from bilingual translation examples", Journal of Applied Intelligence, pp 1-24, 2001.

[31] S. Henikoff and J.G. Henikoff. "Amino acid substitution matrices from protein blocks", Proc. Natl. Acad. Sci. U.S.A. 89(2), pp 10915-10919, November 1992.

[32] Z. Oz and I. Cicekli, "Ordering translation templates by assigning confidence factors", Proceedings of AMTA'98-Conference of the Association for Machine Translation in the Americas, Lecture Notes in Computer Science 1529, Springer Verlag, pp 51-61, Langhorne, PA, USA, October 1998.

[32] G. Myers, S. Selznick, Z. Zhang and W. Miller, "Progressive multiple alignment with constraints", RECOMB'97, pp 220-225, Santa Fe New Mexico, USA, 1997.

[33] J.R. Quinlan, R.M. Cameron-Jones, "FOIL: A midterm report", Proceedings European Conference on Machine Learning, pp 3–20, Vienna,1993.

[34] R.S. Michalski, I. Mozetic, J. Hong and N. Lavrac, "The multipurpose incremental learning system AQ15 and its testing application to three medical domains", Proceedings of Fifth National Conference on Artificial Intelligence, Philedelphia, pp 1041-1054, 1986.

[35] J.R. Quinlan, "Learning logical definitions from relations", Machine Learning 5, pp 239-266, 1990.

[36] S. Muggleton and C. Feng, "Efficient induction of logic programs", Inductive Logic Programming, pp 281-198, 1992.

[37] S. Muggleton and W. Buntine, "Machine invention of first-order predicates by inverting resolution", Proc. of The Fifth International Conference on Machine Learning, pp 339-352, 1988.

[38] G.D. Plotkin, "Automatic methods of inductive inference", PhD thesis, Edinburg University, 1971.

[39] G.D Plotkin, "A further note on inductive generalization", Machine Intelligence, vol 6, 1971.

[40] E.L. Anson and E.W. Myers, "ReAligner: A program for refining dna sequence multi-alignments", Proceedings of the First Conference on Computational Molecular Biology, pp 9-16, ACM-Press, 1997.

[41] M.E. Califf, "Learning the past tense of English verbs: An inductive logic programming approach", unpublished project report, 1994.

[42] Dataset for past tense learning: http://www-ai.ijs.si/~ilpnet2/apps/ptense.html.

Appendix A

Data Structures

This chapter represents the data structures used in implementation.

Contents of “Aligner.h”

//This header defines the similarity-difference and template structures,
//Alignment manager module as Aligner class and methods of this
//class as the called modules. Aligner class holds the input array and the
//template arrays.
//---//

```
#include <string>

using namespace std;

#include "Genel.h"
#include "Parameter.h"
#include "SimpleParam.h"
#include "MyTemplate.h"

typedef struct out_struct
{
        //string str;
        CSimpleParam pred;
        int nScore;
        int nFrag;
}OUTPUT_TYPE;

typedef struct temp_struct
{
        //string str;
        CSimpleParam pred;
        int nScore[MAX_PARAM];
        int nFrag[MAX_PARAM];
        unsigned char nVar[MAX_PARAM];
        float nCoverScore[MAX_PARAM];
        float nTotalScore[MAX_PARAM];
}TEMPLATE_TYPE;

enum SortType {byScore = 0, byFrag, byCoverScore, byTotalScore      };

typedef CMyTemplate           MYTEMPLATEARRAY[200][2] ;
```

```
#include <map>

class CAligner
{
	int _nIndex;			//output array index
	int _nTemplateIndex[MAX_PARAM];	//template index
	int _nFinalTemplateIndex[MAX_PARAM];
	int _nCurrIndex; //kacinci parametre align ediliyor
	int _nRow; //parse ederkenki satir sayisi, ne kadar input oldugu

	float m_fCoverFactor;
	float m_fAlignFactor;

	OUTPUT_TYPE _outputArray[1];//[MAX_ARRAY];
	TEMPLATE_TYPE _templateArray[MAX_TEMP_ARRAY];

//	OUTPUT_TYPE _finalOutputArray[MAX_ARRAY];
	TEMPLATE_TYPE _finalTemplateArray[MAX_ARRAY];

	string cszStr1;
	string cszStr2;

	string pred_name;
	int pred_num;
	int m_nAlphabet;

	CParameter m_param;

public:
	CSimpleParam inputArray[MAX_INPUT_ARRAY];
	int min_templateScoreIndex;

public:
	void createSelectionSets();
	void selectionSet_1();
	void selectionSet_2();
	void selectionSet_3();
	void selectionSet_4();


	void removeDuplicates();
	void moveGenerated();
	void multipleMoveGenerated();
	BOOL parse(FILE *f);
	float calculateTotalScore(int n);
	float calculateConfidenceFactor(string strTemplate, int nIndex, TEMPLATE_TYPE
array[]);

	CAligner();
	virtual ~CAligner();

	void align(string str1, string str2);
	void align();
	BOOL isCovers(string strTemplate, string strInput);
	BOOL isCovers(CSimpleParam prmTemplate, CSimpleParam prmInput);
	void giveScore(string str);
	int divide(int start, char ch, string x1, string &pre, string &post);
	void postProcess(string str);
```

```
        void findAlignment(string x1, string x2, string out, string dif_a, string dif_b, 
BOOL bDif);

        void sort();
        void sortTemplate();
        void sortFinalTemplate(TEMPLATE_TYPE array[], int arrayIndex[],int nType);
        void printTemplates();

        void removeRedundant(int nIdex, MYTEMPLATEARRAY array);

private:
        int min_templateScore;
        void swap(int n1, int n2);
        void swapTemplate(int n1, int n2);
        void swapFinalTemplate(TEMPLATE_TYPE array[], int n1, int n2);
        void addSimDif(string strSimDif, int nFrag, int nScore);
        void addTemplate(string strTemplate, int nFrag, int nScore, int nVarCount);
        void addToFinalTemplate(TEMPLATE_TYPE temp);
        void addToTempFinalTemplate(TEMPLATE_TYPE array[], int arrayIndex[], 
TEMPLATE_TYPE temp);
};
```

Contents of MyTemplate.h

//CmyTemplate is a generic class which can hold the string of the template

//it has methods that can give some of the properties of the template, such as

//fragmentation score, number of constants, etc.

```
//----------------------------------//

#include "Genel.h"

class CMyTemplate : public CMyString
{
public:
        static int numberOfVariables(string s);
        double alphabetCoverage(int nLen=0);
        int fragScore();
        int numOfConsts();
        int numOfFrags();
        CMyTemplate();
        virtual ~CMyTemplate();
        CMyTemplate operator=(string s);
};
```

Contents of Parameter.h

//Cparameter class is used to hold the default and given parameter values.

//Values are set by default or during the parsing of the input file.

//When the value of a paameter is needed, related method is called.

```
//---------------------------------//
class CParameter  :public CSimpleParam
{
```

```
protected:
	string strParam[10];
	string strValue[10];

public:
	int isUnbalancedVariableOK();
	int getConstraintLevel();
	int getMaxTemplateNumber();
	int getSelectionSet();
	float getAlignFactor();
	float getCoverFactor();
	string get(int index);
	BOOL add(string val);
	string get(string key);
	BOOL add(string key, string val);
	BOOL add(int key, string val);
	CParameter();
	virtual ~CParameter();

};
```

Contens of SimpleParam.h

//This is the base class for Cparameter class and

//it is used to hold the arguments of a predicate in template structure.

```
//---------------------------------//

#include <string>
#include "Genel.h"
using namespace std;

class CSimpleParam
{
protected:
	string strValue[MAX_PARAM];
	int nIndex;
public:
	BOOL operator==(CSimpleParam b);
	string get(int index);
	BOOL add(string val);
	BOOL add(int key, string val);
	CSimpleParam();
	virtual ~CSimpleParam();

};
```

//Contents of Genel.h

//In this file defualt values are defined such as the maximum array size.

//default coverage factor, etc.

```
//------------------------------//

#define BOOL int
#define TRUE 1
#define FALSE 0
```

```
const int MAX_ARRAY    =4000;
const int MAX_INPUT_ARRAY=2000;
const int MAX_TEMP_ARRAY=500;
const int STRING_SIZE  =20;
const int MAX_PARAM    =2;
const int MAX_PRED_NUM = MAX_PARAM;
const float ALIGN_FACTOR      = (float)0.15;
const float COVER_FACTOR      = (float)0.50;


#include <string>
using namespace std;

class CMyString :public string
{
public:
        static string lowerCase(string str);
        string lowerCase();
        string lowerCaseOf();
        int findUpper(int nStart);
};
```

Appendix B

Example Sets

This is an example of a file in FASTA format. This file includes some example protein sequences, such as zea, oryza, etc. Most of the sequence alignment programs can take input in FASTA format. The following examples are already aligned sequences.

********FILE STARTS BELOW THIS LINE**********

```
>zea
MTKAIPKIGS---RKKVRIGLRRNARFSLRKSARRITKGIIHVQASFNNT
IITVTDPQGRVVFWSSAGTCGFKSSRKASPYAGQRTAVDAIRTV---GLQ
RAEVMVKGAGSGRDAALRAIAKSGVRLSCIRDVTPMPHNGCRPPKKRRL
>oryza
MTKAIPKIGS---RRKVRIGLRRNARFSLRKSARRITKGVIHVQASFNNT
IITVTDPQGRVVFWSSAGTCGFKSSRKASPYAGQRTAVDAIRTV---GLQ
RAEVMVKGAGSGRDAALRAIAKSGVRLSCIRDVTPMPHNGCRPPKKRRL
>nicotiana
MAKAIPKISS---RRNGRIGSR--------KGARRIPKGVIHVQASFNNT
IVTVTDVRGRVVSWSSAGTSGFKGTRRGTPFAAQTAAANAIRTVVDQGMQ
RAEVMIKGPGLGRDAALRAIRRSGILLTFVRDVTPMPHNGCRPPKKRRV
>spinacia
MAKPIPKIGS---RRNGRISSR--------KSARKIPKGVIHVQASFNNT
IVTVTDVRGRVVSWASAGTCGFRGTKRGTPFAAQTAAGNAIRTVVEQGMQ
RAEVMIKGPSLGRDAALRAIRRSGILLSFVRNVTPMPHNGCRPPKKRRV
>pisum
MAKSIPKIGS---RKTGRIGSR--------KHPRKIPKGVIYIQASFNNT
IVTVTDVRGRVISWSSAGSCGFKGTRRGTPFAAQTAAGNAIQTVVEQGMQ
RAEVRIKGPGLGRDAALRAIYRSGILLKVIRDVTPLPHNGCRAPKKRRV
>geranium
MAKPIRKYWRYNLRRNRRIRLR--------KNIRKIEKGIIHVQANFSNT
```

LVTITDRKGRVVIWDSAGACGFKGRRRGTPFAAQTTTQNAIQPLVRQGMK
RVSVLIKGIGRGRDAALRAIFRSRVRVRLIRDITPMPHNGCRPPKKRRT
********FILE ENDS ABOVE THIS LINE**********

Alignment of oryza and nicotiana by our algorithm is

```
mXkaipkiXs---rrXrigXrXkXarriXkgvihvqasfnnt
iXtvtdXgrvvXwssagtXgfkXrXpXaXqXaXairtvXgXq
raevmXkgXgXgrdaalraiXsgXlXrdvtpmphngcrppkkrrX
```

The only difference between the sequences above and the result of our program is the variables. By the help of these variables we can understand the different and similar parts easily.

Appendix C

Mid-level Output for Past Tense Learning

This is the mid-level output which shows the top scored templates. It is shortened up to 100 top scored templates for space considerations. 1392 examples is used for this experiment. This data is obtained from [44].

Argument 1	**Argument 2**	**Frag.**	**Coverage**	**Total Score**
Xemonstrate	Xemonstrated	12100	0.001437	1815.001
XaYticipate	XaYticipated	5330	0.001437	799.5008
Xnstitute	Xnstituted	5184	0.001437	777.6008
Xrighten	Xrightened	3969	0.001437	595.3508
Xnstruct	Xnstructed	3969	0.001437	595.3508
Xrespond	Xresponded	3969	0.001437	595.3508
Xtribute	Xtributed	3136	0.002155	470.4011
Xstitute	Xstituted	3136	0.002155	470.4011
Xescribe	Xescribed	3136	0.001437	470.4008
Xreserve	Xreserved	3136	0.001437	470.4008
XrYighten	XrYightened	2405	0.002155	360.7511
XoYstruct	XoYstructed	2405	0.001437	360.7507
XtYighten	XtYightened	2405	0.001437	360.7507
Xighten	Xightened	2304	0.003592	345.6018
Xlatter	Xlattered	2304	0.001437	345.6007
Xregard	Xregarded	2304	0.001437	345.6007
Xlisten	Xlistened	2304	0.001437	345.6007
Xstruct	Xstructed	2304	0.002155	345.6011
Xresent	Xresented	2304	0.002155	345.6011
XtYribute	XtYributed	1850	0.002155	277.5011
cXculate	cXculated	1850	0.001437	277.5007
XsYtitute	XsYtituted	1850	0.002155	277.5011
XlYminate	XlYminated	1850	0.001437	277.5007
XeYculate	XeYculated	1850	0.001437	277.5007
Xrespond	XresponYdZ	1813	0.001437	271.9507

Xrender	XrYendered	1800	0.001437	270.0007
Xresent	rXesented	1800	0.001437	270.0007
Xlocate	Xlocated	1764	0.001437	264.6007
Xnounce	Xnounced	1764	0.002155	264.6011
Xsemble	Xsembled	1764	0.001437	264.6007
Xassure	Xassured	1764	0.001437	264.6007
Xelieve	Xelieved	1764	0.001437	264.6007
Xculate	Xculated	1764	0.002874	264.6014
Xchange	Xchanged	1764	0.001437	264.6007
Xnclude	Xncluded	1764	0.001437	264.6007
Xnspire	Xnspired	1764	0.001437	264.6007
Xcrease	Xcreased	1764	0.001437	264.6007
Xeserve	Xeserved	1764	0.002155	264.6011
Xfigure	Xfigured	1764	0.001437	264.6007
Xminate	Xminated	1764	0.002155	264.6011
Xtimate	Xtimated	1764	0.001437	264.6007
Xevolve	Xevolved	1764	0.001437	264.6007
Xmulate	Xmulated	1764	0.001437	264.6007
Xroduce	Xroduced	1764	0.001437	264.6007
Xresume	Xresumed	1764	0.001437	264.6007
Xrinkle	Xrinkled	1764	0.001437	264.6007
Xtumble	Xtumbled	1764	0.001437	264.6007
Xrefer	Xreferred	1600	0.001437	240.0007
disappXrY	disappXrYed	1517	0.001437	227.5507
disappXeY	disappXrYed	1517	0.001437	227.5507
transfXrY	transfXrYed	1517	0.001437	227.5507
improvXe	improvXed	1480	0.001437	222.0007
commenX	commenXed	1440	0.001437	216.0007
designX	designXed	1440	0.001437	216.0007
flatteX	flatteXed	1440	0.001437	216.0007
disappXeY	disappXeYd	1406	0.001437	210.9007
disappXrY	disappXeYd	1406	0.001437	210.9007
transfXrY	transfXeYd	1406	0.001437	210.9007
XpreciYate	XpreciYated	1394	0.001437	209.1007
Xnounce	XnYounced	1332	0.002155	199.8011
Xapprove	XaYpZroved	1323	0.001437	198.4507
Xcounter	XcountYeZd	1323	0.001437	198.4507
XcYulate	XaYculated	1300	0.001437	195.0007
XcYatter	XcYattered	1300	0.001437	195.0007
XeYclaim	XeYclaimed	1300	0.001437	195.0007
sXatter	sXattered	1300	0.001437	195.0007
sXagger	sXaggered	1300	0.001437	195.0007
XnYounce	Xnounced	1274	0.002155	191.1011
Xtribute	XiYbuted	1274	0.002155	191.1011

Xcounter	XnYtered	1274	0.001437	191.1007
Xbound	Xbounded	1225	0.001437	183.7507
Xcount	Xcounted	1225	0.002874	183.7514
Xdress	Xdressed	1225	0.002155	183.7511
Xallow	Xallowed	1225	0.001437	183.7507
Xalter	Xaltered	1225	0.001437	183.7507
Xmount	Xmounted	1225	0.002155	183.7511
Xpoint	Xpointed	1225	0.001437	183.7507
Xtract	Xtracted	1225	0.002874	183.7514
Xatter	Xattered	1225	0.00431	183.7522
Xother	Xothered	1225	0.001437	183.7507
Xenter	Xentered	1225	0.001437	183.7507
Xclaim	Xclaimed	1225	0.002874	183.7514
Xnsist	Xnsisted	1225	0.001437	183.7507
Xntain	Xntained	1225	0.001437	183.7507
Xntend	Xntended	1225	0.001437	183.7507
Xcover	Xcovered	1225	0.002155	183.7511
Xcrawl	Xcrawled	1225	0.001437	183.7507
Xesign	Xesigned	1225	0.001437	183.7507
Xelect	Xelected	1225	0.001437	183.7507
Xtreat	Xtreated	1225	0.002155	183.7511
Xblish	Xblished	1225	0.001437	183.7507
Xhibit	Xhibited	1225	0.002155	183.7511
Xpress	Xpressed	1225	0.002155	183.7511
Xasten	Xastened	1225	0.001437	183.7507
Xather	Xathered	1225	0.001437	183.7507
Xinger	Xingered	1225	0.001437	183.7507
Xicker	Xickered	1225	0.001437	183.7507
Xutter	Xuttered	1225	0.002155	183.7511
Xlower	Xlowered	1225	0.001437	183.7507
Xround	Xrounded	1225	0.002155	183.7511
Xammer	Xammered	1225	0.001437	183.7507