A Symmetric Version Space Algorithm for Learning Disjunctive String Concepts

Jacky Baltes

April 16, 1997

Abstract

The symmetric version space algorithm (SVS) learns disjunctions of string patterns by example. The learnable string concepts are a subset of regular expressions. The running time of the algorithm is *reduced*, because the system learns a top–down description of the string concepts. Different parts of the algorithm learn descriptions at different levels of the concept independently. This technique is similar to factoring the version space, in order to restrict the search space. The problem of *fragmentation* of the G–set is overcome by using a symmetric version space approach.

Contents

1 Introduction

In order to develop intelligent agents used in future applications, it is important to be able to learn repetitive tasks involving strings. The motivation for the string learning algorithm was as part of an ongoing research effort to design a system that allows a novice user to teach repetitive operating–system tasks by example. However, the results can be transferred to other domains with strings as primary data such as editors, text formatters, databases, and compilers.

Maintaining a file system, organizing mail messages or news articles are examples of simple, but highly repetitive tasks. These tasks are tedious, frustrating, and error–prone, so they should be automated. Unfortunately, the organization of a file system is very dependent on the personal preferences of the user. For example, some users prefer seperate subdirectories for different projects, whereas others prefer to use filenames to distinguish between projects. So some users prefer a large number of files in a directory, as opposed to only a few files in each directory. The shape of the directory tree also differs from very shallow and broad trees to deep and narrow ones.

Therefore, it is very difficult to write a general purpose program suitable for all or even the majority of users. Specific programs can be written in a general purpose programming language or script language, but programming is a very time consuming task in itself and not all users are skilled programmers. Furthermore, a user's preferences are likely to change over time which means that new programs must be written.

My research is directed towards designing a fast, easy to use system that allows the user to teach the computer these repetitive tasks by simply giving examples of the required procedure. Teaching by example is a very effective way to communicate the necessary task knowledge and seems to be particularly suited for this application.

In the absence of specific domain knowledge which would prevent the developed algorithms being useful in a wider range of domains, the system has to learn to decide whether a command has to be applied to given strings (representing file names, mail addresses, subjects of messages etc.) or not, based on the syntax of these strings.

Conjunctive concepts are too restrictive. Although many concepts can be expressed as a conjunction of attribute values, some common concepts can only be expressed as disjunctions. Simple examples in the UNIX domain are: all Gnu Emacs backup files (*~ OR #*#), or all C–language source files (*.c OR *.h).

Therefore, the system must have a learning module that interactively learns disjunctive string concepts by example. Section 1.1 gives a complete description of the learning model.

1.1 The Learning Paradigm

The learning paradigm is based on the assumption that the user shows the system an example of a concept, and after this first example, the algorithm tries to classify all other strings automatically. If the learning algorithm fails to classify a string, it can ask the user for the correct classification. The learning algorithm's model of the concept is then updated.

The main objectives of an algorithm designed for this learning model are:

- To try and automate the task as soon as possible. In fact, the algorithm attempts to automate the task after the first example.
- Minimize the number of questions to the user.
- Simplify the cognitive load on the user, by asking only simple questions. Questions such as "What is the correct regular expression for the concept" are not allowed.

Figure 1 is an abstraction of for example a system, in which the system learns to manipulate files in a directory. The conceptual learning model consists of five distinct entities, the initiator, the task performer, the oracle, the learner/classifier, and an example source. The initiator recognizes the need to learn a concept, in order to perform a given task. It provides the learner/classifier with the first example of a concept, which will always be positive. As will be discussed in section 4.1, the first example plays a special role, because the algorithm described in this paper assumes that the first example is a good representative of the concept to be learned.

The learner/classifier constructs an internal model of the concept to be learned and fetches more examples from the example source. It tries to classify the examples. If the classifier successfully recognizes these examples as members or non–members of the concept, this classification is passed on to the task performer.

Only if the classifier fails to classify an example, it is passed on to the learner. The learner consults an oracle about the correct classification of this example and updates the model of the concept. The previously unknown example is then known and passed on to the task performer.

The task performer is made explicit in this model in order to stress the importance of having a reason to learn. A learning system can not be seen independent of its task. In an ideal system, the task performer would realize the necessity of learning a concept and invoke the initiator.

Although conceptually different, there is no requirement for the initiator, the oracle and the example source to be physically distinct. In one implementation of the learning algorithm, the task performer (the UNIX shell) functions as the example source. This reduces the cognitive load on the oracle (the user in the implementation), because only examples that actually occur in the task must

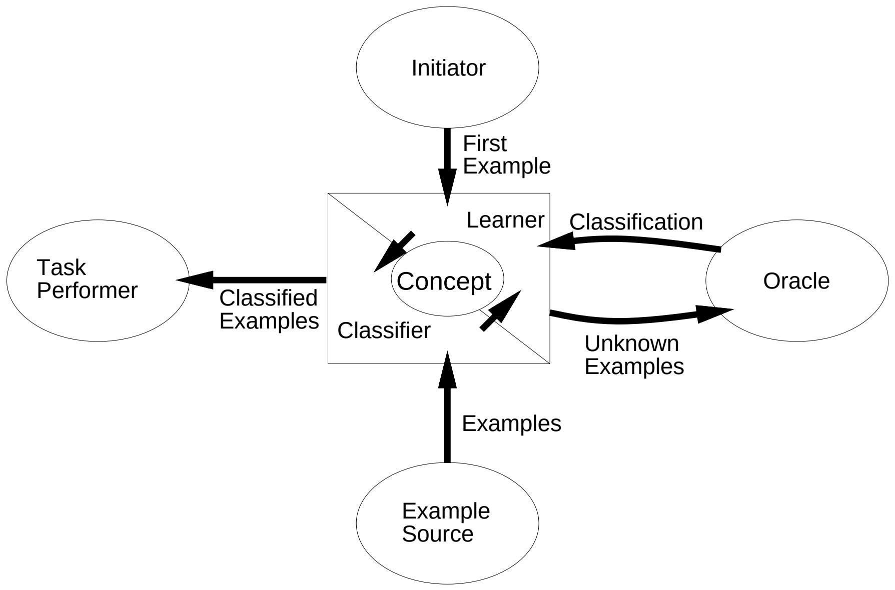


Figure 1: Interactive Learning Model

be learned and classified. It is also conceivable, that a system asks the oracle hypothetical questions such as “Would file *xyz* belong to the concept?” Also, in the example implementation, the user combines the functionality of the initiator and the oracle. Using special instructions, the user can force the generation of a concept and provide the first example (initiator), as well as classify unknown examples (oracle).

The interactive learning model restricts the concepts that can be learned. Every learning algorithm must generalize from past experiences to new examples. The interactive learning model, however, does not allow the system to recover from over–generalization. If the algorithm wrongly classifies a string as either positive or negative, it will be passed on to the task performer, thus making it impossible to detect over–generalization.

Therefore, generalizations must be reasonably controlled. Only generalizations that are justifiable in the given domain are allowed. This paper argues that there are rules that can be applied in a variety of different domains, that lead to useful generalizations. The rules are based on the intuitive use of different characters in a string. The character set is broken up into seven different character classes: lower case, upper case, digits, punctuation characters, operators, whitespace, and special symbols.

1.2 Design of the String Learning Algorithm

The following four problems must be solved:

- Inducing complete regular expressions is too expensive. The algorithm described in this paper restricts the representation language to a subset of regular languages, and thus allows the algorithm to be used in an interactive environment. The representation language is given in table 1. It imposes a hierarchical structure on the regular expression. Each concept is a sequence of *units* (See 4.1). The number of *units* in a concept is determined by the first example. In turn, each *unit* is a disjunction of at most three patterns. The patterns are character classes that must be separated by specific characters.

- Disjunctions must be limited, because otherwise the most specific concept that matches all positive examples is simply the disjunction of all positive examples. The described algorithm imposes a static or dynamic limit on the number of terms in a disjunction. In our example implementation, a static limit of three terms seemed to be adequate. A dynamic limit can be implemented using a similarity metric.

- Using Mitchell's Candidate Elimination Algorithm directly is inappropriate, because the G–set is infinite for limited disjunctions. Even without limited disjunctions, the G–set will grow very large, because of *fragmentation. Fragmentation* occurs because there are a large number of ways in which a pattern can be specialized to not match a given set of strings. The symmetric version space algorithm computes two *cover sets* for the set of positive and negative examples. A *cover set* is similar to the S–set of the candidate elimination algorithm. It is the most specific expression in the representation language that matches all examples in a given set. However, in contrast to the candidate elimination algorithm, if a example matches for example the positive *cover set*, it does not automatically follow that this example should be classified as positive.

- Even using limited disjunctions, any least commitment algorithm will possibly ask about all strings (infinitely many). Some method must be developed, that forces the algorithm to generalize in these situations. There are a number of possible solutions for this problem. The algorithm described in this paper maintains two extra *cover sets* to avoid asking about all strings.

The problems and the solution chosen in this paper will be described in more detail in the following sections. Subsection 1.3 section gives an outline of remainder of the paper.

1.3 Outline of the Paper

Section 2 gives a brief introduction to grammars and version spaces. It also describes Mitchell's candidate elimination algorithm. Section 3 describes the representation language and therefore the learnable concepts. Section 5 describes the symmetric version space algorithm used in this paper. It compares this algorithm to the candidate elimination algorithm using a small example. The symmetric version space algorithm is a higher level algorithm and requires a subroutine to update the cover sets and a similarity metric. Section 4 describes the algorithm to compute the *cover sets*. A *cover set* is the most specific pattern that match a given set of examples. Subsection 4.3 introduces the similarity metric used in the example implementation. Section 6 gives an extended example of the symmetric version space algorithm. Section 7 compares the algorithm to other work. The IBFA algorithm by Smith and Rosenbloom. Section 8 draws conclusions and describes directions for future research.

2 Background and Terminology

Language generators are most commonly described by grammars. Grammars are finite sets of rewrite rules. Terminals are symbols that actually occur in the language. Non–terminals are intermediate symbols that are only used in the generation of the string and do not occur in the language itself. Regular languages are languages where each rewrite rules is of the form $N \Rightarrow \sigma V$, where σ is a terminal and N, V are non–terminals. Regular languages are the weakest class of languages in the Chomsky hierarchy.

Regular languages are equivalent to deterministic finite automatons.

A version space is the space of all possible concepts expressible in the representation language. The predicate *generalizes* implies a partial order on this concept space.

Developed by Mitchell [Mit77]. Uses compact representation of the Version Space. Version Space is partial ordering of all possible concepts by generality. Representation Language determines the learnable concepts.

A presentation is any finite sequence of examples that are classified as either positive or negative.

Candidate elimination is an algorithm designed by Mitchell that uses an efficient way of representing the version space by its boundaries.

Fragmentation of the G–set means that the G–set grows exponentially, because there are many ways in which a pattern can be made more specific.

3 Representation Language

A useful representation language for concepts has to be chosen. The choice of representation language is very important, because a concept can only be learned if it is expressible in the representation language. For example, if the representation language only distinguishes between numbered cards and face cards, concepts such as "Queen of hearts" can never be learned. On the other hand, concept learning can be viewed as search through the space of all possible concepts, which means that the more powerful the representation language, the greater the search space [?].

As a first approximation, the representation language is limited to regular expressions. The algorithm is designed to work in domains, such as operating system shells, that allow the user to specify only subsets of regular expressions in commands. It seems reasonable to assume that users would organize their data such that concepts can be expressed using regular languages.

Even learning regular languages is computationally very expensive. Gold showed that the problem of infering a finite state machine from its input and output is NP–hard in general [Gol78]. Since finite state machines are equivalent to regular expressions, it follows that learning regular expressions from example is NP–hard.

3.1 Restrictions of the DFA

Intuitively, the reason for the exponential complexity is that there are many regular expressions that match a given set of strings. In fact, unless some kind of reducedness criteria is applied, there is an infinite number of regular expressions that are consistent with any finite presentation. This section will describe why the representation language used in this paper reduces the complexity of the learning algorithm. Section 4 explains the algorithms to compute the expressions in the representation language.

Assume that $P = \{p_1, p_2, \ldots, p_n\}$ is the set of positive examples, N is the set of negative examples. The regular expression $R = p_1 \cup p_2 \ldots p_n$ matches all positive examples and none of the negative examples. Since there is an infinite number of strings, a new string s, that is neither in P nor in N can be found. Then the new regular expression $R' = R \cup s$ will also be consistent with the presentation. This process can be applied recursively to generate infinitely many regular expressions consistent with any given finite presentation.

However, the grammars that are generated using this method are redundent, because they make unnecessary assumptions about unknown strings. In fact, a subset of the rules for these grammars are consistent with the presentation. In order to overcome the problem of infinite consistent grammars, it is necessary to restrict the class of languages to *reduced* regular grammars. A regular grammar is *reduced*, if no proper subset of its rules is consistent with the representation.

The following two stage process constructs all reduced regular expressions that match a given set of strings and is similar to Van Lehn and Ball's algorithm [1] to compute *reduced* consistent context free grammars.

[1]See [VB87] for an in depth description

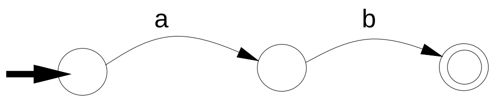
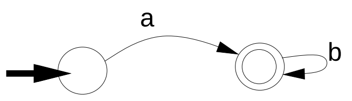
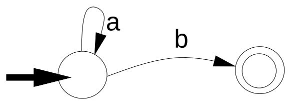
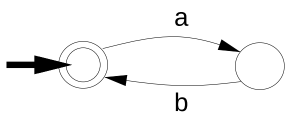
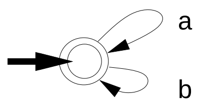
Number	Graph	Grammar	Reg. Expr.
A1		S ==> aA A ==> b	ab
A2		S ==> aA A ==> bA A ==> nil	ab*
A3		S ==> aS S ==> b	a*b
A4		S ==> aA A ==> bS S ==> nil	(ab)*
A5		S ==> aS S ==> bS S ==> nil	(a v b)*

Figure 2: Reduced regular expressions for "ab"

The first stage computes all *reduced* regular expressions for any string. Since regular expressions are equivalent to DFA, it is possible to generate the trace that a string must have taken through a DFA in order to be accepted. A string s of length n must have traversed n arcs. All regular expressions that can be generated by assigning any of the n arcs to end nodes will accept the string. Figures 2 and 3 are examples of this construction for the example strings "ab" and "c."

In the second stage, the DFA's generated by the first stage for all strings are combined to generate DFAs that accept all strings in the given set. Firstly, the cartesian product of all DFAs of the first stage is computed. Then, the nodes of the DFAs have to be assigned to nodes in the resulting DFA in all possible ways. Figure 4 gives an example of this construction for part of the set $\{ab, c\}$. However, some DFAs generated by this method yield new regular expressions. This can be seen in figure 4, where $C1$ and $C2$ represent equivalent regular expressions.

The complexity of this process is exponential in the number of strings and the length of the strings. Clearly, a faster way to compute the necessary strings

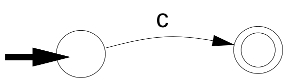
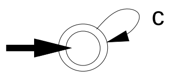
Number	Graph	Grammar	Reg. Expr.
B1		S ==> c	c
B2		S ==>cS S ==> nil	c*

Figure 3: Reduced regular expressions for “c”

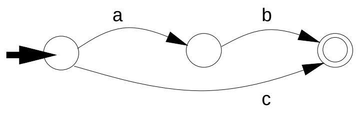
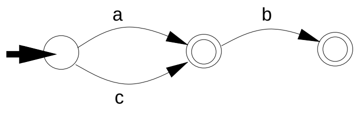
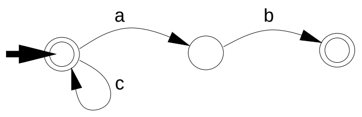
	Number	Graph	Grammar	Reg. Expr.
A1 x B1	C1		S ==> aA S ==> c A ==> b	ab v c
	C2		S ==> aA S ==> cA A ==> nil A ==> b	(a v c) v (a v c)b
A1 x B2	D1		S ==> aA S ==> cS S ==> nil A ==> b	ab v c*

Figure 4: Construction of $A1 \times B1$ and $A1 \times B2$

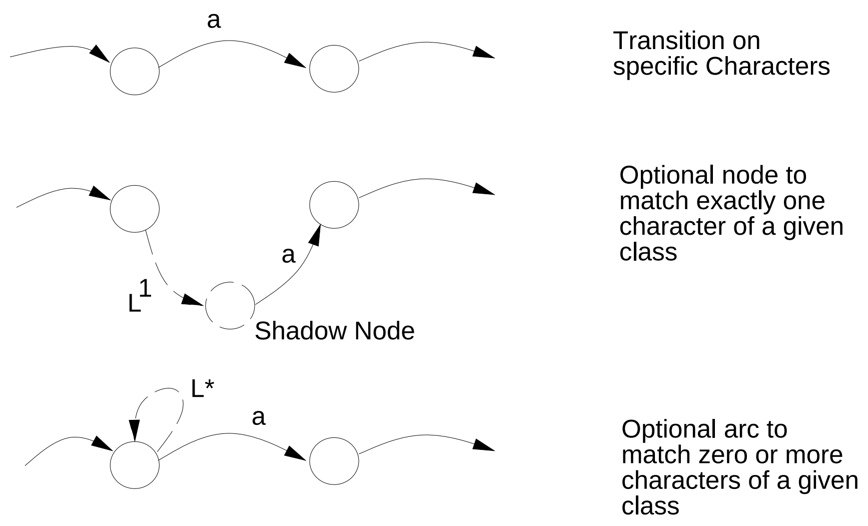

a
Transition on specific Characters
Optional node to match exactly one character of a given class
a
L1
Shadow Node
L*
a
Optional arc to match zero or more characters of a given class

Figure 5: Building Blocks for restricted DFAs

must be found in an interactive environment.

In fact, for the symmetric version space algorithm not all *reduced* regular expression consistent with a presentation are needed, but rather the most specific regular expression that matches all strings in a set. Therefore, instead of computing an exponential number of regular expressions, the symmetric version space algorithm computes only one regular expression.

Firstly, transitions between nodes are only allowed on specific characters. This allows only the construction of DFAs that match specific strings such as *abc*. Therefore, the representation language allows optional nodes in the DFA, that are used to match exactly one character in a character class (see second DFA of figure 5). The character classes are: lower case characters, upper case characters, digits, whitespace, punctuation, operators, special characters, letters (i.e. lower or upper case), alphanumeric, non–alphanumeric, or any character. Loops in the DFA are not allowed, except loops to accept zero or more characters of a specified character class such as lower case or alpha–numeric characters. Therefore, all DFAs consist of the basic building blocks described in figure 5.

3.2 Generalization Hierarchy

The advantage of allowing transitions between nodes only on specific characters is that in the second stage, different DFAs can be merged by assuming that the transitions on the same character correspond to the same nodes in the resulting DFA. In this way, the complexity of matching nodes in all possible ways is avoided.

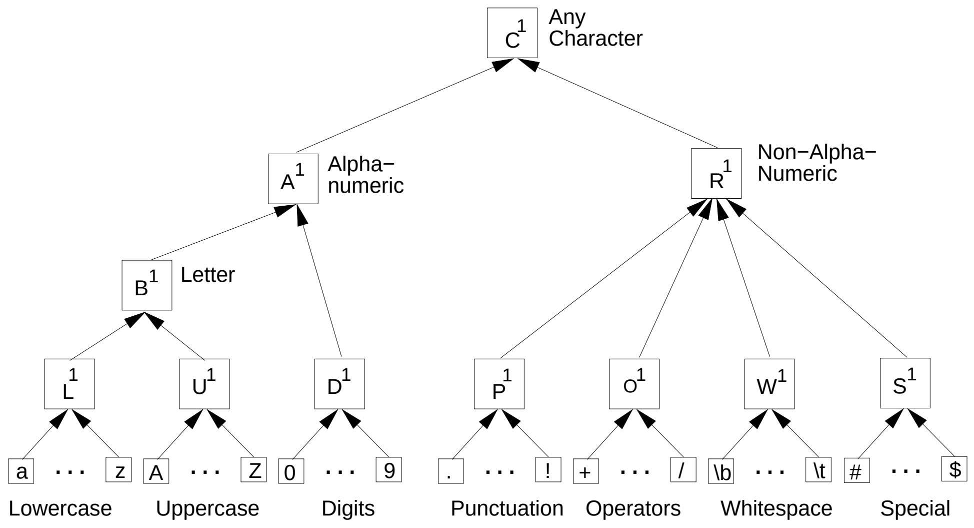


Figure 6: Generalization Hierarchy for Single Character

This requires a method to merge restricted DFAs. Figure 6 describes the generalization hierarchy for single characters. The character classes are based on the intuitive use of characters in a string. The generalization hierarchy must not only be able to generalize single characters, but also strings of characters. Every character class can be specified to match exactly one character, or zero or more characters of a given class. The complete generalization hierarchy is described in figure 7.

3.3 Sequences of Patterns

Because of the problems introduced by disjunctions, they are only allowed between different restricted DFAs, not between different nodes in the restricted DFA. Therefore, there is only one arc between nodes within the restricted DFA.

One problem with this restriction is that sequences of character classes can not be learned. For example, the system can not learn the concept zero or more lower case characters followed by zero or more whitespace characters. The algorithm described in this paper solves the problem by assuming that every concept is a sequence of independent restricted DFAs as described in the previous subsections. Since this extension of the representation language is used to learn sequences of character classes, the strings are broken up into different character classes. These substrings are called *units* of the string. For example, the string “Test123.c˜” is broken up into units as “T” “est” “123” “.” “c” “˜.” The described algorithm assumes that the first positive example is a good prototype of the concept, that is that it contains all *units* in the concept. The

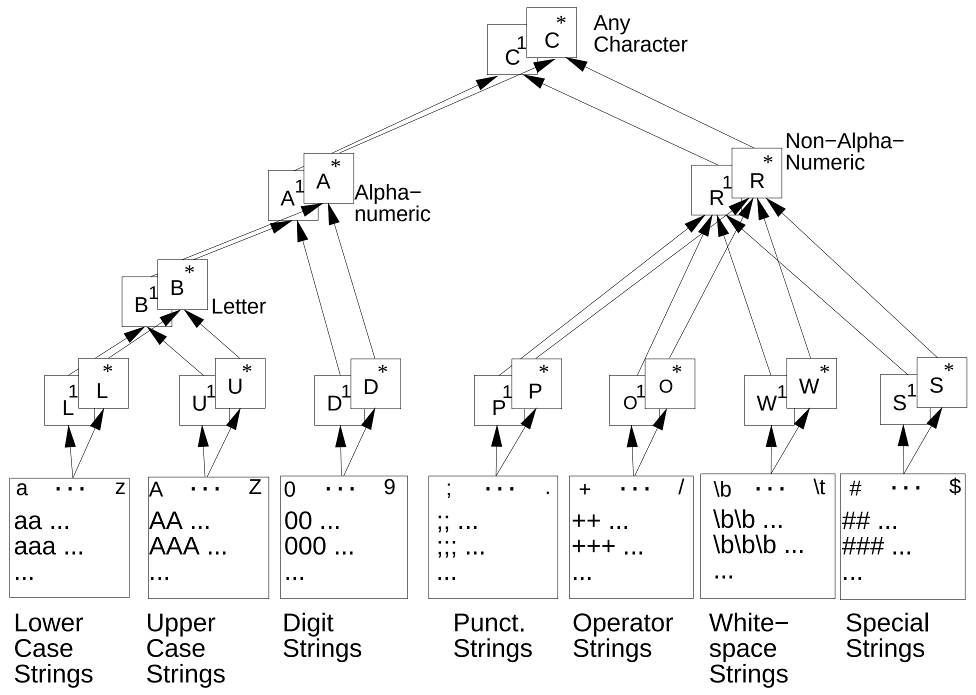


Figure 7: Generalization Hierarchy for Strings

$<concept>$	$\Rightarrow$	$<unit_1><unit_2><unit_3>\ldots<unit_n>$ where n = number of units in the first example
$<unit_i>$	$\Rightarrow$	$<disjunct>$ and not $<disjunct>$ for $i \in \{1\ldots n\}$
$<disjunct>$	$\Rightarrow$	$<pattern>$ or $<pattern>$ or $<pattern>$
$<pattern>$	$\Rightarrow$	$<charclass> \{\sigma^+ <charclass>\}^*$
$<charclass>$	$\Rightarrow$	$\epsilon\| <U> \| <L> \| <D> \| <B> \| <A> \| <C> \| <R> \| <O> \| <P> \| <S> \| <W>$
$<\alpha>$	$\Rightarrow$	$\alpha^1\|\alpha^*$ Metarule for $<U>, <L>, \ldots$

Table 1: Grammar of the Concept Description Language

first example must contain all optional elements. Subsection 4.1 describes how subsequent examples are adjusted to match the first example.

Table 1 gives the complete grammar of the representation language. The patterns in the concept grammar are extensions of Nix's gap patterns (see [Nix83]) and Moh's annotated gap patterns [?]. The character classes in Nix's gap patterns matched any number of characters, independent of their character class. Moh's annotated gap patterns allowed character classes of one or more characters of the same character class. The generalization hierarchy used by Moh is equivalent to the generalization hierarchy given in figure 6. The representation language in this paper improves on the previous work by allowing sequences of patterns and by distinguishing between exactly one and zero or more characters of a given character class. Section 4 describes in detail, how the regular expression that matches a given set of strings (i.e. the *cover sets*) are computed.

4 Updating the Cover Sets

This section describes the algorithm (called UCS algorithm) to compute the most specific description in the representation language (see table 1) for a set of strings. The most specific description that matches all positive examples is called the positive *cover set*. All negative examples are most specificaly covered by the negative *cover set*. The positive *cover set* is similar to the S–set of the Candidate Elimination algorithm. However, in contrast to the Candidate Elimination algorithm, it does not necessarily follow that an item that matches the positive (or negative) *cover set* should be classified as positive (or negative) immediately. The *cover sets* only define the boundary of the concept and must be seen together in order to classify a pattern. See section 5 for a complete description of the symmetric version space.

4.1 Adjusting the Units of a String

In order to learn sequences of patterns, the UCS algorithm breaks the string up into different *units*. This imposes a high level structure on the concept. In

```
Function Adjust(Item1,Item2)
  Find a unit that is equal in Item1 and Item2
  Test from the end of the unit to the front.
    If such a unit exists
      Split Item1 and Item2 into a left and right part
      at this unit.
      Recursively call Adjust(Left Part of Item1,Left Item2)
      Recursively call Adjust(Right Item1,Right Item2)
      return (Left Adjusted,Equal Unit,Right Adjusted)
    Else
      While(length(Item1) not equal length(Item2))
        If length of Item1>Item2
           Concatenate units in Item1 from the beginning
        Else If Item
           Append " " to Item1
    Return(Item1)
```

Table 2: Algorithm to adjust the number of units

order to reduce the complexity, the UCS algorithm assumes that the *units* are independent. This method is similar to factoring the version space as described in [GN87].

Since the SVS algorithm assumes that the number of *units* in the first example is the same as in the target concept, new *units* are never added or deleted from the concept. This means that the number of *units* in the other examples must be adjusted to be equal to the number of *units* in the first examples. This task is accomplished by the algorithm given in table 2.

For example, if the first positive example is the string "test.ss˜ ," it will be broken up into the *units* "test" "." "ss" "˜." If "concept42.c" is a new example, the *units* (i.e. "concept" "42" "." "c") are adjusted to match the *units* of the first example as follows:"concept42" "." "c" "".

4.2 The Generalize Pattern Algorithm

In order to compute the most specific description that matches all strings in a given set, it is necessary to compute the most specific generalization of a string and a pattern or two patterns. For example, the most specific generalization of the strings $test1.ss$ and $test2.c$ in the representation language is $testD^1.L^*$, which matches all strings $test$, followed by exactly one digit, followed by a period and any number of lower case characters.

The generalization of the strings is based on the maximum common subsequence (MCS) of the two strings. The MCS is computed using Hirschberg's algorithm [Hir75]. One problem of the algorithm is that the MCS of two strings is not unique [?]. The MCS of the strings *abc* and *acab* is either *ab* or *ac*. Since the MCS is only computed for *units* of a string, the UCS algorithm picks one MCS randomly.

4.3 The Similarity Metric

1. Similarity metric is the percentage of characters that are not in the max. common subsequence to total number of characters. For example:
2. This very simple metric is powerful enough to learn common concepts.
3. Domain knowledge can improve the similarity metric by trading in variety of possible domains.
4. One possibility is to give more weight to the characters at the beginning and end of a string.

5 Symmetric Version Space

The motivation for the symmetric version space (SVS) algorithm is that the most specific description of a set of strings in a representation language can be readily found and easily represented, whereas the most general patterns that do not match the negative examples are too hard to compute or too hard to represent. Therefore, the basic assumption for the SVS algorithm is that either the concept itself or the *complementary* concept can easily be learned. The Candidate Elimination algorithm assumes that the most specific description as well as the most general description of a set of strings can be computed. As the next subsection will show, the most general description can not be computed, since it is infinite for the version space of limited disjunctions.

5.1 Disjunctive Concepts

Some common concepts are only expressible as disjunctions. Examples in the UNIX domain are all C source files (i.e. *.h or *.c) or all Gnu Emacs backup files. Therefore, the string learning algorithm must be able to learn disjunctive concepts.

Unlimited disjunctions pose an immediate problem, because they allow any learning algorithm to avoid generalization. The most specific concept description that matches all positive examples is simply the disjunction of all positive examples, the so called trivial disjunction. The problem is that given:

- any finite presentation R,
- a new example s, such that $s \notin R$

- a concept description C that is the most specific description that is consistent with R,

a new concept description C' can be found that matches the old examples and only the new example $C' = C \cup s$.

The Candidate Elimination algorithm generalizes, because for some new examples, the most specific concept description matches all previous examples and the new example, plus some examples that are neither in the previous representation nor the new example. For example:

- the representation language only contains four possible concepts descriptions: a,b,c, or *anything*.
- the presentation only contains a as a positive example
- b is added to the presentation as another positive example.
- then the most specific concept that matches a as well as b is *anything*.
- c does appear neither in the presentation, nor in the new example, but will be classified as positive, because the representation language is not powerful enough to describe the concept a and b but not c.

In order to avoid trivial disjunctions, the number of disjunctions must be limited. There are two limits that can be imposed on disjunctions, either a static limit (i.e. maximum of three terms in a disjunction) or a dynamic limit. A dynamic limit will generalize two terms of the disjunction into one, if the two terms are relatively similar. If the terms are not similar enough, the new example is added as a new term to the disjunction. Therefore, a dynamic limit requires a similarity metric. See subsection 4.3 for a description of the similarity metric that was used in the example implementation of the SVS algorithm.

A static limit is a conservative approach to generalization. The algorithm will combine two terms only when the number of terms exceeds a fixed limit. Therefore, the algorithm will generalize only, when it is forced to. This seems to be a reasonable approach in an interactive learning environment because, as mentioned earlier, the algorithm is unable to detect over–generalization.

My experimental evidence suggests that a static limit of size three is adequate to learn commonly used concepts. Although a limit of three terms in a disjunction seems very restricitive, the reader must remember that the user will not see the internal representation of the concept. For the user, the usefulness of the system is not dependent on theoretical restrictions, but on the practical performance on average concepts. Furthermore, unlimited disjunctions do not seem reasonable from a psychological point of view.

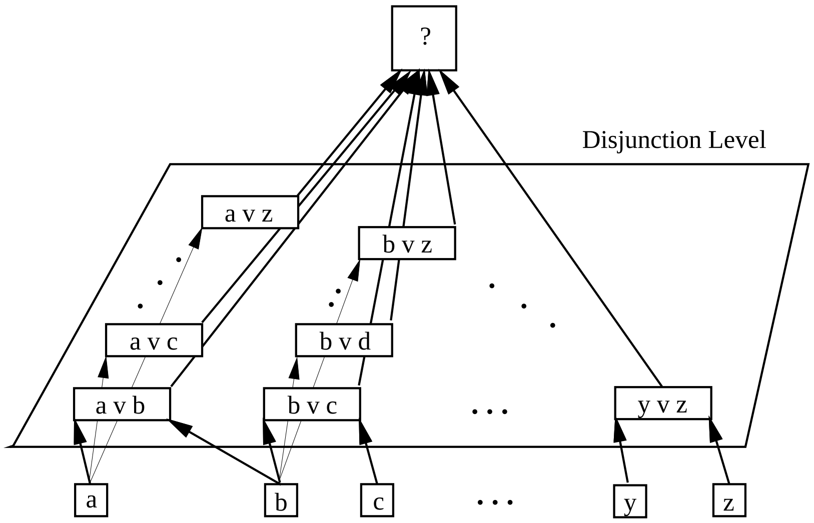


Figure 8: Version Space for Two–Disjunct Lowercase Characters

5.2 Limited Disjunctions

The Candidate Elimination algorithm can be used to learn limited disjunctions in finite domains such as lower case characters or digits. In infinite domains such as strings, however, the candidate elimination algorithm can not be used, because the G–set is possibly infinite.

Figure 8 shows the version space for limited disjunctions of size two for lower case characters. In the string domain, the version space boundaries are infinite, since at the disjunction level all combinations of an infinite number of strings must be represented.

The second problem is inherent to all least commitment algorithms when trying to learn limited disjunctions. The problem will be described using the Candidate Elimination algorithm as an example. When learning limited disjunctions, the algorithm will ask about all possible examples. This problem will always arise when the concept contains less terms in the disjunction than the maximum number of terms allowed. Using the version space described in figure 8, table 3 is a trace of the Candidate Elimination algorithm learning the concept of any lower case character. This is the best case for the Candidate Elimination algorithm. Only three examples are necessary to learn the correct concept.

Table 4 describes the performance of the Candidate Elimination algorithm in order to learn a disjunction of two terms. The example used in the table is the concept $a \cup b$. Although in this example, the system only needs three examples again, this is the best case for the Candidate Elimination Algorithm. The worst

Example	Classification	ask User	S–set	G–set
a	$+$	yes	$\mathbf{a}$	?
b	$+$	yes	$\mathbf{a \cup b}$	?
c	$+$	yes	**?**	?
d	$+$	no	?	?
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
z	$+$	no	?	?

Table 3: Trace of Candidate Elimination for Concept ?

Example	Classification	ask User	S–set	G–set
a	$+$	yes	$\mathbf{a}$	?
b	$+$	yes	$\mathbf{a \cup b}$	?
c	$-$	yes	$a \cup b$	$\mathbf{a \cup b}$
d	$-$	no	$a \cup b$	$a \cup b$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
z	$-$	no	$a \cup b$	$a \cup b$

Table 4: Trace of Candidate Elimination for Concept $a \cup b$

case for this presentation occurrs, when the correct concept is $a \cup z$. In that case, the Candidate Elimination algorithm asks about all lower case characters before learning the correct concept. In general, the Candidate Elimination algorithm requires the two positive examples to generate the correct disjunction, plus one extra example to rule out any lower case character as a possible concept.

The problem arises when the Candidate Elimination algorithm learns a disjunction with less terms in the disjunction than the maximum number of terms allowed. Table 5 is an example of this problem. The concept is the single lower case character "a" (one term disjunction). In this case, the correct concept is learned only after asking about all possible other examples. In contrast to the previous example, there is no distinction between best and worst case performance. The number of questions is independent of the ordering of the examples.

Although the Candidate Elimination algorithm generalizes correctly for concepts that are at the top of the generalization hierarchy, the worst case for the other two classes of concepts is to ask about all possible examples. In the string domain, this is equivalent to asking about an infinite number of strings. This behavior is more efficiently implemented as a database of all examples. Therefore, we must have a mechanism to reduce the questions to the oracle. A number of different mechanism are possible and should be seen independently of the learning algorithm.

One approach is to augment the interactive learning paradigm by allowing

Example	Classification	ask User	S–set	G–set
a	$+$	yes	$\mathbf{a}$	?
b	$-$	yes	a	$\mathbf{a} \cup \mathbf{c}, \mathbf{a} \cup \mathbf{d}, \ldots, \mathbf{a} \cup \mathbf{z}$
c	$-$	yes	a	$\mathbf{a} \cup \mathbf{d}, \ldots, \mathbf{a} \cup \mathbf{z}$
d	$-$	yes	a	$\mathbf{a} \cup \mathbf{e}, \ldots, \mathbf{a} \cup \mathbf{z}$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
z	$-$	yes	a	$\mathbf{a}$

Table 5: Trace of Candidate Elimination for Concept a

the oracle to refuse to classify a pattern for the classifier/learner. This would force the classifier/learner to make an "educated" guess of the correct classification. It is conceivable that after being certain that the system has enough information to classify all patterns, the user can switch the system into non–interactive mode. Before the system can enter this mode, the system must have seen enough positive examples to generalize to the correct node in the generalization hierarchy. If the system is in non–interactive mode, it can use the closed world assumption and classify everything that does not match the S–set as negative.

Although this approach is very simple, the example implementation of the SVS algorithm uses two extra *cover sets* to avoid requiring that the user realizes when sufficient positive examples are presented to generalize to the correct node in the version space.

5.3 Description of the Symmetric Version Space Algorithm

As section 4 explains, the most specific pattern in the representation language that matches a given set of strings can be readily computed. However, the most general patterns that do not match a set of strings are harder to compute. The following paragraphs describe how the most general patterns (i.e. G–set) can be computed. However, even without disjunctions, the G–set grows rapidly.

The SVS algorithm maintains two *cover sets*, one for the positive and the second one for all negative examples. The *cover sets* are the most specific concepts that match the positive and negative examples respectively. In this respect, the *cover sets* are similar to the S–set of the Candidate Elimination algorithm. However, if an example matches either the positive or negative *cover set*, it does not follow that it is classified as a positive or negative example respectively. In contrast, the Candidate Elimination algorithm classifies an example that matches the S–set always as positive.

The SVS algorithm uses a single method to update the concept model, independently of whether the example is a positive or negative. The Candidate Elimination algorithm uses different update algorithms for positive and negative examples.

The update algorithm computes the most specific concept for the previous *cover set* and the new example. Therefore, it is quite possible that the positive and negative *cover sets* overlap. In fact, if the concept is a normal concept (i.e. it is exactly those items that match a specific description), the negative *cover set* will be generalized to the most general concept, if sufficient negative examples are provided. On the other hand, if the concept is a *complementary* concept (i.e. everything with the exception of those items that match a specific description), the positive *cover set* is most general after sufficient examples.

The classifier algorithm has to consider four different cases, when trying to classify a new example:

1. The new example neither matches the positive nor the negative *cover set*. In this case, there is not enough information to classify the new example. The new example is passed on to the oracle, in order to find out the correct classification. If the example is positive, call the update algorithm with the new examples and the positive *cover set*, else with the negative *cover set*.

2. The new example matches the positive as well as the negative *cover set*. The algorithm uses a similarity metric to determine the "goodness" of the match with the different *cover sets*. The example is classified as positive or negative, depending on which *cover set* yields the best match.

3. There is a match of the new example with the positive *cover set* and no match with the negative *cover set*. The SVS algorithm calls the routine to update the *cover sets* with the new example and the negative *cover set*, yielding the most specific description that matches them both. If the match with the provisional *cover set* is better than the match with the positive *cover set*, the new example is passed on to the oracle for correct classification. Otherwise it is classified as positive. If the oracle classifies the new example as negative, the negative *cover set* is replaced by the provisional *cover set*.

4. The new example matches the negative *cover set* and not the positive *cover set*. This case is equivalent to the previous case, with the roles of the positive and negative *cover set* reversed.

The behavior of the SVS algorithm in case three (and four with the roles of positive and negative reversed) requires some more explanation. In case three, one could be inclined to classify the example as positive. This classification, however, is possibly inappropriate, if the negative *cover set* does not match the example, because not sufficient negative examples were represented so far. Therefore, the SVS algorithm must ask the oracle for the correct classification. If the oracle classifies the example as negative, the negative *cover set* must be updated to include the example. In that case, the new negative as well as the positive *cover set* will match the example. On the other hand, in case one, the algorithm chooses the best match, if both *cover sets* match the example. Therefore, if the positive *cover set* yields a better match than the new negative one, the example will be classified as positive, which is in contradiction to the classification given by the oracle. Therefore, the SVS algorithm computes the *cover set* that would result from a negative classification. Only if there is a better match with the negative *cover set*, the example will be passed to the oracle. Otherwise, it will automatically be classified as positive.

Another problem occurs when in case three, the example is passed on to the oracle and classified as positive. In that case, neither *cover set* is updated, which means that the algorithm does not learn. If the same example is presented again, it will be passed on to the oracle again for classification. Although there are a number of different solutions to this problem, I chose to maintain two extra *cover sets*. If the example is classified as positive by the oracle, the positive extra *cover set* is updated to include this new example.

By using a *cover set*, the SVS algorithm avoids possibly asking about all possible examples. However, there is no requirement that the representation language for the *cover sets* and the extra *cover sets* are equivalent. For example, in order to restrict generalizations, the example implementation allowed disjunctions of up to 16 terms.

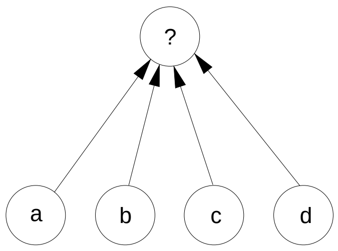
5.4 Candidate Elimination and SVS

Since the SVS algorithm learns normal as well as *complementary* concepts, it is able to learn more concepts than the Candidate Elimination algorithm in the same version space. Figure 9 compares the performance of the SVS algorithm and the Candidate Elimination algorithm on a simple example.

A more extensive example is described in tables 8, 7, and 6, it shows the performance of the SVS algorithm on three examples of lower case character concepts for limited disjunctions with at most two terms. The Candidate Elimination algorithm learns these concepts as described in tables 5, 4, 3.

As can be seen in table 6, the SVS algorithm requires more examples than the best case performance of the Candidate Elimination algorithm, which learned the correct concept after only three examples. The reasons for the extra questions is that the SVS algorithm can represent more concepts than Candidate Elimination. Although the positive *cover set* is generalized to the correct concept after three example, the algorithm requires the remaining examples to rule out the *complementary* concepts such as everything but the character x or everything but $x \cup y$.

The example in table 7 is a trace of the SVS algorithm learning a disjunction of two terms. As in the previous example, the best case performance of the Candidate Elimination algorithm is superior to the SVS algorithm. However, the Candidate Elimination algorithm will possibly ask about all lower case



Candidate Elimination

	Concept	S set	G set
Specific	a b c d	{a} {b} {c} {d}	{a} {b} {c} {d}
General	? =avbvcvd	{?}	{?}

Symmetic Version Space

	Concept	pos. Cover Set	neg. Cover Set
Specific	a b c d	{a} {b} {c} {d}	{?} {?} {?} {?}
General	? =avbvcvd	{?}	{nil}
Com – ple – ment– ary	? \ {a} = b v c v d	{?}	{a}
	? \ {b} = a v c v d	{?}	{b}
	? \ {c} = a v b v d	{?}	{c}
	? \ {d} = a v b v c	{?}	{d}

Figure 9: Comparison between SVS and Candidate Elimination

Example	Class.	ask User	pos. C	neg. C	pos. EC	neg. EC
a	$+$	yes	**a**	*nil*	*nil*	*nil*
b	$+$	yes	$\mathbf{a} \cup \mathbf{b}$	*nil*	*nil*	*nil*
c	$+$	yes	**?**	*nil*	*nil*	*nil*
d	$+$	yes	?	*nil*	*nil*	**d**
e	$+$	yes	?	*nil*	*nil*	$\mathbf{d} \cup \mathbf{e}$
f	$+$	yes	?	*nil*	*nil*	**?**
g	$+$	no	?	*nil*	*nil*	?
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
z	$+$	no	?	*nil*	*nil*	?

C = Cover set,EC = Extra cover set

Table 6: Trace of SVS algorithm for Concept ?

Example	Class.	ask User	pos. C	neg. C	pos. EC	neg. EC
a	$+$	yes	**a**	*nil*	*nil*	*nil*
b	$+$	yes	$\mathbf{a} \cup \mathbf{b}$	*nil*	*nil*	*nil*
c	$-$	yes	$a \cup b$	**c**	*nil*	*nil*
d	$-$	yes	$a \cup b$	$\mathbf{c} \cup \mathbf{d}$	*nil*	*nil*
e	$-$	yes	$a \cup b$	**?**	*nil*	*nil*
f	$-$	yes	$a \cup b$	?	**f**	*nil*
g	$-$	yes	$a \cup b$	?	$\mathbf{f} \cup \mathbf{g}$	*nil*
h	$-$	yes	$a \cup b$	?	**?**	*nil*
i	$-$	no	$a \cup b$	?	?	*nil*
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
z	$-$	no	$a \cup b$	?	?	*nil*

C = Cover set,EC = Extra cover set

Table 7: Trace of SVS algorithm for Concept $a \cup b$

characters. The worst case performance of the SVS algorithm is to ask about eight examples. This improvement is even more significant when trying to learn string patterns, because there is an infinite number of strings.

One disadvantage of the SVS algorithm is that since it will only ask about eight examples, the positive examples must be presented as one of the first examples. So the SVS algorithm in this example is unable to learn the concept $a \cup z$ given this presentation. However, there is no requirement that the generalization algorithm for the *cover sets* and the extra *cover sets* are equivalent. In a domain, where the positive examples are not presented as some of the first examples, a different algorithm that postpones generalization longer can be applied. The main advantage is that the SVS algorithm separates the two problems and allows to control these two aspects separately.

Example	Class.	ask User	pos. C	neg. C	pos. EC	neg. EC
a	$+$	yes	**a**	*nil*	*nil*	*nil*
b	$-$	yes	a	**b**	*nil*	*nil*
c	$-$	yes	a	$\mathbf{b} \cup \mathbf{c}$	*nil*	*nil*
d	$-$	yes	a	**?**	*nil*	*nil*
e	$-$	yes	a	?	**e**	*nil*
f	$-$	yes	a	?	$\mathbf{e} \cup \mathbf{f}$	*nil*
g	$-$	yes	a	?	**?**	*nil*
h	$-$	no	a	?	?	*nil*
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
z	$-$	no	a	?	?	*nil*

C = Cover set, EC = Extra cover set

Table 8: Trace of SVS algorithm for Concept a

Table 8 shows the major advantage of the SVS algorithm. Instead of asking about all lower case characters, the SVS algorithm learns the correct concept after only seven examples. The number of examples is dependent on the generalization algorithm used for the extra *cover sets*. The separation of these two algorithms allows easy control of the worst case performance of the SVS algorithm. In our experiments, the update algorithm described in section 4 is used to compute the extra *cover sets*. But instead of limiting the disjunctions to three elements, the extra *cover sets* allow disjunctions of up to 16 terms.

6 Examples

7 Discussion and Related Work

1. A^q.
2. INBFA.
3. Clustering Algorithms (CLASS-IT) are not interactive. A^q is not incremental.
4. IBFA Algorithm by Smith and Rosenbloom [**?**].
5. An interactive learning algorithm can not backtrack from over generalization. On the other hand, a least commitment algorithm can possible involve asking about all possible examples. This will be explained in section 5.
6. Kurt van Lehn learns context free grammars. However, method exponential in the length of strings [VB87]. Van Lehn and Ball designed an

algorithm that learns context free grammars from example using a version space approach. They showed that for context free grammars, the boundary sets are infinite. Faced with a similar problem, Van Lehn and Ball showed that by using a reducedness criteria, only a finite number of reduced grammars can construct a given set of strings. From this finite set, the algorithm eliminates all grammars that generate any of the negative examples.

Since regular languages are a proper subset of context free grammars, this approach is a possible approach to learning strings by example. This paper chooses a different approach for the following two reasons:

- The number of grammars that possible constructed a given set of strings grows exponentially with the length of the string. This made the algorithm unfeasible in the interactive domain.
- Van Lehn and Ball were interested in all possible concepts that are consistent with a given presentation. As mentioned in the description of the learning model, the algorithm wants to generate one or at least only a few concepts that are justifiable in the domain.

7. CLASS–IT.

8. Dan Moh's Thesis ??? Does not learn sequences of patterns. Stores only exception list of negative examples. Is not interactive. Does only learn zero or more characters of a given character class.

8 Conclusion

1. This paper describes a fast and efficient algorithm to learn string concepts.
2. It is fast and efficient, and useful in a variety of domains.
3. It is based on a very simple user interaction model.
4. The Version Space Approach can be used in a variety of situations. The performance of the candidate elimination algorithm is very dependent of the version space graph.
5. Although candidate elimination algorithm can learn limited disjunctions, the shape of the graph means that the CE algorithm can possibly ask about all possible examples.
6. Can not learn relations such as five lower case characters followed by three upper case chars. These concepts seem not useful, because normally commands do not allow you to specify these concepts.
7. Similarity metric can be improved.

References

[GN87] Michael R. Genesereth and Nils J. Nilson. *Logical Foundations of Artificial Intelligence*, chapter 7.3, pages 170–174. Morgan Kaufmann, 1987. Induction, how to factor a VS.

[Gol78] E. M. Gold. Complexity of automaton identification from given data. *Information and Control*, 37:302–320, 1978.

[Hir75] D. S. Hirschberg. A linear space algorithm for computing maximal common subsequences. *Communications of the ACM*, 18(6):341–343, 1975.

[Mit77] Tom M. Mitchell. Version space: A candidate elimination algorithm approach to rule learning. In *Proceedings of the 5th International Conference of on Artificial Intelligence*, pages 305–310, 1977.

[Nix83] R. Nix. *Editing from Examples.* PhD thesis, Yale University, 1983. Introduces gap patterns.

[VB87] Kurt VanLehn and William Ball. A version space approach to learning context–free grammars. *Machine Learning*, 2(1):39–74, 1987.