A Symmetric Version Space Algorithm for Learning Disjunctive String Concepts

Jacky Baltes
Knowledge Science Institute
The University of Calgary
2500 University Drive NW
Calgary, Alberta T2N 1N4, Canada
phone: (403) 220 5112
email: baltes@cpsc.ucalgary.ca

November 14, 1997

Abstract

The paper describes an algorithm to learn disjunctive string patterns from examples. An implementation of the algorithm learns concepts such as all C source files (`*.c or *.h`) in the UNIX domain. The algorithm is designed for an interactive learning model in which the number of questions to the user are minimized. Furthermore, only simple questions to the user are admissible. The representation language is restricted to a subset of regular expressions in order to reduce the complexity. The patterns in the concept grammar are extensions of Nix's gap patterns (see [Nix83]) and Mo's annotated gap patterns [Mo90]. To learn sequences of patterns, the strings are broken up into *units*. The algorithm assumes that the first example is a good representative of the concept, that is it contains all optional *units*. Even for limited disjunctions, the *G*–set of Mitchell's Candidate Elimination algorithm is infinite for domains that contain an infinite number of elements. A symmetric version space (SVS) algorithm using *cover sets* of the positive and negative examples computes only the most specific description that matches a set of examples and can therefore be used to learn in version spaces that make it impossible or infeasible to compute the most general descriptions. Two extra *cover sets* are maintained to avoid asking the user about all possible examples, a problem that arises when the number of terms in the concept is less than the maximum number of terms in a disjunction. The correctness of the SVS algorithm in version spaces with minimal generalization hierarchies, so called *k-disjunctive* version spaces, is shown. Furthermore, it is argued that an extended version of the SVS algorithm solves the *swapping* and *splitting* problem. The last result of the paper establishes that though the size of the version space grows exponentially, the sample size complexity increases only linearly with the maximum length of *units*.

Contents

1 Introduction **3**

2 The Interactive Learning Paradigm **4**

3 Representation Language **4**
3.1 Restrictions on the Patterns 5
3.2 Generalization Hierarchy 6
3.3 Sequences of Patterns 6

4 Updating the Cover Sets **6**
4.1 Adjusting the Units of a String 7
4.2 The Similarity Metric 8
4.3 The Generalize Algorithm 8

5 Symmetric Version Space **9**
5.1 Disjunctive Concepts 9
5.2 Limited Disjunctions 10
5.3 Description of the SVS Algorithm 11
5.4 Extra Cover Sets 13
5.5 Candidate Elimination and SVS 13

6 Correctness of the SVS Algorithm **14**
6.1 Errors in the SVS Algorithm 14
6.2 Discussion of the Correctness 17
6.3 Correctness in Disjunctive Version Spaces 17

7 Analysis **18**
7.1 Size of the Version Space 18
7.2 Example Complexity 19

8 Empirical Evaluation **21**

9 Conclusion **23**

10 Acknowledgements **23**

A Complete Example Trace **25**

1 Introduction

To develop intelligent agents for future applications, it is important to be able to learn repetitive tasks involving strings. Maintaining a file system and organizing mail messages or news articles are examples of such simple, but highly repetitive tasks. These tasks are tedious, frustrating, and error–prone, so they should be automated. It is possible to develop programs for these tasks in a general purpose programming or script language. Unfortunately, the organization of a file system is very dependent on a user's preferences. For example, some users prefer separate 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. Furthermore, the personal preferences of users are likely to change over time. This means that new programs must be developed.

The learning algorithm described in this paper is part of a research effort to design and implement a fast and easy to use system (Shell–Clerk) that allows the user to teach a computer repetitive operating system tasks by simply giving examples of the required procedures. Teaching by example is a very effective way to communicate the necessary task knowledge and seems to be particularly suited for this application. The results of the string learning algorithm can be transferred to other domains with strings as primary data, such as editors, text formatters, databases, and compilers. Learning a task from examples requires that the system learns to decide whether a command should be applied to given strings (representing file names, mail addresses, subjects of messages etc.) or not, based on the syntax of these strings. Although this decision can be improved using semantics, it would also require that the system knows the semantics of all intended domains.

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. Therefore, the system must have a learning module that learns disjunctive string concepts by example.

I assume an interactive learning model which is described in section 2. Learning disjunctive string concepts in this learning model requires that the following four problems are solved:

- Inducing complete regular expressions is too expensive [Gol78]. The symmetric version space (SVS) algorithm described in this paper restricts the representation language to a subset of regular languages, and thus allows it to be used in an interactive environment.

- Trivial disjunctions must be disallowed. In my implementation, a static limit of three terms seems to be adequate to learn common concepts in the UNIX domain.

- Mitchell's Candidate Elimination (CE) algorithm is inappropriate, because the *G*–set is infinite for limited disjunctions. The SVS algorithm computes the most specific description for all positive examples (called the positive *cover set*) and the most specific description for all negative examples (negative *cover set*).

- Even using limited disjunctions, any least commitment algorithm will possibly ask about all strings and there are infinitely many. Some method must be developed, that forces the algorithm to generalize in these situations. The SVS algorithm allows separate control over this problem. In the implementation of the Shell–Clerk, the SVS algorithm maintains two extra *cover sets*. These extra *cover sets* are used to learn concepts of positive or negative files that must be classified by the algorithm.

The complete learning algorithm consists of two separate modules and a similarity metric. The SVS algorithm which described in section 5 is the top level control module. It requires a method to update the *cover sets* (UCS algorithm, see section 4) and a similarity metric as subroutines.

Section 3 describes the representation language and therefore the set of learnable concepts. Section 4 describes the algorithm to compute the *cover sets*. Subsection 4.2 introduces a similarity metric used in the implementation. Section 5 describes the SVS algorithm used in this paper. Section 6 analyses under what assumptions the SVS algorithm learns the correct concept. The example complexity and the size of the version space are examined in section 7. Section 8 summarizes and evaluates the results obtained from training the SVS algorithm on some filename concepts. Section 9 draws conclusions and describes directions for future research.

2 The Interactive Learning Paradigm

If a user wants to teach the computer to copy all source files into a different directory and archive them, the computer must absorb two types of knowledge. First, it has to learn how to copy a given file into a different directory and archive it. This includes induction of loops and variables and is called task knowledge. Secondly, the computer has to learn what files must be transferred into a different directory. That is, the computer has to acquire the necessary concept knowledge to distinguish filenames belonging to the concept "source files" from other filenames. This paper focuses on the concept learning.

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 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.

The conceptual learning model (see figure 1) consists of five entities, the initiator, the task performer, the oracle, the learner/classifier, and an example source. The initiator recognizes the need to learn a concept that is necessary to execute a given task. It provides the learner/classifier with the first example of a concept, which will always be positive.

The task performer is the part of the system that is interacting with the environment. For example, it is the part of the system that issues commands to the UNIX shell to execute a task. In order to be able to execute the task, the task performer must be told the classification of elements that occur in the domain (for example filenames).

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 these new examples. If the classifier successfully recognizes these new 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, is it 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 now known and passed on to the task performer.

The interactive learning model restricts the set of concepts that can be learned. Every learning algorithm must generalize from past experiences to new examples. Otherwise, the learning algorithm will degrade into a simple database recording the classification of previously seen 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 and that lead to useful generalizations.

In the implementation, some of the conceptually different functions are combined. First, the user acts as initiator and as oracle. Secondly, the task performer also functions as example source. Other combinations are also possible. One interesting direction for further research is the combination of task performer and initiator. The task performer automatically recognizes the need to learn a concept and provides the first example.

3 Representation Language

The grammar for the representation language is given in table 1. The patterns in the concept grammar are extensions of Nix's gap patterns (see [Nix83]) and Mo's annotated gap patterns [Mo90]. 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.

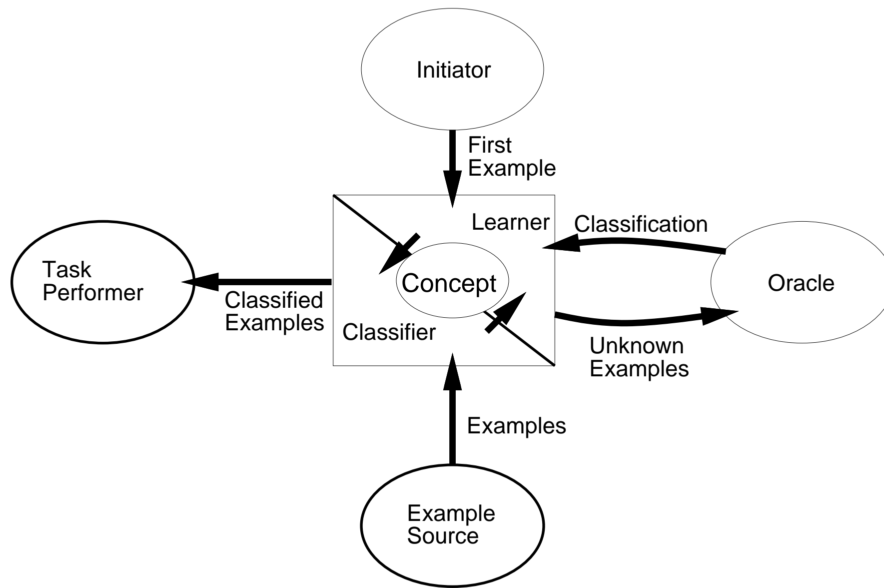


Figure 1: Interactive Learning Model

$<concept>$	$\Rightarrow$	$<unit_1><unit_2><unit_3>\ldots<unit_n>$ 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 $<pattern>$
$<pattern>$	$\Rightarrow$	$<charclass>\{\sigma^+ <charclass>\}^*$
$<charclass>$	$\Rightarrow$	$\varepsilon \mid <U> \mid <L> \mid <D> \mid <B> \mid <A> \mid <C> \mid <R> \mid <O> \mid <P> \mid <S> \mid <W>$
$<\alpha>$	$\Rightarrow$	$\alpha^1 \mid \alpha^*$ Metarule for $<U>, <L>, \ldots$
σ	$\Rightarrow$	Any terminal (e.g. $a, b, A, B, 0, 1, \ldots$)

Table 1: Grammar of the Concept Description Language

The choice of concept grammar is important, because a concept can be learned only if it is expressible in this language. 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, the greater the search space [Mit77].

The algorithm is designed to work in domains, such as operating system shells, that allow the user to specify only regular expressions in commands. However, learning regular languages is computationally expensive. Gold showed that the problem of inferring 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 examples is NP–hard.

3.1 Restrictions on the Patterns

The reason for the complexity is that there are many regular expressions that match a given set of strings. The representation language must be restricted. It will be described as restrictions on the deterministic finite automata (DFAs) accepting the regular expression.

First, transitions between nodes are allowed only on specific characters. This allows the construction only of DFAs that match specific strings such as *abc*. Therefore, the representation language also allows optional *shadow nodes* in the DFA, that are used to match exactly one character in a character class (see second DFA of figure 2). Loops are allowed only to accept zero or more characters of a specified character class (such as lower case or alpha–numeric characters). Therefore, all DFAs are sequences of the basic building blocks described in figure 2. The DFAs are

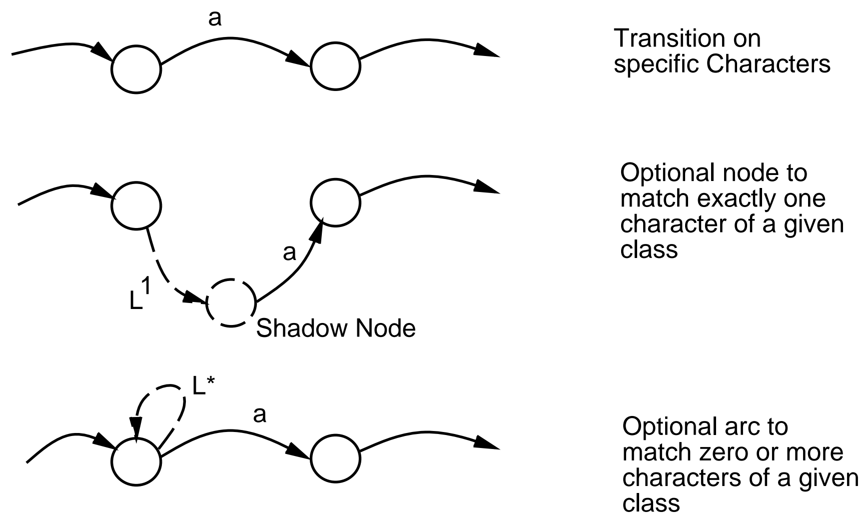


Figure 2: Building Blocks for restricted DFAs

equivalent to the patterns described in table 1.

3.2 Generalization Hierarchy

The algorithm uses a generalization hierarchy when merging different DFAs. The character set is broken up into seven different character classes: lower case, upper case, digits, punctuation characters, operators, whitespace, and special symbols. This subdivision is 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 (i.e. $L^1, O^1, \ldots$) , or zero or more characters of a given class (i.e. $U^*, D^*, \ldots$). This extension yields the complete generalization hierarchy which is shown in figure 3. The generalization of two strings is the lowest common ancestor of the two strings. For example, `"a"` and `"b"` will be generalized to exactly one lower case character D^1. The strings `"test"` and `"1"` are generalized to A^*.

3.3 Sequences of Patterns

Sequences of character classes cannot be learned without further enhancements because transitions between nodes are allowed on specific characters only. For example, the system cannot learn the concept at least one lower case character followed by at least one whitespace character. The SVS algorithm solves the problem by assuming that every concept is a sequence of a limited disjunctions of independent restricted DFAs (equivalent to patterns) as described in the previous subsections. Since this extension of the representation language is used to learn sequences of character classes, all strings are broken up into different character classes. These substrings are called *units* of the string. This imposes a high level structure on the concept. In 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]. 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 it contains all *units* of the concept. In other words, the first example must contain all optional elements. Subsection 4.1 describes how subsequent examples are adjusted to match the first example.

4 Updating the Cover Sets

Figure 4 is a pseudo code description of the UCS algorithm. It computes a most specific description in the representation language (see table 1) for a set of strings. For example, the UCS algorithm updates a limited disjunction of patterns such as “file1” ∪ “file2” ∪ “BackupD^*” with a new example (e.g. “work”), to compute the new *cover set*, in

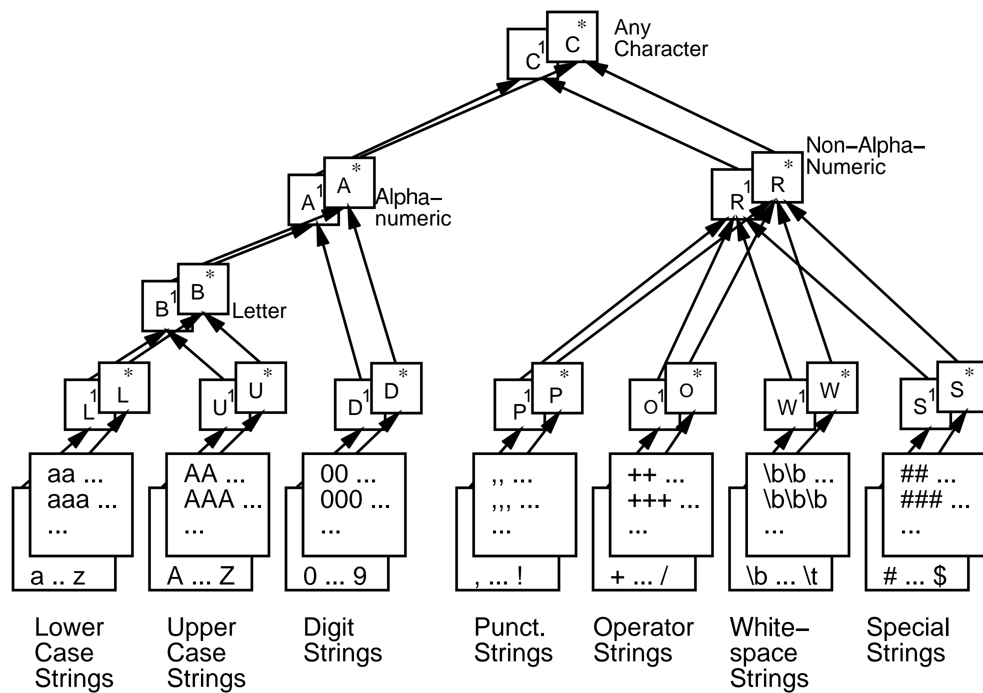


Figure 3: Generalization Hierarchy for Strings

this case "fileD^1" ∪ "work" ∪ "BackupD^*." This example shows that it is not necessarily the case that generalizing the new example with an element of the disjunction yields the most specific description. In the example, it is much more reasonable to generalize "file1" and "file2."

Three different tasks are required to update a *cover set*

- All strings are broken up into *units*. When handling a new example, it is necessary to adjust the *units* of the new example so that they produce the best match.
- Once the *units* are adjusted, a similarity metric is used to determine the two terms that are most similar and will thus yield the most specific generalization.
- After determining the most similar terms, these two terms must be generalized.

4.1 Adjusting the Units of a String

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. 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 example.

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" ""`.

The algorithm to adjust the *units* of a string to the length of the original example is a recursive method. First, the algorithm finds a unit that occurs in both strings. The strings are broken up at the matching *unit* and the algorithm is called recursively for the left and right substrings. If there is no matching *unit* in the strings, the *units* of the new string are concatenated or empty *units* are appended to the new string, so that the new string has the same number of *units* as the original string. Adjusting *units* requires that when trying to find matching *units*, the first and the last *unit* of the original string must be ignored because otherwise, it is possible that the algorithm has to concatenate *units* in the new string to match the empty *unit* which is impossible.

```
Name: UCS(item,cover−set)

Input:
  item: new string to add to the cover set
  cover−set: most specific description of a set of strings

Output:
  The most specific description that matches the item and the
  old cover set

IF size−of−disjunct(cover−set) < MAX−DISJUNCT−ALLOWED (=3) THEN
  return(cover−set + item)
ELSE
  min1,min2 := find−the−two−most−similar−items(cover−set + item)
  new−item := Generalize(min1,min2)
  del−cover−set := delete min1 and min2 from (cover−set + item)
  return(del−cover−set + new−item)
```

Figure 4: The UCS algorithm

Given the new example `"A" "test" "." "c"` and the prototype `"test" "." "ss"`, the adjust algorithm finds the *unit* `"."` and breaks the *units* up into a left (adjust(`"A" "test"`,`"test"`)) and right (adjust(`"c"`,`"ss"`)) subproblem. In the left subproblem, the matching *unit* `"test"` must be ignored because otherwise, the algorithm will try to justify `"A"` and `""`. Therefore, the new example is adjusted to the following *units*: `"Atest" "." "c"`.

4.2 The Similarity Metric

The similarity metric determines which two patterns should be generalized so as to not exceed the limit on the number of terms in the disjunction. When a new pattern is added to the disjunction, the most similar patterns must be generalized, which means that all combinations of generalizing two patterns must be tested. It is not necessarily the case that the new example and one previous term of the disjunction are combined.

The similarity metric used in the paper is based on the maximum common subsequence (MCS) of two strings. Since there is a simple string representation of patterns, the algorithm can also be used to compute the similarity between two patterns (e.g. L^*123 and L^*xyzA^*).

The similarity is the ratio of characters in the MCS against the total number of characters in the patterns. Instead of using the similarity metric directly, the implementation uses a difference measure for efficiency reasons. This ratio is expressed as a percentage and can be calculated using the following formula:

$$difference(p_1,p_2) = \frac{length(p_1)+length(p_2)-2*length(mcs(p_1,p_2))}{length(p_1)+length(p_2)} * 100$$

If two patterns are identical, the MCS of the two patterns is the pattern itself. Therefore, the difference value of two identical patterns will be 0. If the two patterns do not have a single character in common, the difference value will be 100. This difference measure can be easily converted into a similarity measure, since $similarity(p_1,p_2) = 100 - difference(p_1,p_2)$. Section 6 discusses problems associated with the ambiguity of the similarity metric.

4.3 The Generalize Algorithm

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 of two patterns. For example, the most specific generalization of the strings "test1.ss" and "test2.c" in the representation language is "test$D^1.L^*$." Figure 5 contains the pseudo code for generalizing two strings. Since the implementation of the UCS algorithm represents single character classes as a hash symbol followed by the lower case character for the character class (i.e. \$l = L^1) and an upper case character for zero or more characters (i.e. \$C = C^*), the same method can be used to generalize two patterns.

```
Name: Generalize(item1,item2)
Input:
  item1: first string or pattern to generalize
  item2: second string or pattern
Output:
  The most specific description that matches item1 and item2

  mcs = max-common-subsequence(item1,item2)
  gen := ""
  prev := START
  FOR each char c in mcs DO
    s1 := substring(item1,prev,c)
    s2 := substring(item2,prev,c)
    temp := lca-generalization-hierarchy(s1,s2)
   gen := gen + temp + c
   prev := c
  return(gen)
```

Figure 5: The Generalize algorithm

The generalization of the strings is based on the MCS of the two strings. The MCS is computed using Hirschberg's algorithm [Hir75]. A *subsequence* of a string s is a sequence of substrings $c = c_1 c_2 \ldots c_k$ that maintains the sequential ordering of $c_1, c_2, \ldots, c_k$ in s. That is, c_i occurs in c before c_j if and only if c_i occurs before c_j in s ($i < j$). A string c' is a *common subsequence* of strings $s_1, s_2, \ldots, s_m$ if and only if c' is a subsequence of all strings $s_1, s_2, \ldots, s_m$. A MCS is a *common subsequence* with maximal length. Hirschberg's algorithm computes the MCS of two strings in time $O(n^2)$ and space $O(n)$, where n is the length of the longest string.

Hirschberg's algorithm scans both strings from left to right and records the maximal possible length of the MCS if the current character is an element of the MCS. The length of the MCS must be bounded by the minimum length of the remainder of the two strings. For example, given the strings $s_1 =$ `"abcdef"` and $s_2 =$ `"bbbaaa"`, it is easy to see that the length of the MCS cannot be greater than $min(|abcdef|, |aaa|)$, if a is the first element of the MCS. This can be concluded without checking the remainder of string s_2. On the other hand, the MCS is bounded by 5, if b is the first element of the MCS.

5 Symmetric Version Space

The CE algorithm is not directly applicable to learning limited disjunctions of string patterns. As will be shown in subsection 5.2, the CE algorithm fails because the most general description that does not match the set of negative examples cannot be computed. The motivation for the SVS algorithm is that it is only necessary to compute the most specific description of a given set.

5.1 Disjunctive Concepts

Unlimited disjunctions pose an immediate problem, because they allow any learning algorithm to avoid generalization completely. The most specific concept description that matches all positive examples is the disjunction of all positive examples, the "trivial" disjunction. Therefore, the number of terms must be limited.

My experimental evidence suggests that a static limit of size three is adequate to learn commonly used concepts in the UNIX domain. Although a limit of three terms seems very restrictive, 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 plausible from a psychological point of view, which suggests that users do not organize their data using disjunctions with large numbers of terms. Although the implementation of the SVS algorithm limits the number of terms in a disjunction to three, for simplicity the examples in the paper assume a limit of size two.

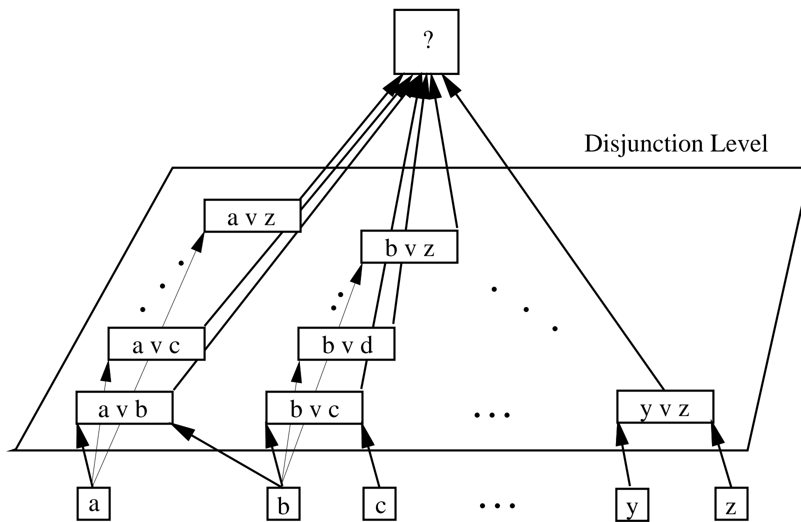


Figure 6: Version Space for Two–Disjunct Lowercase Characters

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

Table 2: Trace of Candidate Elimination for Concept ?

5.2 Limited Disjunctions

The CE 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 CE approach cannot be used, because the *G*–set is possibly infinite.

Figure 6 shows the version space for limited disjunctions of size two for lower case characters. In the string domain, the most general version space boundary is 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 and will be described using CE as an example. The problem is that when learning limited disjunctions, the algorithm will ask about all possible examples.

Using the version space described in figure 6, table 2 is a trace of the CE algorithm learning the concept of any lower case character. This is the best case for the CE algorithm. Only three examples are necessary to learn the correct concept.

Table 3 describes the performance of the CE 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 CE algorithm. The worst case for this presentation occurs when the correct concept is $a \cup z$. In that case, the CE algorithm asks about all lower case characters before learning the correct concept. In general, the CE 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.

A problem arises when CE learns a disjunction with less terms in the disjunction than the maximum number of terms allowed. Table 4 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. There is no distinction between best and worst case performance; the number of questions is independent of the ordering of the examples.

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

Table 3: Trace of Candidate Elimination for Concept $a \cup b$

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

Table 4: Trace of Candidate Elimination for Concept a

Every least commitment algorithm for the interactive learning paradigm is faced with this problem. Therefore, we must have a mechanism to reduce the questions to the oracle. A number of different mechanisms are possible and should be seen independently of the learning algorithm. The implementation of the SVS algorithm maintains two extra *cover sets* as will be described in subsection 5.4.

5.3 Description of the SVS Algorithm

Figure 7 is a slightly simplified pseudo code implementation of the SVS algorithm. Figure 7 ignores the extra *cover sets* (See subsection 5.4) that are used to limit the questions to the oracle. However, the implementation of extra *cover sets* is straightforward.

When adding a new example to a *cover set*, the UCS algorithm computes the most specific concept for the previous *cover set* and the new example. Therefore, it is possible that the positive and negative *cover sets* overlap. In fact, if the concept is a *direct* 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 after sufficient negative examples. On the other hand, if the concept is an *indirect* concept (i.e. everything with the exception of those items that match a specific description), the positive *cover set* is most general, if sufficient positive examples are provided.

The behavior of the SVS algorithm in the case where only one cover set matches a new string (See case 3 and 4 in figure 7 requires some explanation. For example, if there is only a match with the positive *cover set*, one could be inclined to classify the example as positive. This classification, however, may be inappropriate, if the negative *cover set* does not match the example because insufficient negative examples were presented 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 new 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 of the SVS-Classify algorithm in figure 7, 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. The rational is that generalizations cannot avoid possibly false classifications, however, the SVS algorithm tries to classify all examples consistently. An in depth discussion of the correctness of the SVS algorithm is presented in section 6. Examples of the SVS algorithm when learning the concepts "any character" and "a or b" in the $2-disjunctive$ lower case character domain are given in table 5 and in table 6 respectively.

```
Name: SVS-Algorithm(raw-item,pos-cover,neg-cover,first-example)
Input:
  raw-item: new string to learn/classify
  pos-cover: most specific description of all positive examples
  neg-cover: most specific description of all negative examples
  first-example: prototype of this concept
Output:
  The classification(positive or negative) and the updated
  positive and negative cover sets

   item := adjust-units(raw-item,first-example)
   IF SVS-Classify(item,pos-cover,neg-cover) = POSITIVE THEN
     return(POSITIVE,pos-cover,neg-cover)
   ELSE IF SVS-Classify(item,pos-cover,neg-cover) = NEGATIVE THEN
     return(NEGATIVE,pos-cover,neg-cover)
   ELSE
     class := Ask-User(item)
     IF class := POSITIVE THEN
       new-pos-cover := UCS(item,pos-cover)
       return(POSITIVE,new-pos-cover,neg-cover)
     ELSE
       new-neg-cover := UCS(item,neg-cover)
       return(NEGATIVE,pos-cover,new-neg-cover)
```

```
Name: SVS-Classify(item,pos-cover,neg-cover)
Input:
  item: string to classify
  pos-cover: most specific description of all positive examples
  neg-cover: most sepcific description of all negative examples
Output:
  The classification for the item (POSITIVE,NEGATIVE, UNKNOWN)

IF (NOT match(item,pos-cover)) AND (NOT match(item,neg-cover))          /* Case 1 */
  return(UNKNOWN)
ELSE IF match(item,pos-cover) AND match(item,neg-cover) THEN            /* Case 2 */
  IF difference(item,pos-cover) <= difference(item,neg-cover) THEN
    return(POSITIVE)
  ELSE
    return(NEGATIVE)
ELSE IF match(item,pos-cover) AND (NOT match(item,neg-cover)) THEN      /* Case 3 */
  new-neg-cover = UCS(item,neg-cover)
  IF difference(item,pos-cover) <= difference(item,new-neg-cover) THEN
    return(POSITIVE)
  ELSE
    return(UNKNOWN)
ELSE IF (NOT match(item,pos-cover)) AND match(item,neg-cover) THEN      /* Case 4 */
  new-pos-cover = UCS(item,pos-cover)
  IF difference(item,new-pos-cover) <= difference(item,neg-cover) THEN
    return(UNKNOWN)
  ELSE
    return(NEGATIVE)
```

Figure 7: The SVS Algorithm

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

C = Cover set,EC = Extra cover set

Table 5: Trace of SVS algorithm for Concept ?

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

C = Cover set,EC = Extra cover set

Table 6: Trace of SVS algorithm for Concept $a\cup b$

5.4 Extra Cover Sets

One problem with the algorithm as described in figure 7 is that in some cases no *cover set* is updated. In table 7 e, f, and g are examples of this problem. The SVS algorithm asks the oracle in these cases, to rule out *indirect* concepts such as "everything but $a\cup e$." Since e is, however, a negative example, and since the negative *cover set* already covers e, no *cover set* is updated. So if e is presented again as an example, the SVS algorithm will ask the oracle again.

One solution for this problem is to maintain a list of examples that were already handled in this way. Instead, I chose to maintain two extra *cover sets*, since in this way also the problem of asking about all possible examples can be solved. The generalization method for the extra *cover sets* may not be the same as the one for the *cover sets*. The SVS algorithm allows control of these two aspects of the learning algorithm separately.

When classifying a new example, the extra *cover sets* are tested in cases, where an example matches only one of the *cover sets*, but there is possibly a better match with the non–matching one. The corresponding extra *cover set* is tested, and the example is classified as an example of the matching *cover set* if the new example matches the corresponding *cover set*. Otherwise, the new example is passed on to the user. For example, in table 5 the positive extra cover set is updated, ruling out the *indirect* concepts such as "all lower case characters except d." In table 6 and 7, the negative extra cover set rules out "a or some other character" as possible concepts.

5.5 Candidate Elimination and SVS

An example is given in table 7. It shows the performance of the SVS algorithm when learning the concept a. The SVS algorithm requires more examples than the best case performance of the CE algorithm. The reason for the extra questions is that the SVS algorithm can represent more concepts than the CE algorithm, since the SVS algorithm learns

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

C = Cover set, EC = Extra cover set

Table 7: Trace of SVS algorithm for Concept *a*

direct as well as *indirect* concepts. The SVS algorithm requires the remaining examples to rule out *indirect* concepts.

Table 7 also shows how the SVS algorithm handles the problem of asking about every example. 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 my experiments, the update algorithm described in section 4 is used to compute the extra *cover sets*.

6 Correctness of the SVS Algorithm

This section discusses the correctness of the SVS algorithm. It gives an informal argument that under the assumptions that special cases occur "early" in the presentation and that the most specific description of a set of instances is unique, an extended version of the SVS algorithm learns the correct concept.

6.1 Errors in the SVS Algorithm

There are four errors that lead to incorrect classifications of the SVS algorithm:

1. If a new example matches only one *cover set*, it is passed on to the oracle only if the non–matching *cover set* when augmented with the new example yields possibly a better match. Otherwise, it is classified as a member of the matching *cover set*. What happens if future examples generalize the matching *cover set* to be more general than the non–matching one? In that case, the new example could have been a member of the non–matching *cover set*. This problem is called the *swapping* problem.

2. It is also possible that by adding a new example a *cover set* is generalized so that the new concept is more specific than the opposite *cover set*, but also covers some examples of the opposite *cover set*. This problem is called the *splitting* problem.

3. The extra *cover sets* will possibly over–generalize.

4. The most specific description for a given set of strings is not unique. This causes a problem for the SVS algorithm because it selects one description at random.

The first and second problems are the most serious, because the SVS algorithm will classify examples not only incorrectly, but also inconsistently. Figure 8 shows one possible version space for a domain with eight instances $\{i1, i2, \ldots, i8\}$.

Two examples of the *swapping* problem are given in table 8. The table ignores the extra *cover sets* for simplicity. Item *i*3 is classified inconsistently in row 3, 5, and 7. First, the positive *cover set* is more general, but the addition of *i*4 to the negative *cover set* makes the positive *cover set* more specific than the negative one. Adding *i*8 to the positive *cover set* reverses this situation again.

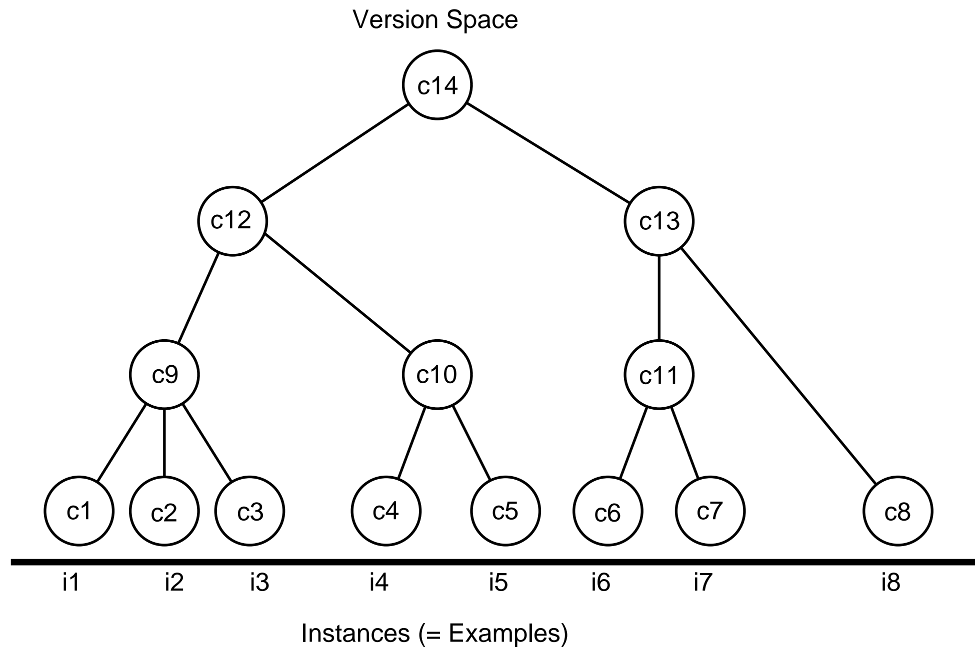


Figure 8: A Sample Version Space

Number	Example	Class.	ask User	pos. C	neg. C
1	$i1$	$+$	yes	$\underline{c1}$	*nil*
2	$i2$	$+$	yes	$\underline{c9}$	*nil*
3	$i3$	$-$	yes	$c9$	$\underline{c3}$
4	$i4$	$-$	yes	$c9$	$\underline{c12}$
5	$i3$	$+$	no	$c9$	$c12$
6	$i8$	$+$	yes	$\underline{c14}$	$c12$
7	$i3$	$-$	no	$c14$	$c12$

C = Cover set

Table 8: The *Swapping* Problem

Number	Example	Class.	ask User	pos. C	neg. C
1	$i1$	+	yes	$\underline{c1}$	*nil*
2	$i2$	+	yes	$\underline{c9}$	*nil*
3	$i4$	−	yes	$c9$	$\underline{c4}$
4	$i3$	+	no	$c9$	$c4$
5	$i8$	+	yes	$\underline{c14}$	$c4$
6	$i3$	−	yes	$c14$	$c12$

C = Cover set

Table 9: The *Splitting* Problem

Table 9 shows a trace of the SVS algorithm in which the classification of examples is inconsistent because of the *splitting* problem. Example $i3$ is classified as positive (Number 4) and as negative (Number 6). The addition of $i3$ to the negative *cover set* (Number 5) splits the positive one. Therefore, the classification of $i1$ and $i2$ also changes when $i8$ is presented as a positive example and the positive *cover set* is updated to the most general description ($c14$, see number 5).

However, the *swapping* problem and the *splitting* problem are similar in that the SVS algorithm misinterprets an instance, because it allows the user to classify too many examples. The representation language for the SVS algorithm allows only *direct* concepts (i.e. nodes in the version space) or *indirect* concepts (i.e. all instances with the exception of those matching one node in the version space). So the SVS algorithm asks too many questions because the user can specify a concept such as "$i1$ and $i2$, but not $i3$ and $i4$" (Table 8) or "everything but $i3$ and $i4$ (but including $i1$ and $i2$)" (Table 9) which are not expressible as *direct* or *indirect* concepts. After classifying $i1$ and $i8$ as positive examples and $i4$ as a negative example, the only consistent concepts in the representation language are "everything but $i4$" and "everything but $c10$ (i.e. not $i4$ and not $i5$)." The SVS algorithm correctly learns concepts in the representation language, but by passing too many examples on to the oracle, it allows the oracle to specify concepts inexpressible in the representation language. It would not be considered an "error" of the CE algorithm to classify $i3$ as positive after the presentation of $i1$ and $i2$ as positive examples in the version space given in figure 8. However, it would be annoying if the CE algorithm allows the user to inconsistently classify $i3$ as a negative example. The *swapping* and *splitting* problems occur only when one *cover set* is a totally covered by its opposite. If only a few instances match both *cover sets*, these problems will not arise. On the other hand, one of the *cover sets* will be generalized to the most general description after sufficient examples. Therefore, after a certain number of examples, the two *cover sets* completely overlap.

Two simple extensions to the described SVS algorithm allow us to avoid unnecessary questions and therefore the *swapping* and *splitting* problem. First, insure that once a *cover set* is a superset of the opposite *cover set*, it is generalized to the most general description. This eliminates the *swapping* problem. For example, once it is known that $i1$ and $i4$ are positive examples, and that $i2$ is a negative example, the only consistent concept in the representation language is "everything but $i2$." The second extension is to maintain a list of examples, called *ground-list*, that were used to construct the *cover set*. Note that the length of this list is bounded by the length of the maximum chain in the version space. Add an element to a *cover set* only if the resulting *cover set* is either not a subset of the opposite cover set or it does not cover anything in the *ground-list* of the opposite *cover set*. This guarantees that generalizing a *cover set* does not split the opposite *cover set*.

These two extensions to the SVS algorithm are ignored in the implementation, because the problem only occurs in narrow version spaces where concepts overlap. In the string domain, there are many possible specializations for each concept. This means that the version space is very broad and concepts seldom overlap until one *cover set* is generalized to match all strings. This intuition is supported through an empirical evaluation (Section 8) in which the problems never occurred.

Clearly, the over–generalization of extra *cover sets* (problem three) prevents the SVS algorithm from learning some concepts expressible in the concept description language by some presentations. For example, using the presentation in table 7, it is easy to see that concepts such as $a \cup h, \ldots, a \cup z$ cannot be learned. On the other hand, the SVS algorithm will possibly ask about an infinite number of strings without generalization of the extra *cover sets*. The bias using the extra *cover sets* is that special cases appear "early" in the presentation. In other words, the assumption is that there are enough examples in the presentation that will generalize the more specific *cover set* to the correct node, before the opposite extra *cover set* is generalized to ignore these examples. In the given scenario, it means that special cases occur

within the first seven examples. Furthermore, the SVS separates this problem from the general learning algorithm and it is easy to change the meaning of "early" by using different generalization techniques. Even when the assumption of "early" special cases does not hold, and thus there is no generalization in the extra *cover sets*, the SVS algorithm performs better than the CE algorithm because it can learn more concepts in the same version space.

The fourth problem is due to the simple similarity metric which is based on the MCS of two strings. Furthermore, the generalization of two strings is also based on their MCS. However, the MCS of two strings is not unique. For example, `ab` as well as `cd` are both MCSs of the strings `cdab` and `abcd`. Without additional knowledge, it is impossible to decide whether L^*abL^* or L^*cdL^* is the correct generalization. One simple heuristic that can improve the similarity metric and the generalization algorithm is to give more weight to prefixes and suffixes because they seem to be more commonly used in concepts than letters in the middle of a string.

6.2 Discussion of the Correctness

In the following section, problems three and four are ignored. One assumption is that there are enough examples in the presentation to generalize the more specific *cover set* to the correct node in the version space before the extra *cover set* will force the SVS algorithm to ignore those examples (i.e. special cases are "early"). In this case, the extra *cover sets* can be interpreted as lists of examples. The second assumption is that the SVS algorithm always computes the correct most specific description of a set of strings.

If it is known that the concept to be learned is a *direct* concept, the negative *cover set* can be initialized to the most general description. In this case, the SVS algorithm behaves very similar to the INBF algorithm by Smith and Rosenbloom [SR90] which is a learning algorithm for tree-structured version spaces. It can be proven for the INBF algorithm that the upper bound for the size of the boundary sets is polynomial in the number of examples.

Given a new example, the SVS algorithm tests whether the example matches the positive *cover set* or not. If it does match the positive *cover set*, the SVS algorithm will classify it as positive because it must yield a better match than the negative *cover set*. If the new example does not match the positive *cover set*, it will be passed on to the oracle, because adding the new example to the positive *cover set* will always yield a match as good as the match with the most general concept (i.e. the negative *cover set*). If the oracle classifies the new example as positive, it is added to the positive *cover set*, otherwise it is added to the negative extra *cover set*.

Therefore, the positive *cover set* is identical to the *S*–set of the CE and INBF algorithm and negative examples will be added to the negative extra *cover set*. The INBF algorithm uses the list of negative examples to delay the processing of these examples. Using the INBF algorithm, the *G*–set is updated only after an element is recognized as a near miss. This prevents the so called *fragmentation* of the *G*–set that occurs when a negative example is a far miss. Since there are a number of ways in which a far miss can be made more specific, every far miss will generate a number of elements of the new *G*–set. This means that the size of the *G*–set and therefore also the time complexity of the CE algorithm is exponential in the number of examples.

A similar argument shows that the reverse is true for *indirect* concepts, if the positive *cover set* is initialized to the most general description. Therefore, if one of the *cover sets* is generalized to the most general description, the SVS algorithm learns the concept correctly. The only restriction when generalizing *cover sets* is that the concept must be one node in the version space or the inverse of one node in the version space. The only ways in which this rule can be violated are by *swapping* or *splitting* an existing *cover set*. Since there are two extensions to the SVS algorithm that solve the *swapping* problem and the *splitting* problem, the SVS algorithm learns all concepts expressible in the representation language under the assumptions given at the beginning of this section.

6.3 Correctness in Disjunctive Version Spaces

An interesting subclass of version spaces, called *k–disjunctive* version spaces, are version spaces of limited disjunction (at most *k* terms) and a minimal generalization hierarchy. All elements in the domain are either specific elements or can be generalized to match any element. The version space shown in figure 6 is an example of a *2–disjunctive* version space. Apart from being of theoretical interest for the analysis, there are domains that have no syntactic generalization hierarchy associated with them. For example, there is no obvious further division into different classes of lower case characters without further semantic knowledge. In these domains, *k–disjunctive* version spaces naturally arise. Furthermore, every binary version space with *n* attributes can be interpreted as a 2^{n-1}–*disjunctive* version space, since there are exactly 2^n different elements in the domain. One advantage of *k–disjunctive* version spaces is that the second

assumption in the previous subsection is always satisfied. The most specific description of a set of instances is unique. The following proof is based on the assumption that special cases occur "early."

The correctness of the SVS algorithm can be proved in *disjunctive* version spaces. Given a *k–disjunctive* version space, each *cover set* is either a disjunction of at most k specific terms or matches everything in the domain. A presentation p is a sequence of examples, where each example is classified as positive or negative. To prove the correctness of the SVS algorithm, it has to be shown that given any concept c and any presentation p for c, the concept c' resulting from running the SVS algorithm on p is equal to c.

Every concept description consists of an expression for the positive and the negative *cover set*. One *cover set* is generalized to the most general description. The proof assumes that the concept c is a *direct* concept. The proof for *indirect* concepts is symmetric.

Under the assumption of a *direct* concept, it follows that the positive *cover set* P_c of concept c contains at most k elements, and the negative *cover set* N_c matches at least $k+1$ elements. It remains to show that all positive and all negative examples are correctly learned and classified.

This part shows that positive examples are learned correctly, or in other words that $P_c = P_{c'}$. Consider a new example x_p that is a positive example of c. If x_p has not appeared previously in p, it cannot match $P_{c'}$ because c is a *direct* concept. If $N_{c'}$ is generalized to the most general description, adding x_p to $P_{c'}$ will yield a better match than $N_{c'}$. If $N_{c'}$ is not most general, it can also not cover x_p, unless the oracle classified x_p incorrectly. Under the assumption that x_p is not covered by the negative extra *cover set*, x_p is passed on to the oracle in both cases. After the oracle classifies x_p as positive, it is added to $P_{c'}$. Therefore, it follows that $P_c = P_{c'}$ after all positive examples.

The next paragraph shows that the classification of positive examples is correct. If x_p has already been shown as an example, $P_{c'}$ will already match x_p. If $N_{c'}$ is most general, x_p will yield a better match with $P_{c'}$, and thus x_p is classified as positive. If $N_{c'}$ contains less than $k+1$ terms, x_p cannot be an element of $N_{c'}$ because otherwise x_p was classified inconsistently by the oracle. Furthermore, adding x_p to $N_{c'}$ can never yield a better match than the perfect match with $P_{c'}$. Therefore, x_p is classified as positive. Therefore, the SVS algorithm classifies positive examples correctly.

It is left to show that $N_c = N_{c'} = ?$. Let x_n be a negative example of c. x_n does not match $P_{c'}$. If x_n has not been shown previously in p, then it will either match $N_{c'}$ or not. If it does not match $N_{c'}$ it is passed on to the oracle and will be added to $N_{c'}$ after the classification. Therefore, it follows that $N_{c'}$ will be most general after sufficient examples. If $N_{c'}$ is already most general, then either there is possibly a better match with $P_{c'}$ or not. If $P_{c'}$ contains less than k terms, it will possibly yield a better match. x_n will be passed to the oracle and will be added to the negative extra *cover set*. If $P_{c'}$ is a disjunction of k terms, adding x_n to $P_{c'}$ will not yield a better match. Therefore, x_n will be classified as negative. The proof of correct classification of x_n is similar to the proof for correct positive classification.

This result and the observation, that binary version spaces can be represented as *k–disjunctive* version spaces, proofs that the SVS algorithm is powerful enough to learn any concept in binary version spaces.

7 Analysis

The following section analyses the version space size and example complexity of the SVS algorithm. The analysis yields exact results for *k–disjunctive* version spaces. For the version space generated by the representation language for string patterns, only the upper bounds are computed.

7.1 Size of the Version Space

In general, if there are n different elements in the domain, there are

$$\sum_{i=0}^{n} \binom{n}{i} = 2^n$$

different concepts. As mentioned previously, by restricting the representation language for concepts, a learning algorithm can reduce the complexity. This technique is equivalent to implementing an absolute bias. The version space is the set of all expressible concepts in the representation language. The following section computes the number of concepts that the SVS algorithm can learn in a given version space.

Since the CE algorithm is a bi–directional search for a node in the version space, it follows immediately that the CE algorithm can only learn as many different concepts as there are nodes in the version space.

The SVS algorithm learns additional concepts by allowing *indirect* concepts. An upper limit on the number of concepts learnable using the SVS algorithm is $2N_{VS}-1$, where N_{VS} is the size of the version space, because a concept is either a node in the version space or everything with the exception of a node. Note that the concept "everything with the exception of everything" (or in other words "nothing") cannot be learned because the first example is always a positive example. Figure 9 is an example of a version space in which the number of concepts of the SVS algorithm is equal to the upper bound. This estimate, however, is only an upper bound because the inverse of a node in the version space can again be a member of the version space. An example of this would be the version space resulting from deleting c and d from the version space given in figure 9. For example, everything but a is equivalent to b.

In a *k–disjunctive* version space, the size of the version space can be computed using the following formula:

$$\sum_{i=1}^{k}\binom{n}{i}+\sum_{i=0}^{k}\binom{n}{n-i}=2*\sum_{i=1}^{k}\binom{n}{i}+1$$

This formula is based on the observation that each concept is either a *direct* or an *indirect* concept, or it is the most general concept matching all n elements in the domain. The inverse of the most general concept cannot be learned because it is equivalent to the concept matching elements at all, which is in contradiction to the assumption that the first example is always positive.

An upper limit for the version space generated by string patterns can also be computed based on the observations above. Strings are broken up into *units*, so let n_u be the number of *units* in the first example and n_s the maximum length of any *unit* in the domain. The *units* are independent and each *unit* contains two limited disjunctions. Let k denote the maximum number of terms in a disjunction. Each element of the string can be either a specific character or one of the character classes. The total number of characters or patterns will be denoted n_c. It follows that the size of the version space must be bounded by

$$\Omega(n_u * n_c^{2kn_s})$$

7.2 Example Complexity

One important consideration in the design of the interactive learning model is to minimize the number of questions to the oracle required to learn a concept. This subsection will give an upper bound on the number of questions to the oracle. It is easy to see that after any classification of the oracle, exactly one of the *cover sets* or extra *cover sets* is updated. In the case that any pair of *cover sets* (e.g. positive and extra positive) are generalized to the most general description, all examples will be classified and no more examples will be given to the oracle. Furthermore, as mentioned previously, it is not necessary that the version spaces for the *cover sets* and the extra *cover sets* are identical. Therefore, an upper bound on the number of questions to the oracle is given by:

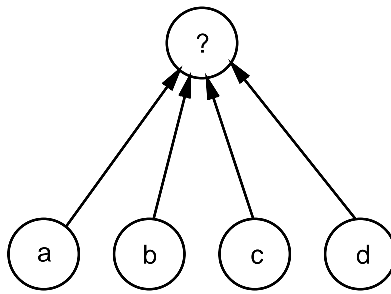
$$2*height(cover_set)+2*height(extra_cover_set)$$

where $height(x)$ is the length of the maximum chain from the most specific to the most general concept in version space x.

Due to the simple structure of the *k–disjunctive* version spaces, this upper bound can be made more specific. Since instances are added to an extra *cover set* only if the instance does match one *cover set*, but there is possibly a better match with the opposite *cover set*, it follows that if element x is added to the positive extra *cover set*, the positive *cover set* is more general than the negative *cover set*. Therefore, it follows that the positive *cover set* matches every element and that the negative *cover set* contains less than k terms. The situation is reversed for negative extra *cover sets*. One pair of *cover sets* will be generalized to the most general description and the opposite *cover set* is more specific than the most general description. The opposite extra *cover set* does not match any elements. It requires at most $k+1$ examples to update a *cover set* to the most general concept. Therefore, only one extra *cover set* is generalized. The maximum number of questions for *k–disjunctive* version spaces is therefore given by:

$$k+k-1+k=3k-1$$

Using a similar technique, an upper limit can be computed on the number of questions needed to learn disjunctive string patterns. Let n_u be the number of *units* in the first example and n_s the maximum length of any *unit* in the domain.



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 (Direct)	a b c d	{a} {b} {c} {d}	{?} {?} {?} {?}
General	? =avbvcvd	{?}	{nil}
Indirect	? \ {a} = b v c v d ? \ {b} = a v c v d ? \ {c} = a v b v d ? \ {d} = a v b v c	{?} {?} {?} {?}	{a} {b} {c} {d}

Figure 9: Size of the version space for the CE and SVS algorithm

The *units* are independent and each *unit* contains four *cover sets*. Let k denote the maximum number of terms in a *cover set*.

If a new example is classified by the oracle, at least one *cover set* in a *unit* is made more general. Therefore, the upper bound on the number of questions grows linearly with the number of *units* in the concept.

$$Complexity = n_u * max\{q | q = C(u_i), 1 \leq i \leq n\} \text{ where } C(u) \text{ is the complexity of learning unit } u.$$

If two patterns are generalized, at least one element of the pattern is made more general. In contrast to *k–disjunctive* version spaces, it is possible that extra *cover sets* overlap. So if l is the length of the maximum chain in the generalization hierarchy, then an upper bound on the example complexity is given by $4kln_un_s$.

Since every pattern that matches exactly one character can be converted to a pattern matching any number of characters, the length of the maximum chain in the generalization hierarchy shown in figure 3 is equal to six. So the maximum number of questions is equal to $72n_un_s$. The empirical evaluation showed that on average far less examples are required than the upper bound.

The example complexity of the SVS algorithm grows only linearly with the length of the *units*. This means that the SVS algorithm is well suited to an interactive domain.

8 Empirical Evaluation

Since the SVS algorithm was implemented as part of system to learn repetitive operating system tasks, it was tested on some common concepts in the UNIX domain. The presentation for the learning algorithm consisted of an alphabetically sorted fixed sequence of 96 file names that were taken from existing project directories. The complete test set is given in table 10. The file names represented Gnu Emacs backup files, C language source files, Chez Scheme source files, and script files. A complete example of the interaction with the Shell–Clerk is given in appendix A. The diversity and number of files ensured that the general *cover sets* were generalized to the most general concept description. The file names were presented in alphabetical order because the UNIX shell returns file names in this order and because the Shell–Clerk system does not allow the user to change the order of the examples. Ten example concepts were chosen and the learning algorithm was trained using this presentation. Tests one to seven are examples of *direct* concepts with a varying number of disjunctive terms. Test eight learns a sequence of character classes. The target concept is “everything with at least one character, followed by at least one digit, a period, and at least one more character.” Although character classes can distinguish only between exactly one and zero or more characters, the prototype `FTEST2.ss` restricts the concept to at least one character in a class. Otherwise, the *units* `"FTEST" "2" "." "ss"` of the new example are adjusted differently. Tests nine and ten are examples of *indirect* concepts. The prototype of the concept, the number of questions to the user, and the number of errors were recorded. Table 11 summarizes the results.

The errors in test nine were due to over–generalization of the extra *cover sets*. The SVS algorithm classified enough negative examples to generalize the negative extra *cover set* to match any string, before classifying the examples `NOT(*learn*)`. Instead, the SVS algorithm learned the concept `*`. Therefore, it is not surprising that the number of questions is identical to the first test. It is important to note that the number of questions asked does not mean that the first examples in the list are only classified by the user. Whether or not the user was asked to classify the example depends on how many similar examples are given previously.

In particular for the UNIX domain, although the SVS–algorithm limits the number of questions, the user still has to provide the correct classification for one third to one half of the examples. The question arises, if the representation language should be further restricted to rule out concepts that are not commonly used. One example is to rule out concepts based on single characters in a string, such as `*s*`. The answer to this question requires a more systematic study of a large number of users.

The analysis of *k–disjunctive* version spaces predicts that the SVS algorithm learns all possible concepts in domain with less than $2k+1$ elements. By applying this result to filenames in a directory, it follows that the SVS algorithm learns to classify any concept in directories with less than seven files ($k = 3$). Furthermore, it requires the correct classification of every element in the directory from the user. This prediction was verified using a directory containing files `a,b,...,f`.

#make-focus.ss#	#script.txt#	#script2.txt#	#script3.txt#
#test.ss#	CTEST.ss	DebMalloc.c	DebMalloc.c~
FTEST2.ss	adjust.ss	adjust.ss.CKP	adjust.ss~
backup.zoo	bak.ss	built-in_amiga.ss	built-in_sun.ss
built-in_sun.ss~	built-in_sun.ver2	clerk-state.ss	clerk.ss
clerk.ss~	command.ss	compile.ss	compile_sun.ss
concepts.ss	concepts.ss~	convertgif.ss	copyss.ss
csrc.ss	csub	csub-old	csub.c
csub.c~	csub.h	csub.h~	csub.o
debmalloc.c~	debmalloc.o	delgif.ss	dialogue-stuff.ss
dummy.ss	dummy2.ss	else.ss	fileio_amiga.ss
fileio_sun.ss	focus.ss	ftest.ss	generic
generic.c	getA3000.ss	getlatt.ss	getnews.ss
grep.ss	junk.ss	junk2.ss	learn-rule.ss
learning.ss	load_amiga.ss	load_sun.ss	load_sun.ss~
loadc_sun.ss	lsdir_amiga.c	lsdir_sun	ltest.ss
make-focus.ss	make-focus.ss~	make-task.ss	mycd
mycd.c	parameters.ss	print.ss	readfile.c
recdir.ss	rm.ss	rmgrep.ss	script.txt
script.txt~	script2.txt	script2.txt~	script3.txt
script3.txt~	stderr	test-session1.txt	test-spl.ss
test-spl.ss~	test.sh	test.ss	test2.ss
test2.ss~	test3.ss	test3.ss~	test4.ss
undo.ss	uudecode.ss	zoo2.ss	zoosrc.ss

Table 10: Set of Filenames in the Evaluation

Test	Concept	Prototype	Num. of Quest.	Reduction	Num. of Errors
1	?	generic	19	80.21%	0
2	test* or #test*	test.ss	32	66.66%	0
3	*(built-in or file-io)				
	(_sun or _amiga)*	built-in_amiga.ss	38	60.41%	0
4	*(amiga or sun)*	lsdir_amiga.ss	42	56.25%	0
5	*e*.ss	clerk.ss	41	57.29%	0
6	*(.c or .h or .ss)	adjust.ss	43	55.20%	0
7	#*# or *~ or *.CKP	adjust.ss~	46	52.08%	0
8	$C^{+}D^{+}.C^{+}$	FTEST2.ss	32	66.66%	0
9	NOT(*learn*)	adjust.ss	19	80.21%	3
10	NOT(C* or c*)	adjust.ss	26	72.91%	0

Table 11: Results of the Experiments (Max. 3 Terms for extra *Cover Sets*)

9 Conclusion

The paper introduces an interactive learning paradigm, which tries to minimize the number of questions and to simplify the types of questions to the oracle. The SVS algorithm learns concepts from a subset of regular expressions and allows limiting the number of questions to the user. The representation language is non–trivial and allows the expression of common concepts in the UNIX and other domains.

Although the version space approach is a very powerful technique, the CE algorithm and focusing algorithms [SR90] are not directly applicable to limited disjunctions because the most general boundary is infinite for domains with an infinite number of elements. The SVS algorithm overcomes this problem by not representing the most general boundary explicitly. Instead, the most specific boundaries (*cover sets*) for all positive as well as all negative examples are computed.

The SVS algorithm improves on previous work by Mo ([Mo90]) in a number of important aspects. The SVS algorithm learns *direct* as well as *indirect* concepts. Character classes can match one and zero or more characters. Sequences of patterns are learned by breaking the example up into *units*. In order to decide whether two patterns should be generalized or the sequence of patterns should be maintained, the algorithm uses the first example as a prototype for the concept. Mo's algorithm only computed the S-set and maintained the negative examples as a list of exceptions. Therefore, Mo's algorithm only generalizes positive examples. The SVS algorithm generalizes positive and negative examples.

The SVS algorithm allows separate control over the problem of asking about all possible examples. The problem arises if the number of terms in the concept is less than the maximum number of terms allowed. In the implementation two extra cover sets are maintained. This restricts the maximum number of questions to be linear in the maximum length of a chain between a specific string and the most general pattern description. In the generalization hierarchy described in this paper, this is equivalent to being linear in the length of the longest *unit* in a string.

Extensions to the SVS–algorithm are described that overcome the *swapping* and *splitting* problem. The extensions overcome these problems by detecting and avoiding the situations that can lead to those problems. Another approach is to detect problematic situations and change the representation language. For example, given the presentation in table 9, the SVS algorithm will detect that the classification of $i3$ in row 6 as negative splits the positive *cover set*. The positive *cover set* is then changed into a high level disjunction for $(i2,i3) \cup i8$. This technique will allow the SVS–algorithm to learn unlimited disjunctions. It is unclear, however, if this method will result in the generation of trivial disjunctions.

One direction of further research is to improve the communication with the teacher. Currently, the teacher only selects the prototype of a concept and classifies unknown examples. The system has to determine which parts of the string are relevant and which one are not. Since it has been shown in previous work that a more systematic communication with the teacher can greatly reduce the example complexity [RS88, Van87], I believe that the algorithm can be improved if the oracle is able to focus the learning algorithms attention on relevant parts of the string.

10 Acknowledgements

I wish to thank my supervisor, Bruce MacDonald, for providing encouragement, motivation, and guidance during my research. In addition, he greatly improved the quality of this paper through invalueable comments on earlier drafts. I also would like to thank him for the financial support that, among others, allowed me to attend the AAAI conference this year. This research was made possible through financial support of the university of Calgary.

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] Thomas M. Mitchell. *Version Spaces: An Approach to Concept Learning*. Phd thesis, Stanford University, Stanford, CA, 1977.

[Mo90] Dan Hua Mo. Learning text editing procedures from examples. Masters, University of Calgary, 1990.

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

[RS88] Ronald Rivest and Robert Sloan. Learning complicated concepts reliably and usefully. In *AAAI–88, Proceedings of the Seventh National Conference on Artificial Intelligence*, volume 2, pages 635–640, St. Paul Minnesota, 1988. AAAI, Morgan Kaufman.

[SR90] Benjamin D. Smith and Paul S. Rosenbloom. Incremental non–backtracking focusing: A polynomially bounded generalization algorithm for version spaces. In *Proceedings of the 8th National Conference on AI*, pages 848–853. AAAI, MIT Press, 1990.

[Van87] Kurt VanLehn. Learning one subprocedure per lesson. *Artificial Intelligence*, 31(1):1–40, 1987.

Appendix

A Complete Example Trace

```
Chez Scheme Version 3.9n
Copyright (c) 1989 Cadence Research Systems

> (load "load_sun.ss")
> (start)
Shell opened ID: 27530
What should I do ? Do an OLD task or Learn a NEW task
((old) (new))
> new
Is there anything I should know before we go through the task ?
YES, focus on this ...
NO, lets begin
((yes) (no))
> no
Lead me through the steps !
Are we DONE ?
Should I do an OLD task ?
Should I FOCUS on something ?
Should I stop focusing on the last item ?
Tell me what to do with items that do not match a focus ?
Do you want to give this task a name ?
((done) (old) (focus) (unfocus) (others) (name))
> ls
Ok. I will remember this step
 #make--focus.ss#         #script.txt#          #script2.txt#         #script3.txt#
 #test.ss#                CTEST.ss              DebMalloc.c           DebMalloc.c~
 FTEST2.ss                adjust.ss             adjust.ss.CKP         adjust.ss~
 backup.zoo               bak.ss                built--in_amiga.ss    built--in_sun.ss
 built--in_sun.ss~        built--in_sun.ver2    clerk--state.ss       clerk.ss
 clerk.ss~                command.ss            compile.ss            compile_sun.ss
 concepts.ss              concepts.ss~          convertgif.ss         copyss.ss
 csrc.ss                  csub                  csub--old             csub.c
 csub.c~                  csub.h                csub.h~               csub.o
 debmalloc.c~             debmalloc.o           delgif.ss             dialogue--stuff.ss
 dummy.ss                 dummy2.ss             else.ss               fileio_amiga.ss
 fileio_sun.ss            focus.ss              ftest.ss              generic
 generic.c                getA3000.ss           getlatt.ss            getnews.ss
 grep.ss                  junk.ss               junk2.ss              learn--rule.ss
 learning.ss              load_amiga.ss         load_sun.ss           load_sun.ss~
 loadc_sun.ss             lsdir_amiga.c         lsdir_sun             ltest.ss
 make--focus.ss           make--focus.ss~       make--task.ss         mycd
 mycd.c                   parameters.ss         print.ss              readfile.c
 recdir.ss                rm.ss                 rmgrep.ss             script.txt
 script.txt~              script2.txt           script2.txt~          script3.txt
 script3.txt~             stderr                test--session1.txt    test--spl.ss
 test--spl.ss~            test.sh               test.ss               test2.ss
 test2.ss~                test3.ss              test3.ss~             test4.ss
 undo.ss                  uudecode.ss           zoo2.ss               zoosrc.ss

Lead me through the steps !
Are we DONE ?
Should I do an OLD task ?
```

```
Should I FOCUS on something ?
Should I stop focusing on the last item ?
Tell me what to do with items that do not match a focus ?
Do you want to give this task a name ?
((done) (old) (focus) (unfocus) (others) (name))
> focus
Ok. What item's name should I focus on?
Please type in the item you want me to focus on ...
> adjust.ss
Is this the correct item, that I should focus on ?
(adjust . ss) YES/NO yes
Lead me through the steps !
Are we DONE ?
Should I do an OLD task ?
Should I FOCUS on something ?
Should I stop focusing on the last item ?
Tell me what to do with items that do not match a focus ?
Do you want to give this task a name ?
((done) (old) (focus) (unfocus) (others) (name))
> done
What is the name of the task ?
> ex6
What should I do ? Do an OLD task or Learn a NEW task
((old) (new))
> old
What is the name of the task ?
> ex6
Ok. I am doing ls now
 #make--focus.ss#       #script.txt#         #script2.txt#    #script3.txt#
 #test.ss#              CTEST.ss             DebMalloc.c      DebMalloc.c~
    ⋮                     ⋮                    ⋮                ⋮
```

New Example: #make–focus.ss#, Pos. Match = –1, Neg. Match = –1
Pos. Cover = (adjust) (.) (ss)
Neg. Cover = ()
Pos. Extra = ()
Neg. Extra = ()

```
I don't know what to do with this item #make--focus.ss#
Should I handle it like adjust.ss ?  YES/NO no
```

New Concept
Pos. Cover = (adjust) (.) (ss)
Neg. Cover = (#make–focus) (.) (ss#)
Pos. Extra = ()
Neg. Extra = ()

New Example: #script.txt#, Pos. Match = –1, Neg. Match = –1

```
I don't know what to do with this item #script.txt#
Should I handle it like adjust.ss ?  YES/NO no
```

New Concept
Pos. Cover = (adjust) (.) (ss)
Neg. Cover = (#make–focus #script) (.) (ss# txt#)
Pos. Extra = ()
Neg. Extra = ()

New Example: #script2.txt#, Pos. Match = –1, Neg. Match = –1

```
I don't know what to do with this item #script2.txt#
Should I handle it like adjust.ss ?  YES/NO no
```

New Concept
Pos. Cover = (adjust) (.) (ss)
Neg. Cover = (#make–focus #script #script2) (.) (ss# txt#)
Pos. Extra = ()
Neg. Extra = ()

New Example: #script3.txt#, Pos. Match = –1, Neg. Match = –1
```
I don't know what to do with this item #script3.txt#
Should I handle it like adjust.ss ?  YES/NO no
```
New Concept
Pos. Cover = (adjust) (.) (ss)
Neg. Cover = (#make–focus #script$D) (.) (ss# txt#)
Pos. Extra = ()
Neg. Extra = ()

New Example: #test.txt#, Pos. Match = –1, Neg. Match = –1
```
I don't know what to do with this item #test.ss#
Should I handle it like adjust.ss ?  YES/NO no
```
New Concept
Pos. Cover = (adjust) (.) (ss)
Neg. Cover = (#make–focus #script$D #test) (.) (ss# txt#)
Pos. Extra = ()
Neg. Extra = ()

New Example: CTEST.ss, Pos. Match = –1, Neg. Match = –1
```
I don't know what to do with this item CTEST.ss
Should I handle it like adjust.ss ?  YES/NO yes
```
New Concept
Pos. Cover = (adjust CTEST) (.) (ss)
Neg. Cover = (#make–focus #script$D #test) (.) (ss# txt#)
Pos. Extra = ()
Neg. Extra = ()

New Example: DebMalloc.c, Pos. Match = –1, Neg. Match = –1
```
I don't know what to do with this item DebMalloc.c
Should I handle it like adjust.ss ?  YES/NO yes
```
New Concept
Pos. Cover = (adjust CTEST DebMalloc) (.) (ss c)
Neg. Cover = (#make–focus #script$D #test) (.) (ss# txt#)
Pos. Extra = ()
Neg. Extra = ()

New Example: DebMalloc.c˜, Pos. Match = –1, Neg. Match = –1
```
I don't know what to do with this item DebMalloc.c˜
Should I handle it like adjust.ss ?  YES/NO no
```
New Concept
Pos. Cover = (adjust CTEST DebMalloc) (.) (ss c)
Neg. Cover = (#make–focus DebMalloc #LsLt$D) (.) (ss# txt# c˜)
Pos. Extra = ()
Neg. Extra = ()

New Example: FTEST2.ss, Pos. Match = –1, Neg. Match = –1
```
I don't know what to do with this item FTEST2.ss
Should I handle it like adjust.ss ?  YES/NO yes
```
New Concept
Pos. Cover = ($uTEST$D adjust DebMalloc) (.) (ss c)
Neg. Cover = (#make–focus DebMalloc #LsLt$D) (.) (ss# txt# c˜)
Pos. Extra = ()

Neg. Extra = ()

New Example: adjust.ss, Pos. Match = 0, Neg. Match = –1
```
Working on adjust.ss
```
New Example: adjust.ss.CKP, Pos. Match = –1, Neg. Match = –1

```
I don't know what to do with this item adjust.ss.CKP
Should I handle it like adjust.ss ?  YES/NO no
```
New Concept
Pos. Cover = ($uTEST$D adjust DebMalloc) (.) (ss c)
Neg. Cover = (CaCus$L DebMalloc #$LsLtD) (.) (ss$C txt# c˜)
Pos. Extra = ()
Neg. Extra = ()

New Example: adjust.ss˜ , Pos. Match = –1, Neg. Match = 28
The positive cover set can possibly yield a better match, therefore ask the user
```
I don't know what to do with this item adjust.ss˜
Should I handle it like adjust.ss ?  YES/NO no
```
New Concept
Pos. Cover = ($uTEST$D adjust DebMalloc) (.) (ss c)
Neg. Cover = (CaCus$L DebMalloc #$LsLtD) (.) (ss$C txt# c˜)
Pos. Extra = ()
Neg. Extra = (adjust) (.) (ss˜)

```
I don't know what to do with this item backup.zoo
Should I handle it like adjust.ss ?  YES/NO no
I don't know what to do with this item bak.ss
Should I handle it like adjust.ss ?  YES/NO yes
I don't know what to do with this item built--in_amiga.ss
Should I handle it like adjust.ss ?  YES/NO yes
I don't know what to do with this item built--in_sun.ss
Should I handle it like adjust.ss ?  YES/NO yes
```

New Concept
Pos. Cover = ($A built–in_amiga built–in_sun) (.) (ss c)
Neg. Cover = (CsA DebMalloc backup) (.) ($C)
Pos. Extra = ()
Neg. Extra = (adjust) (.) (ss˜)

New Example: built–in_sun.ss˜ , Pos. Match = –1, Neg. Match = 49
The positive cover set cannot possibly yield a better match, therefore skip this example
```
Skipping built--in_sun.ss˜
I don't know what to do with this item built--in_sun.ver2
Should I handle it like adjust.ss ?  YES/NO no
I don't know what to do with this item clerk--state.ss
Should I handle it like adjust.ss ?  YES/NO yes
```

New Concept
Pos. Cover = ($A built–in_$L clerk–state) (.) (ss c)
Neg. Cover = (CsA DebMalloc backup) (.) ($C)
Pos. Extra = ()
Neg. Extra = (adjust) (.) (ss˜)

New Example: clerk.ss, Pos. Match = 25, Neg. Match = –1
The negative cover set can possibly yield a better match, and the new example is not covered by the positive extra cover set, therefore ask the user.

```
I don't know what to do with this item clerk.ss
```

```
Should I handle it like adjust.ss ?  YES/NO yes
```

New Concept
Pos. Cover = ($A built–in_$L clerk–state) (.) (ss c)
Neg. Cover = (CsA DebMalloc backup) (.) ($C)
Pos. Extra = (clerk) (.) (ss)
Neg. Extra = (adjust built–in_sun clerk–state) (.) (ss˜ ver2)

```
I don't know what to do with this item clerk.ss˜
Should I handle it like adjust.ss ?  YES/NO no
I don't know what to do with this item command.ss
Should I handle it like adjust.ss ?  YES/NO yes
Working on compile.ss
I don't know what to do with this item compile_sun.ss
Should I handle it like adjust.ss ?  YES/NO yes
Working on concepts.ss
I don't know what to do with this item concepts.ss˜
Should I handle it like adjust.ss ?  YES/NO no
Working on convertgif.ss
Working on copyss.ss
Working on csrc.ss
I don't know what to do with this item csub
Should I handle it like adjust.ss ?  YES/NO no
I don't know what to do with this item csub--old
Should I handle it like adjust.ss ?  YES/NO no
Working on csub.c
I don't know what to do with this item csub.c˜
Should I handle it like adjust.ss ?  YES/NO no
I don't know what to do with this item csub.h
Should I handle it like adjust.ss ?  YES/NO yes
Skipping csub.h˜
Skipping csub.o
I don't know what to do with this item debmalloc.c˜
Should I handle it like adjust.ss ?  YES/NO no
Skipping debmalloc.o
I don't know what to do with this item delgif.ss
Should I handle it like adjust.ss ?  YES/NO yes
Skipping dialogue--stuff.ss
I don't know what to do with this item dummy.ss
Should I handle it like adjust.ss ?  YES/NO yes
I don't know what to do with this item dummy2.ss
Should I handle it like adjust.ss ?  YES/NO yes
Working on else.ss
I don't know what to do with this item fileio_amiga.ss
Should I handle it like adjust.ss ?  YES/NO yes
Working on fileio_sun.ss
Working on focus.ss
Working on ftest.ss
I don't know what to do with this item generic
Should I handle it like adjust.ss ?  YES/NO no
Working on generic.c
I don't know what to do with this item getA3000.ss
Should I handle it like adjust.ss ?  YES/NO yes
Working on getlatt.ss
Working on getnews.ss
Working on grep.ss
I don't know what to do with this item junk.ss
Should I handle it like adjust.ss ?  YES/NO yes
```

```
Working on junk2.ss
I don't know what to do with this item learn--rule.ss
Should I handle it like adjust.ss ?  YES/NO yes
Working on learning.ss
I don't know what to do with this item learning.ss~
Should I handle it like adjust.ss ?  YES/NO no
I don't know what to do with this item load_amiga.ss
Should I handle it like adjust.ss ?  YES/NO yes
Working on load_sun.ss
Skipping load_sun.ss~
Working on loadc_sun.ss
I don't know what to do with this item lsdir_amiga.c
Should I handle it like adjust.ss ?  YES/NO yes
I don't know what to do with this item lsdir_sun
Should I handle it like adjust.ss ?  YES/NO no
Working on ltest.ss
Skipping make--focus.ss
Skipping make--focus.ss~
Skipping make--task.ss
I don't know what to do with this item mycd
Should I handle it like adjust.ss ?  YES/NO no
Working on mycd.c
Working on parameters.ss
Working on print.ss
Working on readfile.c
Working on recdir.ss
Working on rm.ss
Working on rmgrep.ss
Skipping script.txt
Skipping script.txt~
I don't know what to do with this item script2.txt
Should I handle it like adjust.ss ?  YES/NO no
Skipping script2.txt~
I don't know what to do with this item script3.txt
Should I handle it like adjust.ss ?  YES/NO no
Skipping script3.txt~
I don't know what to do with this item stderr
Should I handle it like adjust.ss ?  YES/NO no
```

New Example: test–session1.txt, Pos. Match = –1, Neg. Match = 69
The positive cover set can possibly yield a better match, and the new example is not covered by the negative extra cover set, therefore ask user

```
I don't know what to do with this item test--session1.txt
Should I handle it like adjust.ss ?  YES/NO yes
```

New Concept
Pos. Cover = (LlC $A learn–rule) (.) (ss c h)
Neg. Cover = ($B CsA) ($P –) ($C)
Pos. Extra = (lLdL_amiga $A junk) (.) (ss c h)
Neg. Extra = (CsA $B) ($P –) ($C)

```
Skipping test--spl.ss
Skipping test--spl.ss~
Skipping test.sh
Working on test.ss
Working on test2.ss
Skipping test2.ss~
Working on test3.ss
```

```
Skipping test3.ss~
Working on test4.ss
Working on undo.ss
Working on uudecode.ss
Working on zoo2.ss
Working on zoosrc.ss
I completed the task you told me to do
((continue))
> continue
What should I do ? Do an OLD task or Learn a NEW task
((old) (new))
> exit

Process lisp finished
```