Forgetting and Compacting data in Concept Learning

Gunther Sablon and Luc De Raedt
Department of Computer Science, Katholieke Universiteit Leuven
Celestijnenlaan 200A, B-3001 Heverlee, Belgium
Email: {Gunther.Sablon,Luc.DeRaedt}@cs.kuleuven.ac.be
Fax: ++ 32 16 32 79 96 Telephone: ++ 32 16 32 75 50

Abstract

Incremental concept learning algorithms using backtracking have to store previous data. These data can be ordered by the "is more specific than" relation. Using this order only the most informative data have to be stored, and the less informative data can be discarded. Moreover, under certain conditions some data can be replaced by automatically generated, more informative data.

We investigate some conditions for data to be discarded, independently of the chosen concept learning algorithm or concept representation language. Then an algorithm for discarding data is presented in the framework of Iterative Versionspaces, which is a depth-first algorithm computing versionspaces as introduced by Mitchell. We update the datastructures used in the Iterative Versionspaces algorithm, while preserving its most important properties.

1 Introduction

Incremental concept learning algorithms maintaining a hypothesis consistent with all data (usually called *examples* or *instances)* have to store *all* previous data as soon as any backtracking is involved. Exceptions are, e.g., the Candidate Elimination algorithm [Mitchell, 1982], because it searches bi-directionally (i.e., specific-to-general and general-to-specific) and breadth-first, or algorithms searching specific-to-general in a conjunctive tree-structure language, as Incremental Non-Backtracking Focusing [Smith and Rosenbloom, 1990]. [Hirsh, 1992] even prefers a representation storing all negative examples together with *S* over storing *S* and G in case G can grow exponentially or can be infinite. [Bundy *et al.*, 1985] argues that for learning disjunctive concepts all data will have to be stored anyway.

One of the goals of concept learning is *compaction* of the information provided to the algorithm. Therefore, in cases where all instances have to be memorized, preferably no redundant information should be stored. In this paper, we remove redundant instances in a language independent way by *partially ordering* them, according to their information contents. In [Sebag, 1994] and [Sebag and Rouveirol, 1994] this is done for negative examples in a conjunctive tree-structure, resp. first order logic language. According to the partial order, we only have to store minimal and maximal instances, while forgetting the ones with less information content. However, we have to take care that the search algorithm does not lose any solutions, does not search previously discarded parts of the search space again, and retains its most interesting properties.

We develop this idea in the framework of the Iterative Versionspaces algorithm (ITVS) [Sablon *et al.*, 1994]. Nevertheless, we argue that it has a much wider application potential. The theory is formulated independently from any concept learning algorithm or search strategy and independently from the chosen concept representation language. The partial order on instances is defined solely in terms of the "is more specific than" relation. Identifying and removing redundant instances can be used in any incremental algorithm that stores all instances, and even in a preprocessing phase of a non-incremental concept learning algorithm, to reduce its actual processing time. The reason for studying this problem in the context of ITVS, is that we believe the datastructures and complexity measures of ITVS contribute to understanding the nature and the complexity of concept learning.

We ensure that the main properties of ITVS are retained: a worst case space complexity linear in the number of instances, and a worst case time complexity of testing a candidate hypothesis for maximal generality or maximal specificness also linear in the number of instances. The cost of extending the ITVS algorithm is a global increase in time complexity quadratic in the number of instances. The gain is twofold: firstly storing less instances will reduce the memory needed by the algorithm. Secondly, in case the size of *S* or G is exponential in the number of instances, the worst case time complexity of the search is exponential in the number of instances. With a branching factor 6, reducing the number of instances with a factor k, would then reduce the time complexity with a factor b^k.

The paper is organized as follows: Section 2 briefly reviews the definitions and invariants of the datastructures used in ITVS. Section 3 describes the theoretical background for discarding instances. In Section 4 we give an algorithm for updating ITVS's datastructures consistently. In Section 5 we show that the most important properties of ITVS are preserved. Related work is discussed in Section 6. Finally we conclude in Section 7.

2 Iterative Versionspaces

In this section, we will briefly introduce some notation and review the ITVS algorithm. For a more detailed description of ITVS, we refer to [Sablon *et al.*, 1994].

The language of hypotheses is denoted by $\mathcal{L}$. We assume the single representation trick applies[1], so that instances also belong to $\mathcal{L}$. Both relations "is more specific than" and the classical "is covered by" on $\mathcal{L}$ can then be denoted by one symbol $\preccurlyeq$. We assume there is a maximal element $\top$ and a minimal element $\bot$ in $\mathcal{L}$. As in [Mellish, 1991] ITVS accepts four kinds of instances:

- a positive lowerbound (or positive example) i must be more specific than the target concept c, i.e., $i \preccurlyeq c$;
- a negative lowerbound (or negative example) i must not be more specific than the target concept c, i.e., $\neg(i \preccurlyeq c)$;
- a positive upperbound i must be more general than the target concept c, i.e., $c \preccurlyeq i$;
- a negative upperbound i must not be more general than the target concept c, i.e., $\neg(c \preccurlyeq i)$.

The set of all instances at a given moment is denoted by I. Positive lowerbounds and negative upperbounds are referred to as s-bounds. Positive upperbounds and negative lowerbounds are referred to as g-bounds. That hypothesis c is consistent with the instance i, is denoted by $c \sim i$. For a given positive lowerbound, resp. negative upperbound, i and a hypothesis c, the *complete* generalization operator $lub(c , i)$ (least upperbounds), resp. $msg(c , i)$ (most specific generalizations), computes the maximally specific generalizations of c consistent with i. For a given positive upperbound, resp. negative lowerbound, i and a hypothesis c, the *complete* specialization operator $glb(c , i)$ (greatest lowerbounds), resp. $mgs(c , i)$ (most general specializations), computes the maximally general specializations of c consistent with i. $\mathcal{S}$ is the set of all maximally specific hypotheses consistent with I, $\mathcal{G}$ the set of all maximally general hypotheses consistent with I. In order to represent the set of *all* consistent concept representations by $\mathcal{S}$ and $\mathcal{G}$, we assume that the admissability constraint holds on $\mathcal{L}$, i.e., every chain in $\mathcal{L}$ contains a maximal and a minimal element [Mitchell, 1978].

ITVS is a bi-directional incremental depth-first search algorithm on $\mathcal{L}$ using the following datastructures:

- ITVS stores one current maximally specific hypothesis s and one current maximally general hypothesis g, both consistent with all s-bounds and g-bounds;
- ITVS stores all s-bounds in an array I_s and all g-bounds in an array I_g. n_s is the total number of s-bounds, n_g is the total number of g-bounds;

[1][Sablon, 1995] shows that the presented results can be generalised beyond the single representation trick.

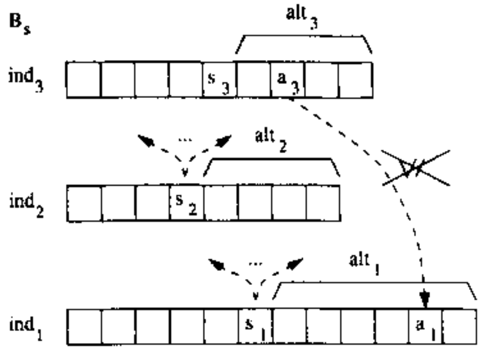

Figure 1 Maximal specificness on B_s

- ITVS stores specific-to-general backtrack information on the stack[2] B_s, containing triplets[3] $(ind , s_{ind} , alt_{ind})$, called choicepoints (see Figure 1); ind is an index in I_s, s_{ind} is a hypothesis, and alt_{ind} is a non-empty list of hypotheses used to backtrack on s and to test maximal specificness of s. Similarly, the stack B_g contains choicepoints $(ind , g_{ind} , alt_{ind})$, where ind is an index in I_g, g_{ind} is a hypothesis, and alt_{ind} is a non-empty list of hypotheses used to backtrack on g and to test maximal generality of g.

Figure 1 shows a stack B_s with three choicepoints. Elements of $\mathcal{L}$ are drawn as squares. In each choicepoint the squares on the left of s_{ind} are already discarded alternatives. alt_{ind} contains the alternatives still to be explored. s_{ind} is the alternative currently explored: its minimal generalizations are in the next choicepoint.

The Candidate Elimination algorithm of [Mitchell, 1982] is a bi-directional breadth-first algorithm which computes and stores $\mathcal{S}$ and $\mathcal{G}$ completely for every new instance. ITVS being a depth-first algorithm only stores one element $s \in \mathcal{S}$ and one $g \in \mathcal{G}$, together with backtrack information B_s and B_g. When s (or g) is inconsistent with a new instance, ITVS generalizes s (resp. specializes g), or selects a next alternative on B_s (resp. on B_g) and reprocesses all instances encountered since the choicepoint of the alternative was created. Therefore ITVS stores all instances in I_s and I_g. By backtracking, ITVS can reconstruct $\mathcal{S}$ and $\mathcal{G}$ for each new instance.

During the search the following invariants hold:

1. $s \in \mathcal{S}$ and $g \in \mathcal{G}$;
2. for all choicepoints (ind_1 , s_1 , alt_1) on B_s: $s_1 \preccurlyeq s$ and $\neg(a_1 \preccurlyeq s)$ for every a_1 in alt_1; s_1 and all elements of alt_1 are maximally specific hypotheses consistent with *all* g-bounds and the first ind_1 s-bounds; for every choicepoint (ind_2 , s_2 , alt_2)

[2]We employ the usual operations push, pop and is_empty on stacks. Elements are added on and removed from *the top* of the stack.

[3]W.r.t. [Sablon *et al.*, 1994] choicepoints are extended with s_{ind}, resp. g_{ind}, necessary for an efficient implementation of the algorithms in Section 4 (see also footnote 7).

closer to the top of B_s: $ind_1 < ind_2$, $s_1 \prec s_2$, $s_1 \prec a_2$ and $\neg(a_1 \preccurlyeq a_2)$ for every a_1 in alt_1 and for every a_2 in alt_2;

3. for all choicepoints (ind_1 , g_1 , alt_1) on B_g: $g \preccurlyeq g_1$ and $\neg(g \preccurlyeq a_1)$ for every a_1 in alt_1; g_1 and all elements of alt_1 are maximally general hypotheses consistent with *all* s-bounds and the first ind_1 g-bounds; for every choicepoint (ind_2 , g_2 , alt_2) closer to the top of B_g: $ind_1 < ind_2$, $g_2 \prec g_1$ and $a_2 \prec g_1$ $\neg(a_2 \preccurlyeq a_1)$ for every a_1 in alt_1 and for every a_2 in alt_2;
4. Completeness of s and B_s: for all $c \in \mathcal{L}$, consistent with I, s or an alternative on B_s[4] is more specific than c;
5. Completeness of g and B_g: for all $c \in \mathcal{L}$, consistent with I, g or an alternative on B_g is more general than c.

Note that if the top choicepoint of B_s has index n_s, s_{ind} of the top must be equal to s. Also note that the alternatives on B_s are actually the roots of the search subtrees that are still to be searched. ITVS tests for maximal specificness of an hypothesis a_3 by testing whether or not there is an alternative a_1 on B_s which is more specific than a_3 (see Figure 1). In other words, this tests whether the search subtree rooted by a_3 is a subtree of the tree rooted by a_1. If it is, a_3 is not further generalized at this moment (so it is not allowed in alt_3), because either it is not maximally specific, or it will be generalized later when all generalizations of a_1 are being searched. Thus an optimal generalization operator[5], meaning that every hypothesis is generalized only once, is implemented explicitly. All these arguments dually hold for g and B_g.

An important theoretical result about ITVS is the fact that its worst case space complexity is linear in the number of instances, while the worst cast case time complexity could be exponential (though only a linear factor worse than the Candidate Elimination algorithm, if $\mathcal{S}$ and $\mathcal{G}$ are exponential in size and have to be computed for each new instance). Another result is the fact that the worst case time complexity of testing maximal specificness of s or maximal generality of g is also linear in the number of instances.

3 Redundant instances

In this section, we will develop a theory to reason about redundant instances. Due to space limitations proofs are omitted and can be found in [Sablon, 1995].

We first define the information elements we are focussing on. Theorem 2 proves they are redundant.

Definition 1 (s-prunable and g-prunable)

- $i_1 \in I_s$ *is s-prunable w.r.t.* $i_2 \in I_s$ *iff*
  - i_1 *and* i_2 *are both positive lowerbounds such that* $i_1 \preccurlyeq i_2$, *or*
  - i_1 *and* i_2 *are both negative upperbounds such that* $i_1 \preccurlyeq i_2$, *or*
  - i_1 *is a negative upperbound and* i_2 *is a positive lowerbound such that* $i_1 \prec i_2$.
- $i_1 \in I_s$ *is s-prunable in* I_s *iff* $\exists i_2 \in I_s$ *such that* i_1 *is s-prunable w.r.t.* i_2.
- $i_1 \in I_g$ *is g-prunable w.r.t.* $i_2 \in I_g$ *iff*
  - i_1 *and* i_2 *are positive upperbounds such that* $i_2 \preccurlyeq i_1$, *or*
  - i_1 *and* i_2 *are negative lowerbounds such that* $i_2 \preccurlyeq i_1$, *or*
  - i_1 *is a negative lowerbound and* i_2 *is a positive upperbound such that* $i_2 \prec i_1$.
- $i_1 \in I_g$ *is g-prunable in* I_g *iff* $\exists i_2 \in I_s$ *such that* i_1 *is g-prunable w.r.t.* i_2.

Theorem 2 *Given* $c \in \mathcal{L}$. *Also given* $i_1, i_2 \in I_g$ *such that* i_1 *is g-prunable w.r.t.* i_2, *or* $i_1, i_2 \in I_s$ *such that* i_1 *is s-prunable w.r.t.* i_2. *Then* $i_2 \sim c$ *implies* $i_1 \sim c$. □

The proof of Theorem 2 is a repeated application of the transitivity of $\preccurlyeq$. As a consequence of Theorem 2, we do not have to store *all* instances, but rather only the non s-prunable ones and the non g-prunable ones. In other words, only the *maximally general* positive lowerbounds and negative upperbounds, and the *maximally specific* positive upperbounds and negative lowerbounds are to be stored. Whenever we detect that a previously stored s-bound i_1 is s-prunable w.r.t. a newly provided one i_2, we replace i_1 by i_2 in I_s.

So far we assume all instances are *provided* to the concept learning algorithm. However, under certain conditions new instances with an information content equivalent to the provided ones can be *automatically created*. Moreover, the automatically created instances enforce that more provided instances will become s-prunable or g-prunable, so that less instances have to be stored, i.e., data is compacted without loss of information content. We will now describe some conditions under which such instances can be automatically generated.

Lemma 3 *Given* $c \in \mathcal{L}$. *Also given*

3.1 two positive lowerbounds i_1 *and* i_2 *such that* $lub(i_1 , i_2) = \{ i \}$, *or*

3.2 two positive upperbounds i_1 *and* i_2 *such that* $glb(i_1 , i_2) = \{ i \}$.

Then $c \sim i$ *iff* $c \sim i_1$ *and* $c \sim i_2$. □

This means that whenever two positive lowerbounds have a unique least upperbound they may be *replaced* by this least upperbound without loss of information, and whenever two positive upperbounds have a unique greatest lowerbound, they may be replaced by this greatest lowerbound. Note that in a conjunctive tree-structure language and in Inductive Logic Programming using θ-subsumption *lub* and *glb* are always unique.

Theorem 4 *Given* $i_1 \in I$ *and a new instance* i_2, *fulfilling Condition 3.1 or Condition 3.2. The set of hypotheses consistent with* $I \cup \{ i \}$ *is the set of hypotheses consistent with* $I \cup \{ i_2 \}$. *Moreover,* i_1 *is s-prunable w.r.t.*

[4] With "an alternative on B_s" we mean "an element of alt_{ind} for a choicepoint (ind , s_{ind} , alt_{ind}) on B_s". Also, with "all alternatives on B_s" we mean "all elements of alt_{ind} for all choicepoints (ind , s_{ind} , alt_{ind}) on B_s".

[5] An optimal generalization operator is dual to an optimal refinement operator (see [Sablon *et al.*, 1994]).

i under Condition 3.1, and g-prunable w.r.t. i under Condition 3.2. □

In particular, in the case of a positive lowerbound i_2, for instance, i can be provided to ITVS instead of i_2 without losing any solutions. Then i_1 becomes redundant. Whenever a least upperbound i replaces a positive lowerbound i_1, all other s-bounds will have to be checked whether they are not s-prunable, or whether they have more than one least upperbound with i. Unfortunately, the result of replacing instances repeatedly depends on the order in which the instances are provided.

Negative instances cannot be generalized or specialized in the same way. However, a special kind of negative instances can be transformed to positive instances.

Definition 5 (A general notion of near-miss) *A near-miss*[6] *w.r.t.* $c \in \mathcal{L}$ *is a negative lowerbound* i_n *such that* $\{ x \in mgs(\top , i_n) \mid c \preccurlyeq x \}$ *is a singleton* $\{ i_u \}$.

Because of the single representation trick, positive lowerbounds are also in $\mathcal{L}$. Given a near-miss i_n w.r.t. a positive lowerbound i_p, the target concept must be more general than i_p to be consistent with i_p, but also more specific than the corresponding i_u to be consistent with i_n. In other words, i_u is a positive upperbound constraining the search space in the same way as i_n does. If we replace each near-miss w.r.t. a positive lowerbound by its equivalent positive upperbound, we can apply Theorem 4 when appropriate. [Sablon, 1995] also formulates a dual result for s-bounds.

This definition of near-miss is consistent with the usual definition in a conjunctive tree-structure language, since $mgs(\top , i_n)$ will only be a singleton if i_n and s differ in only one attribute. Theorem 4 explains that providing the only consistent maximally specific hypothesis s as positive lowerbound is equivalent to providing all actual positive lowerbounds. In this case all near-misses can be replaced by exactly one positive upperbound, since the corresponding positive upperbounds have only one greatest lowerbound (glb). This corresponds to the result of [Smith and Rosenbloom, 1990].

4 The algorithm

In the previous section we have determined which instances are redundant. We will now modify ITVS such that no redundant instances are stored. This will be enforced by the following two extra invariants:

6. no element in I_s is s-prunable , and

7. no element in I_g is g-prunable.

Given these invariants, how to update ITVS's datastructures once a redundant instance is detected? We describe an extension of ITVS to discard s-prunable instances. Pruning g-prunable instances from I_g is done dually. Also, replacing instances using Theorem 4 is not discussed because of space limitations.

In ITVS, when i_1 is s-prunable w.r.t. i, a naive method to update B_s would be replace i_1 in I_s by i and to reprocess all s-bounds with an index in I_s larger than the index of i_1. However, this reprocessing could lead to recomputing and generalizing previously discarded elements of $\mathcal{L}$. Therefore, we will *update* all alternatives on B_s, instead of recomputing them, while respecting B_s's invariants. A dual argument holds for B_g.

Further on we also need the following lemmas:

Lemma 6 *Given* $c_1, c_2 \in \mathcal{L}$ *with* $c_1 \preccurlyeq c_2$ *and a positive lowerbound* i:
$\forall x_2 \in lub(c_2 , i) : \exists x_1 \in lub(c_1 , i) : x_1 \preccurlyeq x_2$. □

Lemma 7 *Given* $c_1, c_2 \in \mathcal{L}$ *with* $c_1 \preccurlyeq c_2$ *and a negative upperbound* i:
$\forall x_2 \in msg(c_2 , i) : \exists x_1 \in msg(c_1 , i) : x_1 \preccurlyeq x_2$. □

These lemmas assure that generalization in some sense preserves the property of being more general. Analogous lemmas exist for *glb* and *mgs* (see [Sablon, 1995]).

We will now describe how we extended ITVS to satisfy Invariant 6 and Invariant 7. When a new s-bound is provided to the original ITVS, this s-bound is added to I_s, then g and B_g are updated, and then s and B_s are updated. W.r.t. the original ITVS only the latter operation (i.e., the call *generalize*(s , B_s , s_{ind}) in line ©on page 399 of [Sablon *et al.*, 1994]) should be replaced by a call to Algorithm 1 with the same arguments.

Given the s-bound i:
1 Search I_s from 1 to n_s for the first instance $I_s[n_c]$ that is s-prunable w.r.t. i, or such that i is s-prunable w.r.t. $I_s[n_c]$
2 If no such instance can be found, then handle i as before in ITVS
3 Otherwise, remove i from I_s, and:
  4 If i is s-prunable, then do nothing
  5 Otherwise, $I_s[n_c]$ must be s-prunable. Then:
    6 Replace $I_s[n_c]$ by i
    7 Generalize B_s as in Algorithm 2
      This yields s, B_s and *ind*, the index up to where s is consistent with all s-bounds
    8 Find s consistent with I and corresp. B_s with the call *generalize*(s , B_s , *ind*)
9 Return s and B_s

Algorithm 1 Handling a new s-bound

Algorithm 1 first checks whether Invariant 6 on I_s is still satisfied (Step 1). If it is (Step 2), ITVS continues as before (i.e., with the call *generalize*(s , B_s , s_{ind}); see above). If i is s-prunable, it is just removed from I_s (Step 3 and Step 4). Otherwise it is also removed from I_s, and then replaces $I_s[n_c]$ (Step 6). Then B_s must be updated to satisfy Invariant 2 and Invariant 4 of Section 2 (Step 7). This is explained in Algorithm 2. The result is a new maximally specific concept representation s, the updated B_s, and an index *ind* such that s is consistent with all g-bounds and with the first *ind* s-bounds. Finally, a new maximally specific concept representation consistent with *all* instances must be computed using the procedure *generalize* of [Sablon *et al.*, 1994]. This call will satisfy Invariant 1. In Algorithm 1 and Algorithm 2 neither g, nor B_g, nor I_g, nor n_g are changed, so Invariant 3 and Invariant 5 are preserved all the time.

[6] Originally introduced by [Winston, 1975].

10 Initialize B_h to an empty stack
11 If B_s is empty, or else if the top choicepoint of B_s does not have index n_s, then push (n_s , s , []) onto B_h
12 Pop all (ind , s_{ind} , alt_{ind}) with $n_c \leq ind$ from B_s, and push them on B_h
13 Initialise new_ind and n'_s as n_c
14 Initialise $prune\_B_h$ as $false$
15 Repeat
  16 Pop (ind , s_{ind} , alt_{ind}) from B_h
  17 Generalise s_{ind} minimally to be consistent with i; assign to $gens_1$ the list of generalisations x such that:
    18 $x \sim I_g$, and
    19 $\neg\exists a$ in alt_{ind}: $a \preccurlyeq x$, and
    20 $\neg\exists a$ on B_s: $a \preccurlyeq x$, and
    21 ($s \preccurlyeq x$ or $\exists a$ on B_h: $a \preccurlyeq x$)
  22 Generalise the elements of alt_{ind} minimally to be consistent with i; assign to $gens_2$ the list of generalizations x such that:
    23 there exists no other such generalization x', such that $x' \preccurlyeq x$, and
    24 $x \sim I_g$, and
    25 $\neg\exists a$ on B_s: $a \preccurlyeq x$, and
    26 $\neg\exists a$ in $gens_1$: $a \preccurlyeq x$
  27 If $gens_1$ is empty, then
    28 Let $prune\_B_h$ be true
    29 Move all $I_s[new\_ind+1], \ldots, I_s[n_s]$ not s-prunable to $I_s[n'_s+1], \ldots, I_s[k]$ and let n'_s be equal to k
  30 Otherwise (i.e., $gens_1$ is not empty):
    31 Move all $I_s[new\_ind+1], \ldots, I_s[ind]$ not s-prunable to $I_s[n'_s+1], \ldots, I_s[k]$ and let n'_s be equal to k
  32 Take the first possible choice of:
    - Remove an element new_s_{ind} from $gens_1$ and then let alt_{ind} be $gens_1 \cup gens_2$
    - Remove an element new_s_{ind} from $gens_2$ and then let alt_{ind} be $gens_1 \cup gens_2$
    - Pop (ind , s_{ind} , alt_{ind}) from B_s and then remove a new_s_{ind} from alt_{ind}
    - Fail and halt
  33 Let new_ind be equal to ind
  34 If alt_{ind} is not empty, Then push (ind , new_s_{ind} , alt_{ind}) onto B_s
35 Until B_h is empty or $prune\_B_h$
36 Let n_s be equal to n'_s
37 Return new_s_{ind}, B_s and new_ind

Algorithm 2 Generalizing B_s

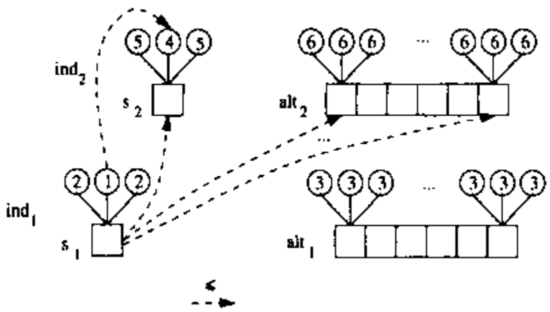

Figure 2 Two consecutive choicepoints of B_s

In general terms, Algorithm 2 works as follows: it first pops the choicepoints of B_s that are to be generalized from B_s and pushes them onto a temporary stack B_h. It then generalizes these choicepoints one by one w.r.t. the new instance i. The result is a new B_s satisfying Invariant 2 and Invariant 4. Simultaneously, all instances i' s-prunable w.r.t. i are removed from I_s, by shifting the instances following i' towards the front. We will now discuss Algorithm 2 in more detail.

Step 10 to Step 14 consist of initializations. B_h consists of all choicepoints of B_s with index larger than or equal to n_c in reversed order (Step 12). Also, it is ensured that the bottom choicepoint on B_h has s as s_{ind} (Step 11) to handle s as any other s_{ind}, and to be certain that B_h is not empty. For each ind, new_s_{ind} will replace s_{ind} on the generalized B_s; new_ind is the index in I_s up to where elements of I_s have been checked for not being s-prunable, and up to where new_s_{ind} is consistent with all s-bounds; n'_s is the index up to where I_s contains the non s-prunable instances of $I_s[1..n_c]$. Initially, $new\_ind = n'_s = n_c$ (Step 14), since $I_s[n_c]$ was the first s-prunable instance in I_s (see Algorithm 1). Finally, $prune\_B_h$ will be true iff the rest of B_h cannot be generalized consistently; $prune\_B_h$ is initialized to $false$.

Let (ind_1 , s_1 , alt_1) and (ind_2 , s_2 , alt_2) be two consecutive choicepoints such that $ind_1 < ind_2$, s_2 and the elements of alt_2 are more general than s_1, maximally specific and consistent with all g-bounds and with $I_s[1]$ to $I_s[ind_2]$. This situation is presented in Figure 2. The squares depict the state of the two choicepoints *before* Algorithm 2. The dashed arrows represent $\preccurlyeq$. The circles are generalizations of the square they are connected to. The numbers inside the circles are labels.

We explain what happens inside the repeat-loop (Step 15) when choicepoint (ind_2 , s_2 , alt_2) is popped from B_h (Step 16). Suppose that choicepoint (ind_1 , s_1 , alt_1) has already been generalized: the new s_1 is labeled 1, the elements of the new alt_1 are labeled 2 and 3. First s_2 is generalized (Step 17): $gens_1$ should contain all maximally specific consistent generalizations of s_2, not reachable from an alternative on B_s and not yet explored or discarded before. First *all* minimal generalisations consistent with i are computed: if i is a positive lowerbound it is the set $lub($ s_2 , i $)$; if i is a negative upperbound it is the set $msg($ s_2 , i $)$. On Figure 2 these generalizations are labeled 4 and 5. From this set of generalizations, generalizations that are not consistent (Step 18), or not maximally specific, as well as those still reachable from some alternative in alt_2 or on B_s, are removed. The latter two conditions are implemented as in ITVS by Step 19 and Step 20 (see Section 2). This shows the use of an optimal generalization operator is still possible in the extended ITVS. Finally, since all elements in $gens_1$ are generalizations of s_2, and since all alternatives more general than s_2 and still to be explored are the alternatives on B_h, together with s, Step 21 selects only those generalizations more general than s or than some alternative on B_h. The list of selected elements is assigned to $gens_1$. In $gens_1$ all elements are more general than the hypothesis labeled 1, since each of the generalizations of s_2 is more general than some generalization of s_1 (Lemma 6 and Lemma 7) and since the ones more general than the elements labeled 2 (which are on B_s already) are not selected for $gens_1$.

Then all elements of alt_2 are generalized (Step 22): $gens_2$ should contain all maximally specific consistent generalizations of the elements of alt_2, not reachable from another alternative on B_s. First *all* minimal generalizations consistent with i are computed: if i is a positive lowerbound it is the union of all sets $lub(\ a\ ,\ i\ )$, with $a \in alt_2$; if i is a negative upperbound it is the union of all sets $msg(\ a\ ,\ i\ )$, with $a \in alt_2$. On Figure 2 this union consists of the circles labeled 6. From this union, generalizations that are more general than another such generalization (Step 23), not consistent (Step 24), or not maximally specific, as well as those still reachable from some other alternative in $gens_1$ or on B_s, are removed. The latter two conditions are again implemented as in ITVS by Step 25 and Step 26. The selected elements are assigned to $gens_2$. Like for $gens_1$, all elements of $gens_2$ are more general than the hypothesis labeled 1.

Then, if $gens_1$ is empty, every generalization of s_2 consistent with i is more general than an alternative on B_s. Therefore it is not necessary to generalize the other alternatives on B_h, since they are all generalizations of s_2, so $prune\_B_h$ is set to $true$ (Step 28). All remaining non s-prunable instances of I_s are shifted in I_s towards the front (Step 29).

Otherwise, if $gens_1$ is not empty, of all instances $I_s[new\_ind + 1]$ up to $I_s[ind_2]$ only the ones not s-prunable are shifted in I_s towards the front, thus removing the s-prunable ones (Step 31).

Then a new value for new_s_{ind} must be chosen. If $gens_1$ is not empty, one element of $gens_1$ is chosen. If it is empty, an element of $gens_2$ is chosen. In both cases, the rest of $gens_1 \cup gens_2$ contains alternatives to be explored later, and is assigned to alt_{ind}. If $gens_1$ and $gens_2$ are both empty, a choicepoint $(\ ind\ ,\ s_{ind}\ ,\ alt_{ind}\ )$ is popped from B_s, and one of the elements of alt_{ind} is chosen as new_s_{ind}. If $gens_1$, $gens_2$ and B_s are all empty, no hypothesis consistent with all instances exists. Consequently, ITVS fails and halts. In all three non-failing cases, new_s_{ind} and all elements in alt_{ind} are then consistent with all g-bounds, and consistent with $I_s[1]$ up to $I_s[ind]$. Since ind is the index up to where new_s_{ind} is consistent with all s-bounds, and the number up to where the elements of I_s are checked for being s-prunable, ind must be the new value of new_ind.

By induction, Invariant 2 holds after the loop. After the loop n_s' is assigned to n_s (Step 36), since it is the index up to where I_s contains all non s-prunable s-bounds. Finally, new_s_{ind}, consistent with new_ind s-bounds, the updated B_s and new_ind are returned.

[Sablon, 1995] proves that Invariant 4 is also preserved, i.e., that no solutions are lost during the update of B_s.

5 Discussion

First note that Invariant 6 and Invariant 7 are expressed solely in terms of I_s, resp. I_g, and are therefore independent of any search strategy or concept language: they only constrain the set of instances that is stored.

We now discuss the cost of extending ITVS with Algorithm 1 and Algorithm 2.

Theorem 8 *The original ITVS and the extended ITVS have the same worst case space complexity: they are linear in the number of instances.* □

In the case of s-bounds, the proof actually shows the following: suppose $i_1 - i_2 - i_3$ is a sequence of s-bounds, where i_1 is s-prunable w.r.t. i_3, and not s-prunable w.r.t. i_2. When provided to the extended ITVS, this sequence gives exactly the same B_s as when the sequence $i_3 - i_2$ were provided to the original ITVS[7]. Consequently, both algorithms have the same worst case space complexity.

For the worst case time complexity, we count the number of $\preccurlyeq$ tests (see also [Mitchell, 1982]).

Theorem 9 *The worst case time complexity of the extended ITVS has an extra term of $O(\ n^2\ )$, where n is the number of instances.* □

On the one hand, no parts of the search space are explored more than once in the extended ITVS. On the other hand, as a consequence of Theorem 8, the worst case time complexity of the tests for maximal generality and maximal specificness remain linear in the number of instances. Consequently, we only have to add the overhead of testing instances for being s-prunable or g-prunable, and the overhead of updating B_s and B_g. In case no s-prunable instances are provided to the extended ITVS, it will have an overhead w.r.t. the original ITVS of comparing each new s-bound to all previous s-bounds, and comparing each new g-bound to all previous g-bounds. Furthermore, for detecting near-misses and their dual counterpart, s-bounds are also compared to g-bounds, and vice versa. If an instance is found to be s-prunable, the major term in generalising B_s is the comparison of each newly generated hypothesis in Step 22 to all elements on B_s, i.e., also a term of $O(\ n^2\ )$.

[7] Note that we really needed to extend the original representation of ITVS with s_{ind} to obtain this result. If we would not be able to generalise s_{ind} in each choicepoint (see Step 17 of Algorithm 2), but rather generalise only alt_{ind}, and finally generalise s, the worst case space complexity would be worse than the one of the original ITVS. In that case each alternative on B_s would potentially be replaced by b_s others (where b_s is the generalisation branching factor), in the worst case leading to $b_s^{n_s}$ alternatives on B_s.

Although the worst case time complexity has increased w.r.t. ITVS's, it has only increased with a quadratic term, while, if the size of *S* or G is exponential, reducing the number of 5-bounds, resp. g-bounds, would also reduce Bearch time with an exponential factor.

6 Related Work

The ideas extend the work of [Sebag, 1994], [Sebag and Rouveirol, 1994] and [Smith and Rosenbloom, 1990] in a language independent framework. In [Sebag, 1994], which is restricted to conjunctive tree-structure languages, negative lowerbounds are converted into positive upperbounds, and only those *nearest* to the target concept (i.e., the most specific ones) are stored. In Sebag and Rouveirol, 1994] this is extended to negative lowerbounds in ILP, which are represented by integrity constraints and ordered by 0-subsumption. In our framework we can generalize the notion of a *nearest miss* (which is introduced in [Sebag, 1994] and defined as a negative lowerbound which is not 5-prunable) to all negative instances neither 5-prunable nor g-prunable.

Two aspects of INBF [Smith and Rosenbloom, 1990] can be compared to ours. In the specific-to-general search INBF drops all positive examples, because no backtracking is needed in searching specific-to-general in a conjunctive tree-structure language. Using Theorem 4, our approach would also drop all positive lowerbounds, except one (which would then coincide with *s),* because any two positive examples will have only one least upperbound. In the general-to-specific search INBF processes and then forgets all near-misses w.r.t. *s.* Its maximally general hypothesis *upper* is only kept consistent with all positive examples and all near-misses, so no backtracking is needed. Our notion of a near-miss generalizes this approach, by converting all negative lowerbounds to positive upperbounds, and considering their greatest lowerbound (i.e., *upper)* as a positive upperbound.

[Hirsh, 1990] informally describes a technique of "skipping data that do not change the versionspace" in the context of the Incremental Versionspace Merging algorithm. Intersecting a versionspace *VS1* with *VS2,* and then with a subset *VS2'* of *VS2* will always yield the latter intersection. Consequently the first intersection operation was not necessary. In our framework, we more formally describe the approach using the notions 5-prunable and g-prunable, we relate these notions to the concept of near-misses and to INBF, and provide a framework to automatically generate new information elements.

7 Conclusion

We have introduced the notions of 5-prunable and *g*-prunable instances and a generalized notion of near-miss. Using these we identified redundant instances. We also introduced automatically created instances, that made other instances redundant without any loss of information. This resulted in data compaction. Furthermore, as in [Smith and Rosenbloom, 1990], this paper shows that near-misses (and their dual counterpart) play a very important role in converging towards the target concept.

We have also shown how 5-prunable and g-prunable instances can be discarded in the framework of Iterative Versionspaces, without losing the most important properties of the ITVS algorithm.

Acknowledgements

Luc De Raedt is supported by the Belgian National Fund for Scientific Research and by the ESPRIT project Nr. 6020 on ILP, and is co-financed by the Flemish Govemment under contract no. 13/0/4.

References

[Bundy *et a*/., 1985] A. Bundy, B. Silver, and D. Plummer. An analytical comparison of some rule-learning programs. *Artificial Intelligence,* 27:137-181, 1985.

[Hirsh, 1990] H. Hirsh. *Incremental Version-Space Merging: A General Framework for Concept Learning.* Kluwer Academic Publishers, 1990.

[Hirsh, 1992] H. HirBh. Polynomial-time learning with version spaces. In *Proceedings of AAAI-92,* pages 117-122. AAAI Press, 1992.

[Mellish, 199l] C. Mellish. The description identification problem. *Artificial Intelligence,* 52:151 - 167, 1991.

[Mitchell, 1978] T.M. Mitchell. *Version Spaces: An Approach to Concept Learning.* PhD thesis, Stanford University, 1978.

[Mitchell, 1982] T.M. Mitchell. Generalization as search. *Artificial Intelligence,* 18:203-226, 1982.

[Sablon *et a*/., 1994] G. Sablon, L. De Raedt, and M. Bruynooghe. Iterative versionspaces. *Artificial Intelligence,* 69:393-409, 1994.

[Sablon, 1995] G. Sablon. *Iterative Versionspaces with an application in Inductive Logic Programming.* PhD thesis, Dept. of C.S., KULeuven, May 1995.

[Sebag and Rouveirol, 1994] M. Sebag and C. Rouveirol. Induction of maximally general clauses consistent with integrity constraints. In *Proceedings of ILP94,* volume 237 of *GMD-Studien,* pages 195-215. GMD, 1994.

[Sebag, 1994] M. Sebag. Using constraints to building version spaces. In *Proceedings of ECML94,* volume 784 of *Lecture Notes in Artificial Intelligence,* pages 257-286. Springer-Verlag, 1994.

[Smith and Rosenbloom, 1990] B.D. Smith and P.S. Rosenbloom. Incremental non-backtracking focusing: A polynomially bounded generalization algorithm for version spaces. In *Proceedings of AAAI-90,* pages 848-853. AAAI Press, 1990.

[Winston, 1975] P.H. Winston. Learning structural descriptions from examples. In P.H. Winston, editor, *Psychology of Computer Vision.* The MIT Press, 1975.