CHAPTER 2

The Induction of Logical Conjunctions

In Chapter 1 we examined three basic approaches to representing and interpreting conceptual knowledge. The most familiar of these involved logical definitions, which use an 'all or none' match to determine the membership of instances. In this chapter we consider the special case of *logical conjunctions*, which represent concepts as the 'and' of some simple tests. Much of the early work in machine learning focused on the induction of such concepts, and many algorithms for more challenging tasks build upon methods devised for this simpler task. Thus, we will go into some detail about a range of techniques, since they will prove useful in later chapters.

We begin by discussing some general issues in the induction of logical conjunctions, and then turn to some nonincremental algorithms for addressing this task, focusing first on exhaustive methods and then on techniques that use heuristics to constrain search. Next we turn to incremental approaches to the same problem, again presenting both nonheuristic methods and heuristic variants. Finally, we examine the use of genetic algorithms for logical induction, which organize their search along quite different lines from other methods.

2.1 General issues in logical induction

Before examining particular algorithms for the induction of logical conjunctions, we should briefly consider some issues that cut across the various approaches. First we review the representational and performance assumptions that underlie work in this paradigm. After this we present a more precise statement of the induction task in terms of in-

puts and outputs. Finally, we consider the partial ordering that exists among concepts and their logical descriptions, which plays a central role in most of the learning methods we examine in this chapter.

2.1.1 Representation and use of logical conjunctions

The induction algorithms described in this chapter are designed to find a conjunctive description for a single concept C that covers positive instances of C and that fails to cover negative instances. They represent a concept as a logical conjunction of Boolean features, values of nominal attributes, limits on the values of numeric attributes, or some combination of them.[1] We will refer to each component of such a conjunction as a *condition* or a *test*.

One can use the resulting concept description D to classify new instances by matching D against the instance description. Briefly, if the instance matches all conditions in the concept description, it is labeled as a positive instance of the concept C; otherwise it is labeled as negative. This all-or-none flavor is central to the methods we will consider in this chapter, and it has important implications for the learning algorithms themselves.

These representational and performance assumptions combine to produce decision boundaries that enclose singly connected regions of the instance space; that is, for any pair of positive instances, there exists a path that traverses only other positive instances. Moreover, these regions take the form of single hyperrectangles with surfaces orthogonal to the axes of relevant attributes and parallel to the axes of irrelevant ones. For these reasons, methods that induce logical conjunctions are sometimes described as using an *orthogonal rectangle* bias, whereas others have used the term *axis parallel* bias. This idea is best seen with some examples.

Consider a nominal domain that involves three attributes for describing the appearance of cells – the number of `tails` (`one` or `two`), the `color` of the cell body (`dark` or `light`), and the number of `nuclei` (`one` or `two`). In some later examples, we will include a fourth attribute involving the thickness of the cell `wall`, which can be `thick` or `thin`. Figure 2-1 (a) shows one cell taken from a patient with the dreaded disease

1. The same framework also applies to relational descriptions, but we delay their discussion until Chapter 8.

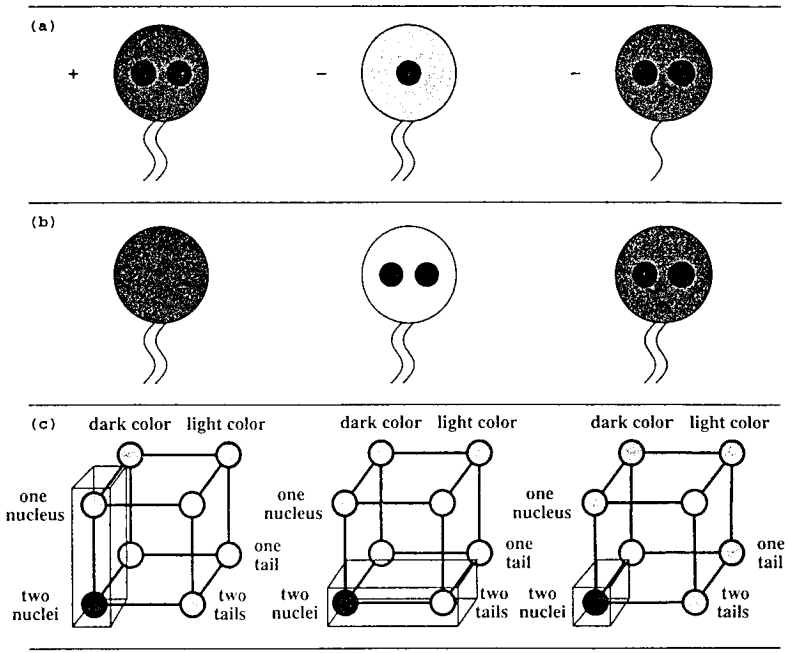

Figure 2-1. Example of a nominal domain involving cells with three attributes, showing (a) one positive (+) and two negative (-) training instances, (b) three logical conjunctions that match the positives but not the negatives, and (c) the axis parallel decision boundaries that each conjunction produces in the instance space.

LETHARGIA (a positive instance of this concept), along with two cells taken from a healthy person (negative instances). For example, the positive case has `two nuclei`, `two tails`, and a `dark color`, whereas the first negative case has `one nucleus`, `two tails`, and a `light color`.

Figure 2-1 (b) graphically depicts three conjunctions that, when combined with a logical matcher, are consistent with these instances, with missing features indicating that they play no role in the description. For example, the abstract cell on the left, which has no `nuclei`, corresponds to the conjunction `two tails` $\wedge$ `dark color`. Similarly, the central cell, which has no `color`, specifies the conjunction `two nuclei` $\wedge$ `two tails`. The rightmost description, which includes all three of the attributes, indicates the very specific conjunction `two nuclei` $\wedge$ `two tails` $\wedge$ `dark color`.

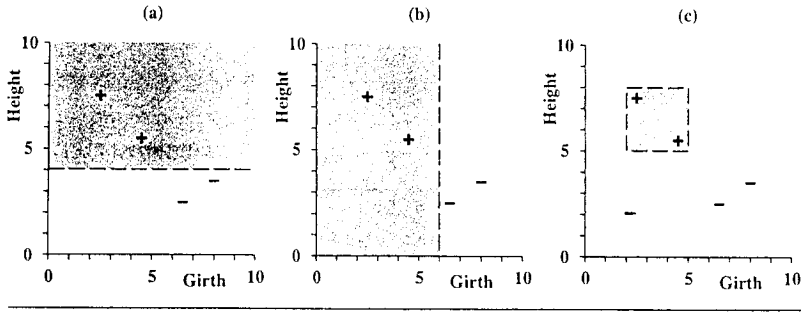

Figure 2-2. Positive and negative instances in a numeric domain involving two attributes, a person's `height` and `girth`. The most general logical conjunctions consistent with the training data, specified in terms of rectangular decision boundaries, appear in (a) and (b), whereas (c) depicts the most specific consistent conjunction.

Because each attribute in the cell domain has two possible values, they are effectively Boolean features. Thus, we can draw the instance space as a cube, with each axis representing a feature and each vertex representing a particular instance. Figure 2-1 (c) uses this notation to show the hyperrectangles that correspond to the logical conjunctions in (b). This clearly shows that all three conjunctions cover the positive instance (the dark vertex) but not the negative instances (white vertices), and also shows that they make different predictions about unseen cases (gray vertices). For instance, the second and third conjunctions classify a cell with `one nucleus`, `two tails`, and a `dark color` as negative, whereas the first conjunction predicts a positive label. Given the observed data alone, there is no reason to prefer any one of these descriptions over the others.

Now let us consider a second domain that involves two numeric dimensions, say `height` and `girth`, along with a concept description language that allows only conditions of the form `(> height x)`, `(> girth x)`, `(< height x)`, and `(< girth x)`, where `x` is a specific integer. Thus, a logical conjunction for this domain will contain between zero and four such conditions. Figure 2-2 depicts two positive instances (+) and two negatives (-) from this domain, along with the decision boundaries for three conjunctions that, when interpreted by a logical matcher, cover the former but not the latter. In particular, (a) shows one of the two most general conjunctions of this sort, and (b) shows the second alternative, which is no more general than the other. In contrast, (c) depicts the most specific logical conjunction consistent with the given instances. As in the nominal domain, decision boundaries take the form of rectangles with each side parallel to some axis in the instance space.

2.1.2 The task of inducing logical conjunctions

Now that we have described the representation and use of logical conjunctions, we can formulate the task of their induction from experience:

- *Given*: A set of positive training instances for the class C;
- *Given*: A set of negative training instances for the class C;
- *Find*: A logical conjunction that, to the extent possible, correctly classifies novel test cases.

To restate, algorithms that address this task receive as input a set of positive and negative instances for some class. In response, they must generate an intensional description – stated as a logical conjunction – for use in classifying future instances.

This learning task assumes that one can actually describe, or at least approximate, the concept using a conjunction of simple conditions, such as those we saw in Figures 2-1 and 2-2. This in turn assumes that the language for describing concepts includes all relevant terms. Note that the goal is *not* necessarily to find a conjunction that perfectly partitions the training cases into positive and negative instances, but to induce a description that accurately classifies novel instances. Some early work on machine learning took the former view, as do some of the methods we examine later in the chapter. However, such methods encounter difficulties in domains that contain noise or in those for which a conjunction only approximates the target concept.

One can use a simple trick to extend the above scheme to multi-class induction, which requires one to learn descriptions for a set of N concepts rather than for a single concept. For each class C, one treats all instances of the other $N-1$ classes as negative instances of C. Thus, one repeats the basic induction task N times, once for each class. However, this approach can produce descriptions that do not match (and thus fail to classify) some instances and that all match (and thus conflict) on other instances. We will return to this issue in Chapter 5, but for now we will restrict our attention to the induction of single conjunctive concepts.

Table 2-1. The EGS algorithm: Nonincremental general-to-specific breadth-first search for logical conjunctions.

```
Inputs: PSET is the set of positive instances.
        NSET is the set of negative instances.
        CSET is a set of consistent descriptions.
        HSET is a set of overly general descriptions.
Output: The set of most general logical conjunctions
        that are consistent with the training data.

Top-level call:  EGS(PSET, NSET, { }, {{ }})

Procedure EGS(PSET, NSET, CSET, HSET)

For each description H in HSET,
    If H does not match all members of PSET,
    Then remove H from HSET.
    If H does not match any members of NSET,
    Then remove H from HSET and add H to CSET.
If HSET is empty,
Then return CSET.
Else let NEWSET be the empty set.
     For each description H in HSET,
     Let SPECS be all one-condition specializations of H that
         fail to match more negative instances than does H.
     For each description S in SPECS,
         If CSET contains no description more general than S,
         Then add S to NEWSET.
     Return EGS(PSET, NSET, CSET, NEWSET).
```

This process continues until the algorithm reaches a level at which the set of overly general descriptions is empty. At this point, it returns the set of consistent descriptions as summaries of the instances it was given as input. Each of these descriptions is guaranteed to cover all of the positive instances and none of the negatives. Moreover, EGS will find the set of all most general descriptions that are consistent with the training data. Other consistent descriptions may exist in the space of conjunctive descriptions, but they will be more specific than at least one of the induced descriptions. Thus, the logical conjunctions generated by EGS include only those attributes that are necessary for discriminating between positive and negative instances of the concept.

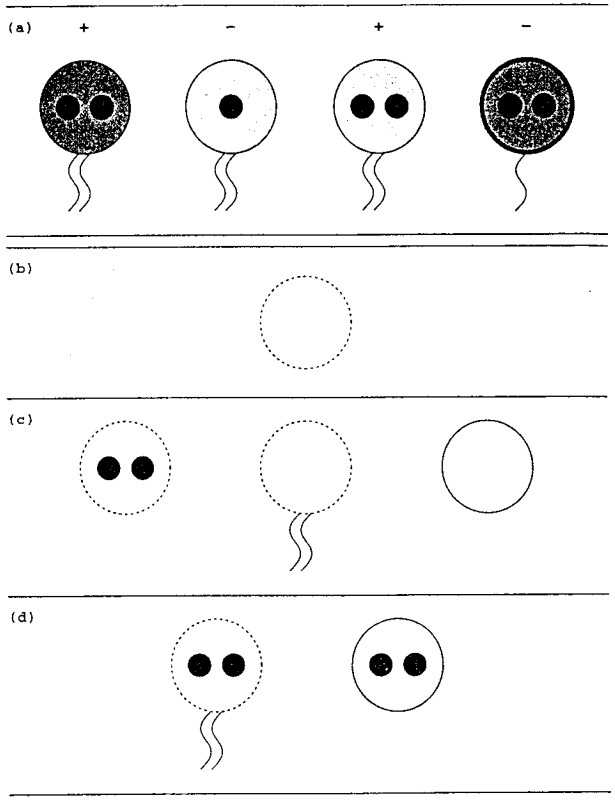

Figure 2-3. Descriptions considered by the EGS algorithm in a nominal domain involving four attributes, given the training instances in (a), at (b) the first level of search, (c) the second level, and (d) the third level.

2.2.2 Examples of EGS in operation

To further clarify the nature of the algorithm, let us consider its behavior on some training instances from the four-attribute cell domain shown in Figure 2-3 (a). EGS begins with the most general description possible, in this case depicted by the featureless cell in (b). The next level of search, in Figure 2-3 (c), shows three hypotheses that provide some improvement over the initial description, in that each eliminates matches on one of the two negative instances. One such description indicates that the cell must have `two nuclei`, another specifies `two tails`, and the

2.1.3 Partial ordering of classes and concepts

As we saw in Chapter 1, the space of possible conceptual classes is partially ordered by generality. That is, if class A includes all the instances of class B, along with other instances as well, then A is strictly more general than B. This space is bounded at one extreme by the most general class possible, which contains all possible instances, and at the other by the set of most specific classes, which contain one instance apiece.

This partial ordering plays an important role in most methods for inducing logical concept descriptions. The reason is that, given such a concept representation, there is typically a direct relation between the relative generality of two classes and their relative concept descriptions. For example, given a simple conjunctive language for concepts, dropping a nominal condition from concept A's description produces a strictly more general concept, whereas adding a nominal condition to A produces a more specific concept. The same holds for conditions that involve numeric attributes, although one can also increase generality by extending a numeric range or decrease it by restricting such a range.

This simple relation between the 'syntax' and 'semantics' of logical concepts provides two benefits. First, it suggests some obvious operators for moving through the space of concept descriptions, as we will see shortly. Second, it means that the partial ordering on extensional classes carries over to concept descriptions, and one can use this ordering to constrain and organize the search for useful concepts. With the exception of genetic algorithms, which we discuss at the end of the chapter, nearly all research on the induction of logical concepts has taken advantage of this insight.

2.2 Nonincremental induction of logical conjunctions

Given a generality ordering on the space of concept descriptions, two obvious search organizations suggest themselves – one can move from general descriptions to specific ones or from specific descriptions to more general ones. In this section we focus on a nonincremental approach to logical concept induction that carries out exhaustive breadth-first search in one direction and then we briefly consider an analogous algorithm that operates in the opposite direction. In closing, we discuss some drawbacks of such exhaustive methods.

2.2.1 The EGS algorithm

We will begin our study of logical induction with a nonincremental method that carries out a general to specific search through the space of concept descriptions. Table 2-1 presents pseudocode for the EGS (exhaustive general-to-specific) algorithm, which is provided at the outset with a set of positive instances (which the induced description should cover) and a set of negative instances (which it should not). In addition, the recursive algorithm inputs a set of concept descriptions that cover all of the positives and none of the negatives; this set is initially empty but tends to grow with the depth of recursion. EGS also carries along a set of overly general descriptions, which cover all positive cases but also some negatives. This set is initialized to contain a single 'null' description with no conditions, which covers all training instances.

The algorithm carries out a breadth-first search through the space of concept descriptions. At each level of the search (i.e., each level of the recursion), EGS considers all specializations of overly general descriptions that involve the addition of one condition. For a nominal attribute not already used in the description, this involves generating one alternative condition for each value of that attribute. For a numeric attribute, one can generate a minimal specialization by iterating through all integer values, then stopping upon reaching an integer that excludes a positive instance. This iteration begins with the lowest observed attribute value for < tests and with the highest value for > tests.[2]

For each such specialized description `H`, EGS checks to ensure that `H` still covers all known positive instances. If not, the algorithm drops this description from consideration as overly specific, which also effectively removes all of its descendants in the space of hypotheses, since it will never consider them (unless it reaches them by another path). For each remaining description `H` not rejected in this manner, EGS checks to see whether `H` still covers any of the negative instances. If not, the method adds `H` to its set of consistent descriptions (which it carries along as it continues its search). If a description still covers some negative instances, EGS adds it to the set of overly general descriptions for the current level, which it then specializes further.

2. We assume the learner has access to the information needed to generate these specializations – the names of features and their possible values. If this information is not provided by the user, a system can generate it by examining all of the training instances.

third refers to a `thin wall`. Other descriptions also specialize the initial one, but fail to cover both positive instances, and thus are rejected.

At the final level of search, EGS has arrived at two descriptions that cover both positive instances but neither negative instance. These are specializations of descriptions at the previous level; one specifies that the cell must have `two nuclei` and `two tails`, whereas the other requires `two nuclei` and a `thin wall`. The algorithm cannot further discriminate between these competitors with the data available, and the set of overly general descriptions is empty, so EGS returns both as its result.

Figure 2-4 presents an example of EGS's operation in the numeric height/girth domain, in this case involving two positive and three negative instances. As before, the initial hypothesis set contains only one description, the most general possible, and again the next level of the search tree includes three specializations, each of which happens to eliminate one negative instance. Note that one of these places an upper limit on `height`, whereas another places a lower limit on the same dimension.

The third level of the search also contains three hypotheses, each of which contains two conditions that rule out two of the negative training instances. Moreover, each is a specialization of two descriptions at the previous level. The fourth and final level contains a single hypothesis that includes all three conditions mentioned in the previous level, and thus is a specialization of all descriptions at that level. Because this hypothesis covers both positive instances and none of the negatives, and because the set of overly general descriptions is empty, EGS halts and returns it as the result. This example is somewhat misleading in that one can obtain the same result by following a single path through the space of descriptions. However, the algorithm is designed to find *all* maximally general logical conjunctions.

2.2.3 Drawbacks of exhaustive methods

Despite its simplicity, one can easily identify some major drawbacks of the EGS technique, the most obvious involving its computational complexity. Even in purely Boolean domains, the worst-case size of the hypothesis set grows rapidly with the number of features. We can ignore the degenerate case in which all instances are negative, but we cannot ignore situations in which many features are relevant, leading to training sets containing only a few positive instances, with negatives for

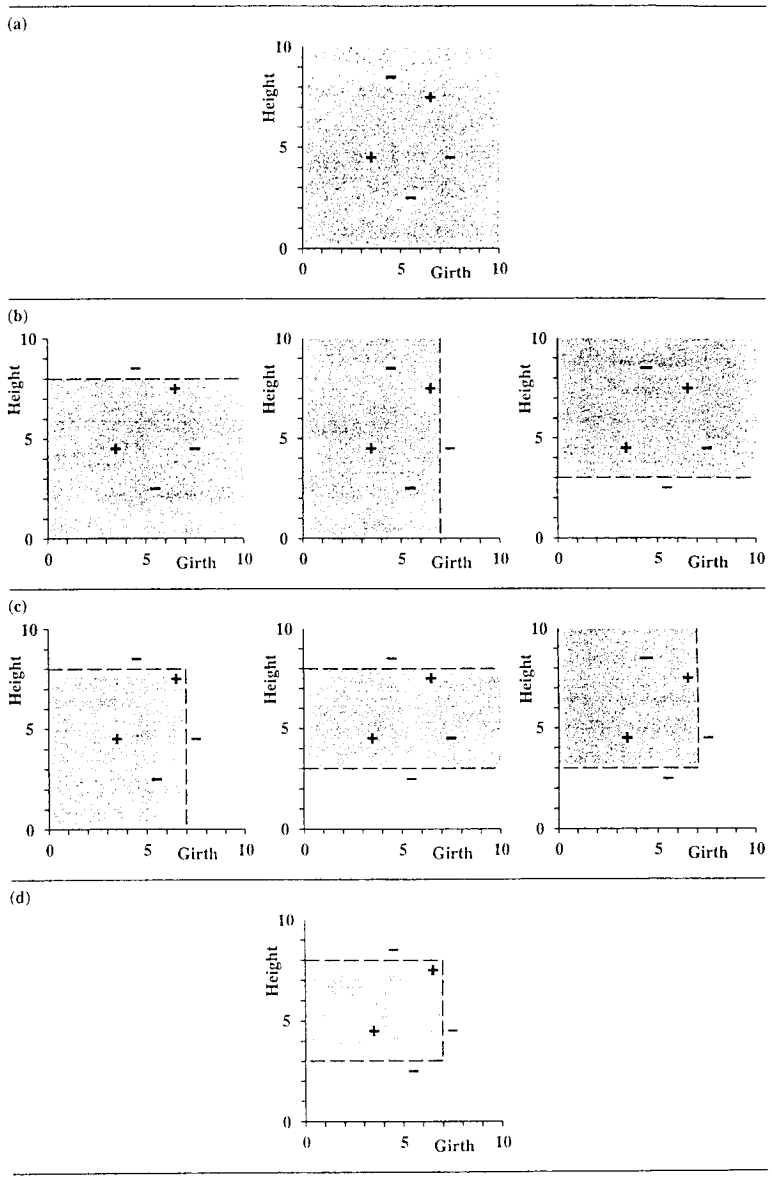

Figure 2-4. Descriptions considered by the EGS algorithm in the numeric height/girth domain at (a) the first level of search, (b) the second level, (c) the third level, and (d) the final level.

the remainder. In such domains, given k relevant Boolean features, the hypothesis set can grow to size $\cdot\binom{k}{\frac{k}{2}}$. The algorithm reaches this point halfway to the correct description, after which the set decreases in size until the end of the run. The basic problem is that EGS carries out an exhaustive breadth-first search and thus is a captive of the exponential growth that typifies such methods.

The EGS algorithm aims to find the most general possible concept descriptions, but there is no inherent reason for this inductive bias. An alternative, the ESG (exhaustive specific-to-general) method, finds the most specific logical conjunction consistent with the training data. For the data in Figure 2-3, this is the conjunction `two nuclei` $\wedge$ `two tails` $\wedge$ `thin wall`, which is strictly more specific than both descriptions generated by EGS. The ESG algorithm has some clear computational advantages over the general-to-specific technique. In attribute-value domains, ESG simply computes the most specific description without any need for search; in fact, it does not even examine the negative instances, since they do not affect the outcome. For each Boolean feature or nominal attribute, the method finds the value shared by all positive training instances; for each numeric attribute, it simply finds the minimum and maximum values.

Provided that there exists a logical conjunction consistent with the training data, the ESG technique will find the maximally specific description without the need for search. However, as we discuss in Chapter 8, these advantages disappear in relational domains, where ESG requires both search and negative instances. Moreover, a belief in Occam's razor has led many researchers to prefer induction methods that favor general descriptions over more specific ones that cover the same training data. Thus, although the ESG algorithm suffers less from combinatorial problems than EGS, it does not provide a general solution.

Another limitation of both methods lies in their inability to handle situations in which no logical conjunction is consistent with the training data. In noisy domains, even one mislabeled class or feature can seriously derail either technique in its search for consistent descriptions. For example, if the second negative instance in Figure 2-3 had been mistakenly labeled positive, EGS would have produced the single overly general description `two nuclei`. In some cases the effect of noise can be even worse, causing the method to eliminate all hypotheses from consideration. Similar problems arise in domains where the target concept is nearly but not perfectly described by a logical conjunction. These problems suggest the need for more robust algorithms that make weaker assumptions about the nature of the training data.

2.3 Heuristic induction of logical conjunctions

Heuristic approaches to concept induction hold promise for overcoming some disadvantages of the EGS and ESG algorithms. Rather than carrying out a breadth-first search through the space of concept descriptions, heuristic methods carry out more selective searches that, in many cases, find satisfactory solutions while considering only a fraction of the space. In this section we focus on *beam search*, a control structure that retains a fixed number of alternatives at each level of the search, using an evaluation function to select the best ones. We will see examples of other heuristic approaches later in the chapter. As before, we first examine search from general to specific descriptions, then briefly consider a specific to general organization.

2.3.1 The HGS algorithm

Table 2-2 presents pseudocode for the HGS (heuristic general-to-specific) algorithm, a more selective analog of the EGS technique. Like its exhaustive relative, this method accepts as input sets of positive and negative instances, along with two sets of concept descriptions, one initialized to the empty set and the other to a set containing a single 'null' description with no conditions. However, HGS interprets these last two inputs differently from EGS, with the first (`CLOSED-SET`) specifying candidate descriptions that cannot be improved by further specialization and the second (`HSET`) indicating descriptions that might still be improved. The two algorithms also differ in the details of their search processes.

At each level of its search, HGS considers all specializations of descriptions in `HSET` that involve the addition of one condition. For each such specialized description `S`, HGS uses an evaluation function to measure `S`'s degree of fit to the training data. If the score for `S` is greater than the score for its parent `H`, the algorithm adds the hypothesis `S` to a set of new descriptions (`OPEN-SET`) that it should consider for further specialization later. If none of the specializations of `H` score better than their parent, then HGS adds `H` to `CLOSED-SET`, since it cannot be improved further.

Table 2-2. The HGS algorithm: Nonincremental beam search for a logical conjunction, directed by an evaluation function Score.

```
Inputs: PSET is the set of positive instances.
        NSET is the set of negative instances.
        CLOSED-SET is a set of locally optimal descriptions.
        OPEN-SET is a set of unspecialized descriptions.
Output: A logical conjunction that matches most of PSET
          and that does not match most of NSET.
Params: Beam-Size is the number of descriptions in memory.

Top-level call:  HGS(PSET, NSET, { }, {{ }})

Procedure HGS(PSET, NSET, CLOSED-SET, HSET)

Let OPEN-SET be the empty set.
For each description H in HSET,
    Let SPECS be all one-condition specializations of H.
    Let NEW-SET be the empty set.
    For each specialized description S in SPECS,
        If Score(S, PSET, NSET) > Score(H, PSET, NSET),
        Then add S to NEW-SET.
    If NEW-SET is empty,
    Then add the description H to CLOSED-SET.
    Else for each description S in NEW-SET,
         Add S to OPEN-SET.
         For each description C in CLOSED-SET,
             If S is at least as specific as C,
             Then if Score(C,PSET,NSET) > Score(S,PSET,NSET),
                  Then remove S from OPEN-SET.
                  Else remove C from CLOSED-SET.
If OPEN-SET is empty,
Then return the highest-scoring member of CLOSED-SET.
Else let BEST-SET be the Beam-Size highest-scoring
         members of the union of OPEN-SET and CLOSED-SET.
     Let CLOSED-SET be the members of CLOSED-SET in BEST-SET.
     Let OPEN-SET be the members of OPEN-SET in BEST-SET.
     Return HGS(PSET, NSET, CLOSED-SET, OPEN-SET).
```


After considering all specializations of the descriptions in HSET, the algorithm checks to see whether any of them have scored better than their parents. If not, HGS simply returns the parent description with the highest score; otherwise, it continues its search. However, before continuing to the next level, the algorithm must first reduce its hypothesis set to manageable size. To this end, HGS selects the Beam-Size highest-scoring members of CLOSED-SET and OPEN-SET, then calls itself recursively, with the reduced OPEN-SET taking on the role of HSET. The algorithm removes other candidates from memory, even if they might ultimately lead to useful descriptions; thus, it trades off optimality for constrained search.

The descriptions generated by HGS are *not* guaranteed to cover all positive instances and no negative instances, nor would this be desirable in domains where the data contain noise or can only be approximated by a logical conjunction. Nor are the descriptions it generates guaranteed to be minimal, in that some strictly more general descriptions may exist down a rejected branch that would have given a better score. These are the prices paid for carrying out an efficient heuristic search rather than an intractable exhaustive one, and experimental studies suggest that, in many domains, such heuristic methods work well despite their superficial drawbacks.

2.3.2 Heuristic evaluation of descriptions

The behavior of HGS depends heavily on the choice of the evaluation function Score. As shown in Table 2-2, this function takes three arguments – the description being evaluated, the set of positive instances, and the set of negative instances. In general, the function's score should increase both with the number P_c of the positive instances covered and with the number $N_{\bar{c}}$ of negatives *not* covered. A good metric should also take into account the total number of positive instances P and negative instances N.

One simple measure that satisfies both of these constraints uses the expression

$$\frac{P_c + N_{\bar{c}}}{P + N} \quad ,$$

which ranges between 0 (when no positives and all negatives are covered) and 1 (when all positives and no negatives are covered). This ratio measures the overall classification accuracy on the training data.

More sophisticated approaches invoke statistical or information theoretic measures, but even a simple measure like $P_c + N_{\bar{c}}$ (the number of covered positives plus the number of uncovered negatives) provides significant direction to the HGS algorithm.

We should briefly consider the relation between the EGS and HGS algorithms. Suppose we measure the score for a description by $(P_c + N_{\bar{c}})/(P + N)$, as suggested above, and set the `Beam-Size` to infinity. In this case, the heuristic method simulates the exhaustive EGS algorithm, searching for all minimal concept descriptions that cover all positive instances and none of the negative instances. In other words, EGS can be viewed as a special case of HGS that results from extreme settings of the latter's parameters. Another important special case occurs when `Beam-Size` is set to 1, which causes HGS to behave as a *greedy* method that selects the single best hypothesis at each level of the search.

2.3.3 The HSG algorithm

Although HGS begins with general descriptions and moves toward more specific ones as needed, one can also carry out heuristic search in the opposite direction using a heuristic specific-to-general method (HSG). However, because this algorithm does not assume there exists a conjunctive target concept consistent with the training data, it lacks the advantages that the simpler ESG method has over the exhaustive EGS. To handle more realistic induction problems, HSG must use the same techniques as HGS – a beam search directed by an evaluation function over both positive and negative instances.

Naturally, the HSG algorithm differs from HGS in its initialization, in the direction of search, and in the characteristics of the resulting descriptions. The new method begins with one of the most specific possible descriptions, each of which corresponds directly to a positive training instance. HSG selects one of these instances (say at random) as its *seed*, which it uses to generate the initial hypothesis from which to start its specific-to-general search.

As in HGS, the new scheme carries along an `OPEN SET` of hypotheses, but in this case the descriptions are not general enough. Thus, each step through the hypothesis space involves generalizing an overly specific description by dropping or generalizing a single condition, rather than by specialization. HSG uses an evaluation metric to order descriptions in terms of their fit to the training data, selects the best `Beam-Size` of these, and iterates. Again, the method halts when generalizing the hypotheses in `OPEN SET` produces no higher-scoring descriptions than their parents, in which case it returns the best of those parents.

Like HGS, the HSG algorithm has a constant memory cost. This means search remains under control, but that it will not always generate the optimal descriptions for a given data set. The technique should be relatively robust with respect to noise and target concepts that are only approximately conjunctive. An important exception involves the seed used to initialize search. If this positive instance has been mislabeled or falls outside the target region, HSG will find a description that covers this instance but few other positive cases of the concept. One response is to run the algorithm multiple times with different seeds; another is to select a seed that falls near the central tendency of the positive instances.

2.4 Incremental induction of logical conjunctions

In Chapter 1 we argued that, in some situations, incremental learning has advantages over nonincremental methods. To this end, we now turn to incremental approaches to the induction of logical conjunctions. As in the nonincremental case, one can search the concept space from general to specific or vice versa. In this section we focus on the first approach, then briefly consider a specific-to-general scheme and some methods that combine both ideas. In each case, we assume a breadth-first search organization, since that is the simplest for tutorial purposes. We also assume that the goal is to induce concept descriptions that cover all of the positive training instances but none of the negatives, although in closing we discuss some drawbacks of this approach.

2.4.1 The IGS algorithm

Table 2-3 presents pseudocode for IGS, an incremental general-to-specific method for the induction of logical conjunctions. The algorithm processes training cases one at a time, retaining a set of hypotheses that cover all the positive instances it has seen so far but none of the observed negative instances. The method initializes this set to one containing the most general possible description.

Upon encountering a new negative instance, IGS checks each description to determine if it covers the instance. If so, the method abandons

Table 2-3. The IGS algorithm: Incremental general-to-specific search for a conjunctive concept description.

```
Inputs: ISET is the set of classified training instances.
Output: The set of most general logical conjunctions
        that are consistent with the training data.

Procedure IGS(ISET)

Let G be the most general possible description.
Let the set of current hypotheses HSET be {G}.
Let the set of processed positives PSET be the empty set.
For each instance I in ISET,
    If I is a negative instance,
    Then for each hypothesis H in HSET that matches I,
            Delete H from HSET.
            Find the set GSET of most general specializations
                 of H that will not match I.
            Let HSET be the union of HSET and GSET.
            For each hypothesis H in HSET,
                If HSET contains a G more specific than H,
                Then remove H from HSET.
            For each hypothesis H in HSET,
                If PSET contains an instance
                   that H fails to match,
                Then remove H from HSET.
            If HSET is empty, then return the empty set.
    Else if I is a positive instance,
         Then add I to the set PSET.
              For each hypothesis H in HSET,
                  If H fails to match I,
                  Then remove H from HSET.
              If HSET is empty, then return the empty set.
Return HSET.
```

the offending hypothesis (since it is overly general) and replaces this description with all minimally more specific variants that fail to cover the instance. However, it abandons any descriptions that fail to cover any positive instances it has seen in the past, as well as any descriptions that are strictly more general than others in the hypothesis set.

The learning algorithm takes quite different actions upon encountering a new positive instance. In such a case, IGS deletes all current hypotheses that *fail* to cover the instance (since they are overly specific). This cycle continues until the algorithm runs out of instances or until the hypothesis set becomes empty, indicating that no descriptions are consistent with the training data. In general, the IGS algorithm uses negative instances to drive the generation of candidate descriptions, and it uses positive instances to winnow the resulting set.

To summarize, IGS produces more specific descriptions when negative instances require this action and eliminates hypotheses when positive instances are not matched. No learning occurs when correct predictions are made, since the existing descriptions are performing as desired. The algorithm has no explicit means for deciding when it has acquired a 'final' concept definition, but at each point it will have in memory the most specific hypotheses that account for the data so far. Note that the algorithm must retain and reprocess an explicit list of positive examples, making it truly incremental only in its processing of negative instances.

2.4.2 An example of IGS in operation

As an example, let us consider IGS's behavior on the set of instances we presented earlier for the cell domain. Figure 2-5 shows the training instances in the order presented, along with the hypothesis set that results after the algorithm processes each one. As noted above, IGS begins with the most general description possible, in this case an abstract cell body with no features specified. After this initialization, the algorithm processes the first positive training instance, which leaves the hypothesis set unaltered (since it is correctly matched by the only member).

The next training instance is negative and, since the initial hypothesis matches this case, IGS must make it more specific. There are two ways to accomplish this and still cover the positive instance.[3] The first involves adding the condition `two nuclei`; the other involves adding the constraint `dark color`. Note that these descriptions are no more specific than necessary to avoid matching the negative instance.

Suppose the algorithm next encounters the second positive instance shown in Figure 2-5 (c). Because the hypothesis `two nuclei` matches

3. In some cases, there exists only one way to generate a more specific hypothesis. Negative instances that lead to this fortuitous situation are termed *near misses*. However, one cannot usually rely on their occurrence, so IGS can learn from 'far misses' as well, albeit converging on the ultimate concept description more slowly.

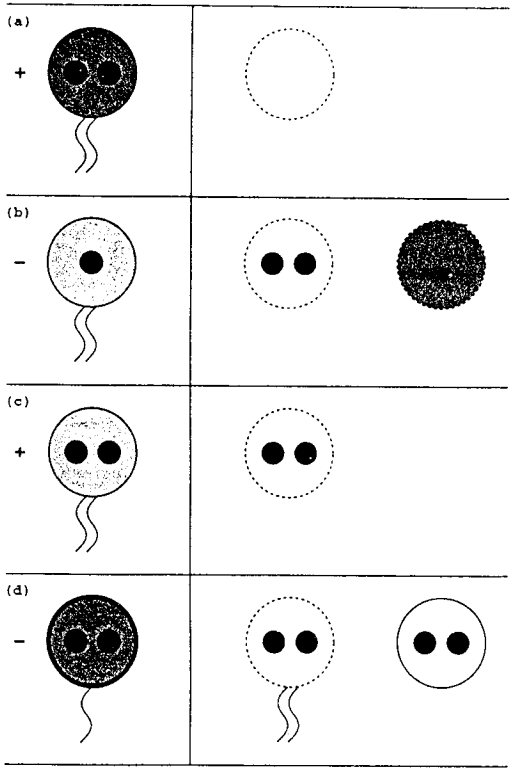

Figure 2-5. Descriptions considered by the IGS algorithm in the cell domain after (a) an initial positive instance, (b) an initial negative instance, (c) a second positive instance, and (d) a second negative instance. Instances are shown on the left, with the resulting set of hypotheses on the right.

this instance, IGS retains it in the set of descriptions. However, the other description, `dark color`, fails to match the instance, and thus is abandoned. Upon seeing a second negative instance, the algorithm finds that its remaining hypothesis matches this cell, and thus is still overly general. As a result, it generates two specializations that match neither negative instance but that match both positive cases. The first of these states that the cell must have `two nuclei` and `two tails`, whereas the other claims the cell must have `two nuclei` and a `thin wall`. Both descriptions are consistent with existing observations, although the algorithm would continue to make revisions if provided with new data.

2.4.3 The ISG algorithm

Now let us consider some other methods for the incremental induction of logical conjunctions. The ISG algorithm differs from IGS in that it carries out a specific-to-general search, but the basic structures of the two methods are quite similar. This technique also retains a set of descriptions that are consistent with the instances it has seen, but it initializes the hypothesis set to the first positive training instance.

Upon encountering a new positive instance, ISG checks each description to determine if it covers the instance. If not, the method deletes the offending hypothesis (since it is overly specific) and replaces it with all minimally more general variants that cover the instance. As we noted in Section 2.2.3, only one such description exists in nominal and numeric domains, but more than one can occur with relational languages. ISG also abandons any descriptions that cover any known negative instances, as well as any descriptions that are strictly more specific than others in the hypothesis set.

In contrast, when ISG encounters a new negative instance, it simply deletes all hypotheses that cover the instance. The algorithm takes this step for the same reason it removes new descriptions that cover previous negatives: search always moves from specific to general, so overly general descriptions cannot be corrected and must be abandoned. This overall cycle, in which ISG uses positive instances to generate descriptions and negative instances to eliminate them, continues until the algorithm stops receiving instances or until the hypothesis set becomes empty, indicating that no descriptions are consistent with the training data.

2.4.4 Bidirectional search techniques

One can also combine the specific-to-general and the general-to-specific approaches for searching the space of logical conjunctions. In one such scheme, either generalization or specialization acts as the primary induction operator, with the other giving the effect of backtracking when an hypothesis has gone too far in the primary direction. However, this approach makes less sense within an exhaustive search framework than it does within a heuristic one, like the one we describe in Section 2.5.

A second alternative, known as the *version space* algorithm, directly combines aspects of the IGS and ISG methods. This algorithm retains two sets of descriptions, one containing the most specific set of descrip-

tions that cover the data (S) and the other containing the most general set of such descriptions (G). These correspond to the sets retained by the ISG and IGS methods, respectively. The version space algorithm invokes a subroutine very similar to ISG to update the S set. In particular, when the method encounters a new positive instance that is not covered by any element of the S set, it replaces that element with one or more generalizations that do match the instance. Similarly, when any element of the G set matches a new negative instance, the version space technique uses a subroutine very similar to IGS to produce more specific versions of that element.

One notable difference is that the learner no longer need retain either positive or negative instances. The S set takes on the role of positive data in constraining search, whereas the G set takes on the role of negative instances. Rather than eliminating descriptions that fail to cover any positive instances, the modified IGS subroutine removes any member of the G set that is more specific than all members of S. Similarly, the modified ISG routine does not delete hypotheses that match any negative instance; instead, it removes any description in S that is more general than all elements in G. Thus, this approach is fully incremental in the sense that it need never reprocess any training instance.

The version space approach has two other interesting features. First, if the S and G sets converge on a single concept description, one knows that the induction task has been completed. Second, although the elements of S and G have identical forms to the descriptions we discussed for the IGS and ISG methods, their interpretation is rather different. Rather than representing hypotheses directly, members of the S and G sets act as *boundaries* on the space of descriptions that are consistent with the data. As the algorithm observes more instances, these boundaries become more constraining, until eventually they eliminate all but one logical conjunction. Basically, the version space method embodies a *least commitment* view of induction, in contrast to the state-space search view we have emphasized throughout the chapter.

2.4.5 Comments on incremental exhaustive methods

The methods we have discussed in this section provide insights into the incremental induction of logical conjuncts, and thus are useful for tutorial purposes. However, they suffer the same drawbacks as the exhaustive algorithms we considered in Section 2.2, making them unsuitable for many applied induction tasks.

We have seen that the IGS, ISG, and version space techniques process only one instance at a time, and that the latter method in particular does not store or reprocess any training cases. However, this statement is misleading, for all three algorithms must retain in memory all descriptions that are consistent with instances seen to date. In some domains, the size of these hypothesis sets can grow exponentially with the number of training instances, although certain training orders can minimize this effect. Hence, one cannot argue that such exhaustive methods are efficient in either space or time, even though they require little or no reprocessing of the instances themselves.

The three algorithms also encounter major difficulties when noise is present and when a conjunction only approximates the target concept. The source of this problem is their concern with finding descriptions that match all positive instances and none of the negative instances. A negative instance that is mislabeled as positive (one form of class noise) will cause ISG to produce overly general descriptions and it can lead IGS to eliminate a legitimate candidate. Similarly, a mislabeled positive instance will cause IGS to generate overly specific descriptions and it can force ISG to remove a perfectly acceptable hypothesis.

Because it uses both methods as subroutines, the version space algorithm suffers from class noise in all of these ways. The effect of attribute noise is different but still considerable. Extensions of the basic algorithm to handle limited amounts of class noise and bounded attribute noise in numeric domains lead to increases in the S and G sets, which may already be very large. In general, one should avoid exhaustive methods such as IGS, ISG, and version spaces when the training data contain noise and when the target concept is not perfectly conjunctive.

2.5 Incremental hill climbing for logical conjunctions

One goal of incremental approaches is to minimize the processing needed for each new training instance, and another is to reduce memory requirements. Yet we have seen that exhaustive approaches to incremental induction can be expensive on both fronts, even when they store and reprocess few instances. Fortunately, the incremental framework supports another alternative that is more efficient and also more consistent with theories of human learning. Rather than retaining a set of competing hypotheses in memory, the learner retains a single hypothesis and (potentially) refines this tentative description after each training instance.

We will refer to this general approach as *incremental hill climbing*, because of its similarity to the well-known search technique. In this section we examine the potential of this approach and consider one incremental hill-climbing method for the induction of logical conjunctions.

2.5.1 Learning as incremental hill climbing

Hill climbing is a classic AI search method in which one applies all possible operators, compares the resulting states using an evaluation function, selects the best state, and iterates until no more progress occurs. There are many variants on the basic algorithm, but these do not concern us here. The main advantage of hill climbing is its low memory requirement; because there are never more than a few states in memory, it sidesteps the high memory costs associated with search-intensive methods.

At each point in learning, an incremental hill-climbing algorithm retains only one knowledge structure, even though this structure may itself be quite complex.[4] Thus, hill-climbing learners cannot carry out a breadth-first search or a beam search through the space of descriptions, nor can they carry out explicit backtracking. They can only move 'forward', revising their single knowledge structure in the light of new experience.

The most important difference between hill-climbing learners and their traditional cousins lies in the role of input. Incremental learning algorithms are driven by new instances, and in the case of incremental hill-climbing methods, each step through the hypothesis space occurs in response to (and takes into account) some new experience. In other words, the learner does not move through the space of descriptions until it obtains a new datum. This mitigates some well-known drawbacks of hill climbing, such as the tendency to halt at local optima and a dependence on step size.

Recall that hill-climbing methods search an n-dimensional space over which some function f is defined. This function determines the shape of an n-dimensional surface, and the agent attempts to find that point with the highest f score. In traditional hill-climbing approaches, the function

4. We do not require that an incremental hill-climbing learner have an explicit evaluation function, or even that it carry out a one-step lookahead. One can replace this approach with a strong generator that computes the successor state from a new input, such as an observed instance.

f is static and thus the shape of the surface is constant. In systems that learn through incremental hill climbing, each new instance changes the form of f (the function used to evaluate alternative descriptions) and modifies the contours of the surface. Like H. A. Simon's wandering ant, the learner's behavior is controlled by the shape of its world. However, the hills and valleys of the hill-climbing learner's space are constantly changing as it gathers more information, altering the path it follows. This feature makes it unclear whether the limitations of traditional hill-climbing methods still hold in the context of incremental learning.

However, this dependence on new instances to control the search process also makes memory-limited learning methods sensitive to the order of instance presentation. Initial nonrepresentative data may lead a learning system astray, and one would like it to recover when later data point the way to the correct knowledge structure. Thus, some researchers have argued for the importance of *bidirectional* learning operators that can reverse the effects of previous learning should new instances suggest the need. Such operators can give incremental hill-climbing learners the effect of backtracking search without the memory required by true backtracking. Of course, the success of this approach remains an empirical question, but it shows enough promise for closer inspection.

2.5.2 The IHC algorithm

Table 2-4 summarizes an algorithm for incremental hill climbing (IHC) through the space of logical conjunctions. For each training instance I, this method checks to determine whether its current description H misclassifies I. If the classification is correct, then it takes no action, which means that it retains H as the current hypothesis. If an error occurs, IHC generates all minimal revisions to H that correct the mistake, then uses an evaluation function to rank the competitors on the last K training cases, which it has retained in memory. The algorithm selects the best-scoring description H′ and, if H′ fares as well or better than H on the stored instances, makes H′ the new hypothesis. Finally, IHC removes the oldest stored training case and stores the new one.[5] This process continues as long as training instances are available.

5. Another scheme would replace the current K training cases with an entirely new set of instances. This would decrease reprocessing of instances but might slow the overall learning rate.

Table 2-4. The IHC algorithm: Incremental hill climbing for a logical conjunction, directed by an evaluation function Score.

```
Inputs: ISET is a set of classified training instances.
Output: A logical conjunction.
Params: K is the number of instances retained in memory.

Procedure IHC(ISET)

Let H be the result of Initialize-Hypothesis.
For each training instance I in ISET,
    Let KSET be the sequence of K instances ending with I.
    If I is a positive instance,
    Then if H does not match I,
         Then let SSET be the set of most specific
                   generalizations of H that match I.
    Else (when I is a negative instance) if H matches I,
         Then let GSET be the set of most general
                   specializations of H that do not match I.
    For each member M of SSET or GSET, compute Score(M, KSET).
    Let H' be the member of SSET or GSET with the best score.
    If Score(H', KSET) ≥ Score(H, KSET),
       Then let H be H'.
Return the hypothesis H.
```

The algorithm responds differently to errors that involve negative and positive instances. If the current hypothesis H misclassifies a negative training case as positive, then IHC concludes that H is overly general and generates a set of minimally specific descriptions that do not match the instance. In contrast, if H misclassifies a positive instance as negative, the algorithm decides that its hypothesis H is overly specific and produces a set of minimally more general descriptions that match the instance. Thus, IHC carries out a bidirectional search through the space of logical conjunctions, with the direction of movement determined by the type of error.

The table does not specify mechanisms for generating revised descriptions, but they are basically the same as those discussed in Section 2.2.1. Producing a minimal generalization is straightforward. For nominal attributes, one simply removes those conditions from the current hypothesis H not shared by the positive instance. For numeric attributes, one extends numeric thresholds in H just enough to include the training case. As we have seen, for nominal and numeric representations, there is only one such generalization. However, the pseudocode allows for more than one minimal generalization, as can occur in the relational domains we discuss in Chapter 8.

The situation for generating specializations is more complicated. Even in nominal and numeric domains, there will typically be more than one minimal specialization; moreover, the current hypothesis H does not contain the information needed to generate this set. As we saw in Section 2.2.1, there are two basic responses. First, one can provide the learning algorithm with knowledge of attributes and their possible values, which it can then use to specialize H. Second, one can retain a positive training instance (say the most recent one), and use differences between this case and the misclassified negative instance to generate more specific variants of the hypothesis.

The pseudocode in Table 2-4 also neglects to mention the scheme for initializing the description. Obviously, one could set the initial H to the first positive instance, as in the ISG algorithm. Alternatively, one could initialize it to the most general possible description, as in the IGS method. Given knowledge of attribute values or predicates, one could even generate a random description and use this as the initial hypothesis. Different approaches would alter the number of training instances needed to reach the target concept but, at least in noise-free domains, should not affect the end result.

2.5.3 An example of IHC in operation

Figure 2-6 portrays the behavior of the IHC algorithm in response to a sequence of training instances, the same ones used in our previous examples but in a different order and with one repetition. The figure shows the descriptions considered after each training instance, along with their scores on the evaluation function from Section 2.3, $(P_c + N_{\bar{c}})/(P + N)$, for the three most recent instances. We assume the method begins with the most general hypothesis possible, a cell with no features specified.

Because the first instance is positive and this description matches it, IHC takes no action except to store the instance for future reference. The initial hypothesis also matches the second training case, but this one is negative. As a result, the algorithm considers four more specific descriptions, each with one additional condition, that do not match the

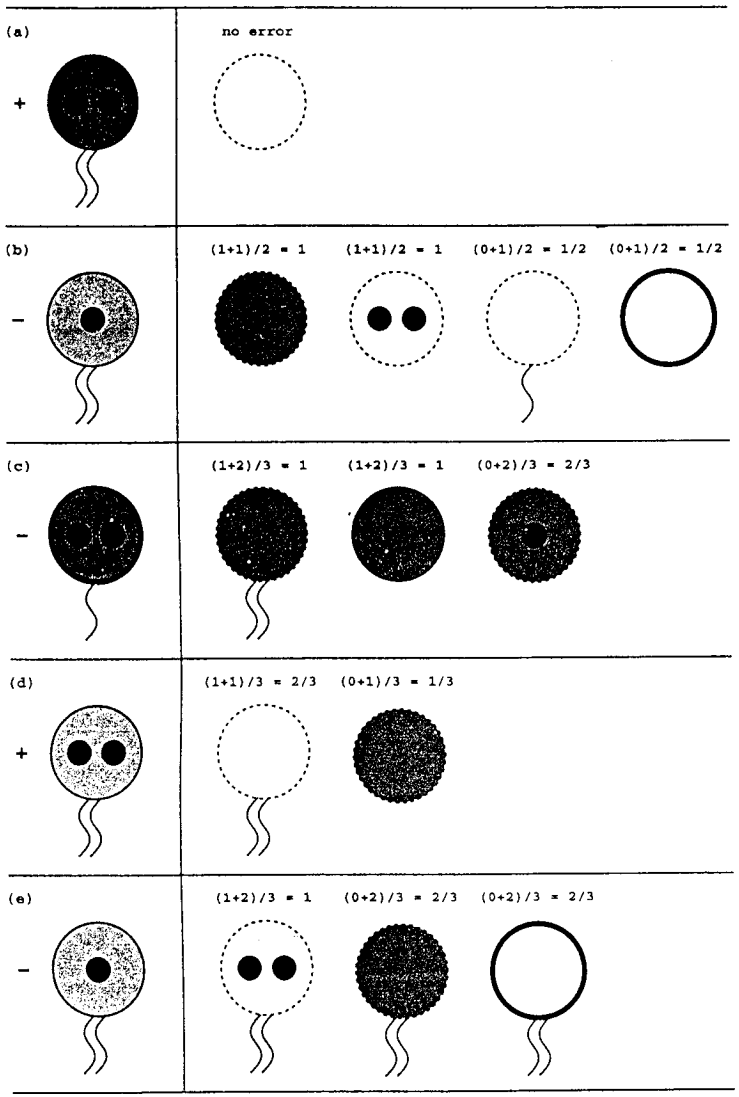

Figure 2-6. Successive training instances (a) through (e) in the cell domain (on the left) and the descriptions generated by the IHC algorithm in response (on the right), along with their scores on the evaluation function $(P_c+N_{\bar{c}})/(P+N)$, based on the current and previous two instances. The figure assumes that, in each case, IHC selects the highest-scoring (left-most) hypothesis for further consideration.

instance. IHC also computes the scores of these revised descriptions on the current and previous training instance. The best revisions are `dark color` and `two nuclei`, each with scores of $\frac{1+1}{1+1} = 1$. This score is at least as good as the score for the current hypothesis ($\frac{1}{2}$), so IHC selects one at random, say the first, as its new hypothesis.

The third instance, which is also negative, reveals that the revised description is still overly general. This time there are only three minimal specializations that do not match the training case. Two of these descriptions correctly classify both the current and the previous two instances, giving them the score $\frac{1+2}{1+2} = 1$. Since the scores for these descriptions equal that for the current hypothesis, IHC selects one at random, say `dark color` $\wedge$ `two tails`, and uses it to replace the current hypothesis.

However, the new description immediately encounters problems on the next positive instance, which it fails to match. Of the two minimally more general hypotheses, only one matches the new training case. This description, `two tails`, has the score $\frac{1+1}{1+2} = \frac{2}{3}$ on the current and previous two instances. This score is the same as that for the current hypothesis, so again IHC goes with the revised description. In addition, it replaces the previously stored positive instance with the current one, for use in future decisions.

The algorithm does not have long to wait before it puts this case to use, finding that the next instance is negative but still matches the description. Actually, this is identical to the second instance in the training sequence, although IHC does not realize it because of its limited memory. This time, the best hypothesis, `two tails` $\wedge$ `two nuclei`, has the score $\frac{1+2}{1+2} = 1$, which is better than for the current description. After replacing the latter with the former, IHC also revises its store of training cases. Given additional instances, the algorithm would continue to revise its hypothesis until it ceased to make errors in classification.

2.5.4 Comments on incremental hill climbing

As we noted earlier, the most obvious advantages of incremental hill climbing are its low requirements for memory and processing. The approach considers only one hypothesis, and it stores and reexamines a constant number of training instances for use in directing the learning process. This compares favorably to the exponential number of de-

scriptions generated by incremental exhaustive algorithms, and even to the constant number of hypotheses and entire training set stored and processed by nonincremental beam search methods.

Naturally, there are prices to be paid for this advantage. We have already mentioned the sensitivity of incremental hill-climbing methods to training order. In addition, such methods typically need more training instances to arrive at the target concept than their more search-intensive relatives. We saw a simple example of this effect in Figure 2-6, where the IHC algorithm required five training instances to induce one of the same concepts that the EGS and IGS algorithms found with only four instances.

Nevertheless, despite the constraints they place on memory and search, incremental hill-climbing methods are robust along a variety of dimensions, in some cases precisely because they cannot maintain consistency with the observed training cases. The incorporation of bidirectional operators contributes to their capabilities, since such operators let the learner simulate backtracking by reversing poor decisions that were based on nonrepresentative training samples.

For those concerned with such issues, the IHC algorithm is guaranteed to converge on a conjunctive target concept, and to retain that description once generated, provided the training instances are sampled randomly with replacement and that they are free of noise. The method will not know when it has completed learning, but humans – the most robust learning systems known – typically do not know this either. Also, IHC is less sensitive to noise than the exhaustive methods, partly because it has limited memory of previous instances and does not attempt to formulate a description that is consistent with all of the training data.

2.6 Genetic algorithms for logical concept induction

The methods we examined earlier in this chapter, which grew out of the tradition of state-space search, all rely directly on the partial ordering by generality that we discussed in Section 2.1.3. Thus, each step these methods take through the hypothesis space involves making an existing description either more specific or more general. However, another important class of methods, known as *genetic algorithms*, come from a different tradition and carry out a very different kind of search. Genetic algorithms have found uses in many disciplines besides machine learning, and even within this field they are not limited to conjunctive induction. However, because they typically operate with logical representations, we will discuss them here.

2.6.1 Genetic representations and operators

Research on genetic algorithms was inspired by an analogy with the theory of evolution, and the basic representation and operators reflect this history. Thus, most work in this paradigm represents knowledge as strings of symbols that correspond loosely to chromosomes. For example, the instance `two nuclei, two tails, thin wall, dark color` might be represented as the bit vector `1 1 0 1`, with ones and zeros indicating the presence or absence of Boolean features. Similarly, a logical conjunction might be represented as `1 * 0 *`, where the symbol * specifies a wild card that matches against either 1 or 0. Such an abstract description specifies a hyper-rectangular region in the instance space, just as does the equivalent expression `two nuclei` $\wedge$ `thin wall`. A common representation for numeric attributes involves transformation into a base two integer, which one then encodes as a bit string.

The evolutionary metaphor extends to the learning operators typically used in such methods. Thus, the simplest genetic operator – *mutation* – randomly replaces one value in a string with another. In the case of logical conjunctions, this can mean replacing a specific attribute value with a wild card (giving a more general description), replacing a wild card with a specific attribute value (giving a more specific description), or replacing one value of an attribute with another (giving a different one at the same level of generality). Typical genetic algorithms do not rely heavily on mutation, using it mainly as a backup to preserve 'genetic diversity' in the population.

A more important operator – *crossover* – also comes from genetic theory, but has a more complex effect. When applied to two strings, A and B, the crossover operator randomly selects a place to split both descriptions, then links the left part of A with the right part of B to form a new description, and links the left part of B with the right part of A to create another. The two resulting strings have some features from each of their parents.

In some situations, the order of features has a clear meaning, but for others (e.g., Boolean features) the order is arbitrary. In the latter

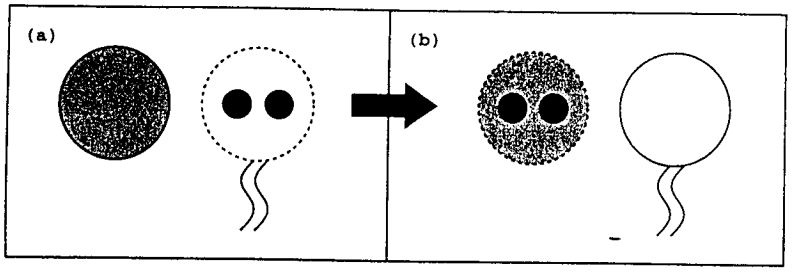

Figure 2-7. An example of crossover's effect in the cell domain. When applied to the two descriptions in (a), the operator produces two new descriptions like those in (b) that include some attribute values from each of the original ones.

case, one can use a variant of crossover that operates on unordered sets of attribute-value pairs by randomly selecting a subset of the attributes and switching their values. Figure 2-7 depicts an example of this scheme in the four-attribute cell domain, in which both the inputs and outputs are conjunctions of two attribute-value pairs.

The crossover operator has some important differences from the specialization and generalization operators we described earlier in the chapter. As we have seen, this operator accepts two descriptions as input rather than one. More important, unlike the other methods we have seen, it makes no use of the partial ordering on the hypothesis space. The resulting descriptions need be neither more nor less general than their parents, since they have aspects of each. Thus, genetic algorithms that incorporate crossover move through the hypothesis space in a quite different manner than the other methods we have described. Nevertheless, experimental studies have repeatedly shown that this approach can lead to useful descriptions.

2.6.2 Search control in genetic algorithms

Table 2-5 shows the basic control structure for a simple genetic algorithm (GA) adapted for use in the induction of logical conjunctions. Basically, the algorithm carries out a form of beam search using the genetic operators discussed above. Initialization involves creating an hypothesis set of `Beam-Size` random entries from the space of possible descriptions. The idea here is to begin search from a representative sample of the description space, so as to decrease the likelihood of halting at a local optimum.

Table 2-5. The GA algorithm: Genetic search for a logical concept description, directed by an evaluation function Score.

```
Inputs: PSET is the set of positive instances.
        NSET is the set of negative instances.

Params: Beam-Size is the overall size of the hypothesis set.
        Cycles is the desired number of iterations.
        Pmutate is the probability of mutated attribute values.

Procedure GA(PSET, NSET)

Let HSET be a set of Beam-Size randomly generated descriptions.
Let CYCLE-COUNT be Cycles.
While CYCLE-COUNT is greater than zero,
      Let TOTAL be zero.
      For each description H in HSET,
          Let S_H be Score(H, PSET, NSET).
          Add S_H to TOTAL.
      Let NEWSET be the empty set.
      Let CHILD-COUNT be Beam-Size.
      While CHILD-COUNT is greater than zero,
            Select hypotheses P and Q from HSET with
               probabilities S_P/TOTAL and S_Q/TOTAL.
            Randomly select a crossover point.
            Apply crossover to P and Q, giving R and S.
            Add hypotheses R and S to NEWSET.
            Decrement CHILD-COUNT by two.
      For each description H in NEWSET,
          For each feature F in H,
          Randomly replace F with probability Pmutate.
      Let HSET be NEWSET.
      Decrement CYCLE-COUNT by one.
Return the member of HSET with the best score.
```

On each cycle, GA uses an evaluation function to measure the quality of the current descriptions with respect to the training data, giving each competitor an associated score. The method then uses the crossover operator to generate `Beam-Size` successors. In each case, it selects two descriptions as parents, drawing them at random from the hypothesis set with probability in direct proportion to their relative scores. That

is, if S_H is the score for hypothesis H and TOTAL is the summed score over all hypotheses, then the probability that a new description will have H as one of its parents is S_H/TOTAL. If H occurs k times in the current set, this probability effectively becomes $k \cdot S_H$/TOTAL.

For each pair of parents, the crossover operator selects a crossover point (or set of attributes), generates two new descriptions, and adds them to its set of revised hypotheses. GA then mutates each feature in each description with some (low) probability. The algorithm evaluates each of the resulting set of descriptions on the training data in turn, creates yet another set of children, and so on, continuing the specified number of iterations.

In many ways, the beam search carried out by genetic algorithms is similar to the mechanisms posited in biological theories of natural selection, and the terminology used in work on this topic reflects the similarity. Papers commonly refer to Beam-Size as the *population size*, to the evaluation metric as the *fitness* function, to the number of *generations* that have occurred, and to the *offspring* in a given population. The scheme for determining the number of children as a probabilistic function of scores corresponds roughly to the idea of survival of the fittest.

The table does not give details about the nature of the evaluation metric, but GA can use any function that reflects accuracy on the training set. Given a conjunctive target concept and a measure like $(P_c + N_{\bar{c}})/(P + N)$ from Section 2.3, the algorithm will tend to converge on a hypothesis set that contains Beam-Size copies of a single description (or slight variations) after sufficient generations have passed. Because GA does not use the generality/specificity ordering to bias its search, this description may be the most specific one that covers the training data, the most general one, or somewhere in between. If simplicity is a concern, one can use a metric that explicitly incorporates this factor.

2.6.3 Comments on genetic algorithms

One can easily alter genetic algorithms to let them process training instances incrementally. Rather than evaluating each generation on the entire training set, one computes fitness scores on the last k instances, with this set being updated after a given number of generations have passed. However, this scheme is best viewed as a form of incremental beam search, rather than as incremental hill climbing.

In terms of robustness, the GA method shares many characteristics with other beam search algorithms for logical induction. Thus, GA should be able to handle noisy training data, as well as target concepts that are only approximated by logical conjunctions. The algorithm is quite tractable in terms of computation time, but it is not guaranteed to converge on an optimal target description.

Again, we should emphasize that, in adapting genetic search to the task of inducing logical conjunctions, we have made many simplifying assumptions. The GA method shown in Table 2-5 conveys the basic ideas underlying genetic algorithms, but such techniques are not restricted to conjunctive induction, nor even to machine learning. We will return to them in later chapters in quite different contexts.

2.7 Summary of the chapter

In this chapter we considered the induction of logical conjunctions. We saw that, combined with an all-or-none matcher, such descriptions produce decision boundaries that are parallel to some of the axes in the instance space. We also found that the space of conjunctive descriptions is partially ordered by generality, and that most methods for inducing them from experience make use of this ordering.

Our discussion of induction techniques focused first on nonincremental methods that use the entire training set to direct search through the hypothesis space. For example, the EGS algorithm carries out a constrained breadth-first search from general to specific descriptions, using negative instances to eliminate entire branches of the search tree. The ESG algorithm instead employs a specific-to-general search organization, using positive training cases to constrain its decisions. These two methods differ in their inductive bias, with the former preferring more general conjunctions and the latter preferring more specific ones. However, we found that such exhaustive methods have a number of drawbacks, including their insistence on noise-free data and, in some domains, an exponential growth in memory.

In response, we examined heuristic approaches to logical concept induction, focusing on the HGS algorithm. This method carries out a beam search through the description space, using an evaluation function that measures fit to the training data to select the best hypotheses

at each level. Like EGS, the HGS method moves from general to specific descriptions, but we also briefly examined a complementary algorithm, HSG, that operates in the opposite direction. Neither method is guaranteed to find the best description as measured by the evaluation metric, but we found that they are more robust than the exhaustive schemes in the presence of noise and target concepts that violate the conjunctive assumption.

We then turned to incremental techniques for the same task. The IGS algorithm operates in much the same manner as EGS, except that each step through the hypothesis space is driven by a single negative training case, with positive instances being used to remove overly specific descriptions. We also considered ISG, which moves in the specific-to-general direction, and two approaches to bidirectional search. Although these methods are incremental in the sense that they reprocess few instances, they suffer from the same problem of exponential memory growth as the EGS and ESG algorithms, and thus violate the spirit of incremental learning.

We found that incremental hill-climbing methods such as the IHC algorithm come closer to this ideal. By considering only one hypothesis at a time and retaining only a few instances in memory, such methods guard against high memory costs. In return, they may suffer from order effects and require more training cases to reach the same target concept, although this drawback can be partially offset by the use of bidirectional learning operators that let the hill climber simulate the effects of backtracking.

Finally, we examined genetic approaches to logical induction, which are based on an analogy with evolutionary theory rather than state-space search. Genetic algorithms retain a fixed-size population of descriptions in each generation, effectively carrying out a beam search through the hypothesis space, using an evaluation function to select promising paths. The primary difference between methods like GA and techniques like HGS and IHC is that genetic algorithms do not take much advantage of the partial ordering on conjunctive descriptions. Instead, they rely on operators like crossover that produce quite different effects from those given by generalization and specialization operators.

Exercises

1. Show the logical expressions for the maximally general conjunctions that are consistent with the training cases `(-, 0.5, 0.5)`, `(-, 0.8, 2.4)`, `(+, 1.8, 1.6)`, `(-, 0.7, 2.2)`, and `(-, 2.4, 2.2)`, which take the form `(class, girth, height)`. Draw the decision regions corresponding to each of these expressions. Now show the expressions for all maximally specific conjunctions consistent with these data, along with their decision regions. Assume an integral description language that allows tests like `(< height 5)` but not ones like `(< height 5.5)`.
2. Trace the behavior of the EGS algorithm (Section 2.2.1) on the training instances:

class	tails	nuclei	color	wall
+	two	two	dark	thin
-	two	two	light	thin
-	two	one	light	thin
-	one	two	dark	thick
-	two	two	dark	thick

   Give the values of the arguments CSET, HSET, and SPECS on each recursive call of the algorithm.
3. Apply the HGS algorithm (Section 2.3.1) to the data in Exercise 2, using the evaluation function $P_c/P + N_{\bar{c}}/N$, rather than $(P_c + N_{\bar{c}})/(P + N)$. Trace the algorithm's behavior, showing values for the arguments CLOSED-SET, HSET, and SPECS on each recursive call, as well as any evaluation scores computed. How does the behavior of HGS compare with that of EGS on these data?
4. Trace the behavior of the IGS algorithm (Section 2.4.1) on the numeric data in Figure 2-4, showing values for the variables I, HSET, and GSET after processing each training case in ISET. Verify that this method produces the same final hypotheses as the EGS algorithm.
5. Write pseudocode for the ISG algorithm (Section 2.4.3), using Table 2-3 as a model. Trace ISG's behavior on the training data `(+, 2.0, 2.0)`, `(-, 3.0, 1.0)`, `(+, 5.0, 4.0)`, `(+, 3.0, 4.0)`, `(+, 4.0, 5.0)`. For each instance I, show the values of the variables HSET and GSET (or their counterparts in your pseudocode).

6. Assume that the IHC algorithm (Section 2.5.2) observes the four training instances `(+, 1.7, 2.2)`, `(-, 2.4, 0.5)`, `(-, 1.2,1.7)`, `(+, 2.6, 2.1)`, which take the form `(class, girth, height)`. Further suppose that the first two instances have led IHC to set the hypothesis `H` to `(< girth 2)`, that the parameter $k = 3$, and that the description language allows only integer tests. Show the values of the variables `GSET`, `SSET`, `H`, and `H'`, along with any scores computed, when the algorithm processes each of the last two instances.
7. Assume a string representation for hypotheses over the cell domain that takes the form `(tails color nuclei wall)`, making the two candidates in Figure 2-7 (a) become `(* dark * thin)` and `(two * two *)`, where `*` represents the absence of a test for the attribute in that position. The two hypotheses in Figure 2-7 (b) illustrate one possible way of applying the crossover operator (Section 2.6.1). Show the results of the other crossovers that are possible (taking the string position into account) for the two hypotheses in Figure 2-7 (a).

Historical and bibliographical remarks

Research on the induction of logical conjunctions occupied a central position in the early machine learning literature, particularly during the 1970s. Although Bruner, Goodnow, and Austin (1956) outlined the basic idea, Winston (1975) was the first to implement and popularize a version of the specific-to-general ISG algorithm. His system used a depth-first search strategy, but both Vere (1975) and Hayes-Roth and McDermott (1978) report breadth-first versions of the same approach. Mitchell (1977) extended this framework, introducing the idea of a version space bounded by S and G sets, generated by a candidate elimination algorithm that combined the ISG and IGS methods in a bidirectional manner. He also did much to popularize the notion of learning as search through a space of hypotheses (Mitchell, 1982).

Langley, Gennari, and Iba (1987) first proposed incremental hill climbing as a general framework for learning. Iba, Wogulis, and Langley (1988) adapted this idea to inducing logical concepts, while Anderson and Kline (1979) and Langley (1987) described similar schemes that rely on incremental beam search. However, none of these researchers limited their attention to conjunctive concepts. By 1980, work in the logical framework had started to focus on the induction of more complex structures, but the new techniques still incorporated as subroutines the methods developed earlier for simple conjunctions.

Nonincremental methods for logical induction emerged from a different line of research than did incremental ones. Here there was no early emphasis on exhaustive methods like the ESG and EGS algorithms (which we introduced only for tutorial purposes). The INDUCE system (Michalski, 1980) carries out an HGS-like general-to-specific search for conjunctive descriptions, and later programs like AQ11 (Michalski & Chilausky, 1980) and CN2 (Clark & Niblett, 1989) incorporate this technique as a subroutine to aid the induction of more complex logical concepts.

Genetic algorithms were first proposed by Holland (1975), and there now exists a large literature on their use in both learning and optimization. Proceedings of annual conferences have been published since 1985 to report current results, and special issues of the journal *Machine Learning* serve a similar function. Booker, Goldberg, and Holland (1989) review a particular approach to genetic learning, and Goldberg's (1990) text gives broader coverage of this area. Examples of genetic methods for supervised concept induction can be found in Wilson (1987) and in De Jong and Spears (1991).