

FIGURE 8.1. Classification in a basic decision tree proceeds from top to bottom. The questions asked at each node concern a particular property of the pattern, and the downward links correspond to the possible values. Successive nodes are visited until a terminal or leaf node is reached, where the category label is read. Note that the same question, **Size?**, appears in different places in the tree and that different questions can have different numbers of branches. Moreover, different leaf nodes, shown in pink, can be labeled by the same category (e.g., **Apple**). From: Richard O. Duda, Peter E. Hart, and David G. Stork, *Pattern Classification*. Copyright © 2001 by John Wiley & Sons, Inc.

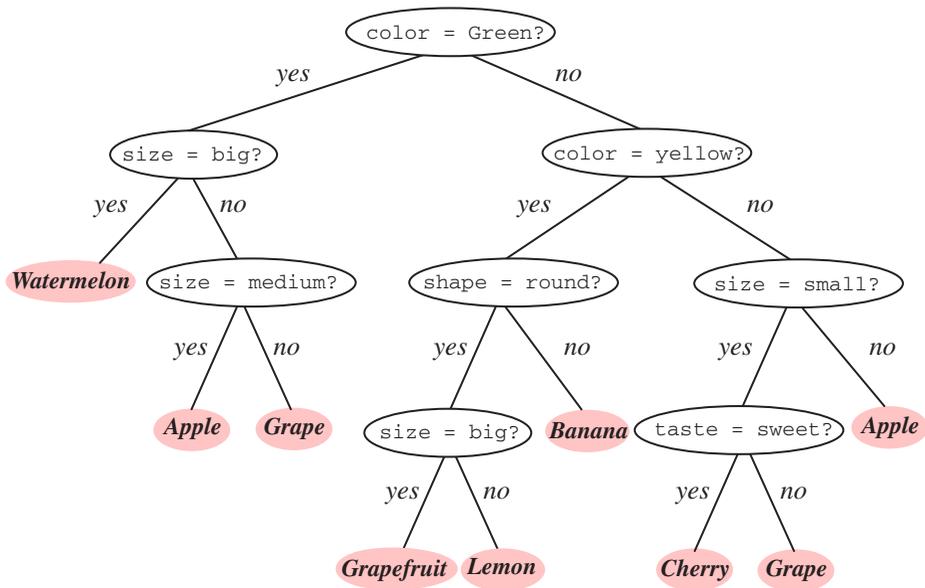


FIGURE 8.2. A tree with arbitrary branching factor at different nodes can always be represented by a functionally equivalent binary tree—that is, one having branching factor $B = 2$ throughout, as shown here. By convention the “yes” branch is on the left, the “no” branch on the right. This binary tree contains the same information and implements the same classification as that in Fig. 8.1. From: Richard O. Duda, Peter E. Hart, and David G. Stork, *Pattern Classification*. Copyright © 2001 by John Wiley & Sons, Inc.

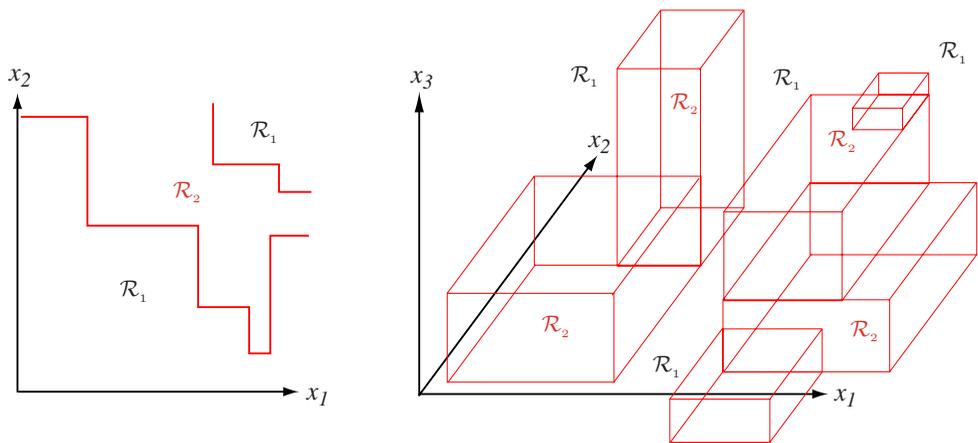


FIGURE 8.3. Monothetic decision trees create decision boundaries with portions perpendicular to the feature axes. The decision regions are marked \mathcal{R}_1 and \mathcal{R}_2 in these two-dimensional and three-dimensional two-category examples. With a sufficiently large tree, any decision boundary can be approximated arbitrarily well in this way. From: Richard O. Duda, Peter E. Hart, and David G. Stork, *Pattern Classification*. Copyright © 2001 by John Wiley & Sons, Inc.

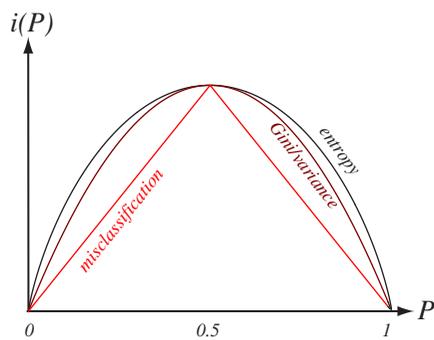


FIGURE 8.4. For the two-category case, the impurity functions peak at equal class frequencies and the variance and the Gini impurity functions are identical. The entropy, variance, Gini, and misclassification impurities (given by Eqs. 1–4, respectively) have been adjusted in scale and offset to facilitate comparison here; such scale and offset do not directly affect learning or classification. From: Richard O. Duda, Peter E. Hart, and David G. Stork, *Pattern Classification*. Copyright © 2001 by John Wiley & Sons, Inc.

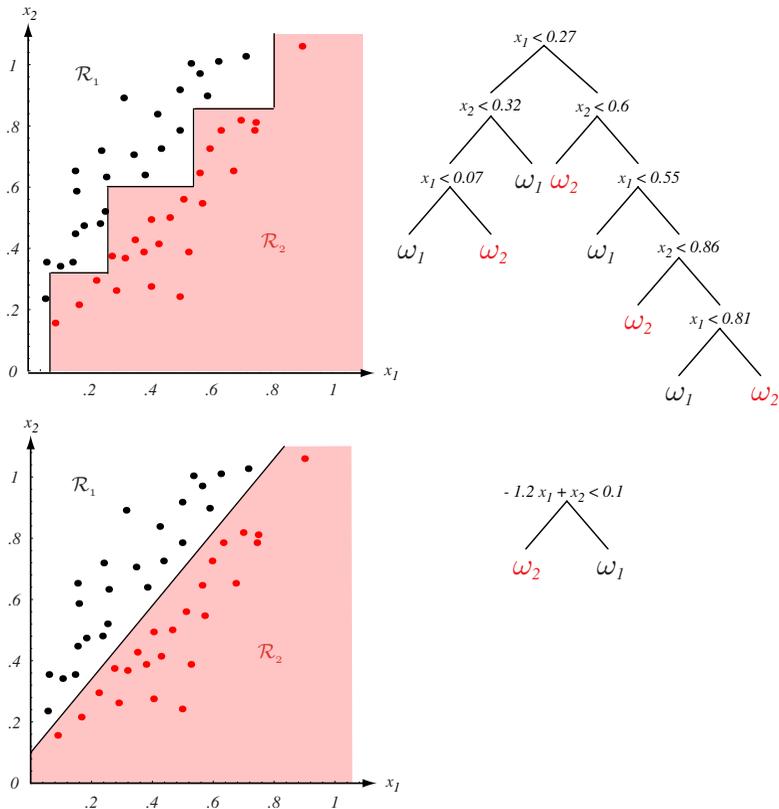


FIGURE 8.5. If the class of node decisions does not match the form of the training data, a very complicated decision tree will result, as shown at the top. Here decisions are parallel to the axes while in fact the data is better split by boundaries along another direction. If, however, “proper” decision forms are used (here, linear combinations of the features), the tree can be quite simple, as shown at the bottom. From: Richard O. Duda, Peter E. Hart, and David G. Stork, *Pattern Classification*. Copyright © 2001 by John Wiley & Sons, Inc.

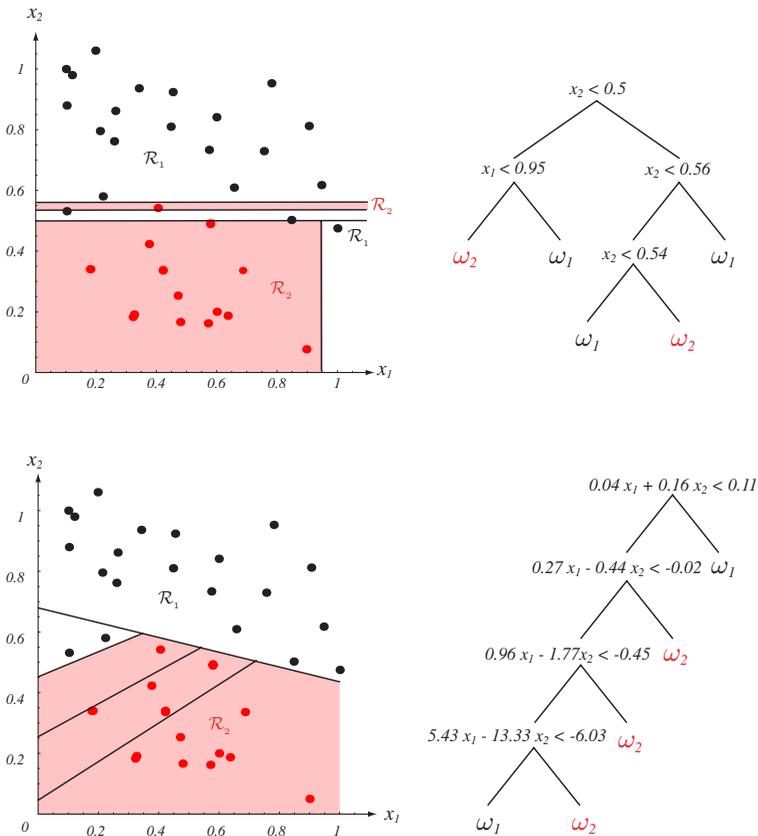


FIGURE 8.6. One form of multivariate tree employs general linear decisions at each node, giving splits along arbitrary directions in the feature space. In virtually all interesting cases the training data are not linearly separable, and thus the LMS algorithm is more useful than methods that require the data to be linearly separable, even though the LMS need not yield a minimum in classification error (Chapter 5). The tree at the bottom can be simplified by methods outlined in Section 8.4.2. From: Richard O. Duda, Peter E. Hart, and David G. Stork, *Pattern Classification*. Copyright © 2001 by John Wiley & Sons, Inc.

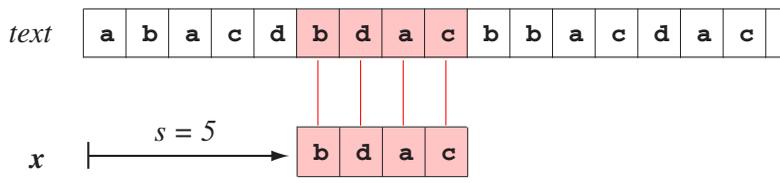


FIGURE 8.7. The general string-matching problem is to find all shifts s for which the pattern x appears in $text$. Any such shift is called *valid*. In this case $x = \text{"bdac"}$ is indeed a factor of $text$, and $s = 5$ is the only valid shift. From: Richard O. Duda, Peter E. Hart, and David G. Stork, *Pattern Classification*. Copyright © 2001 by John Wiley & Sons, Inc.

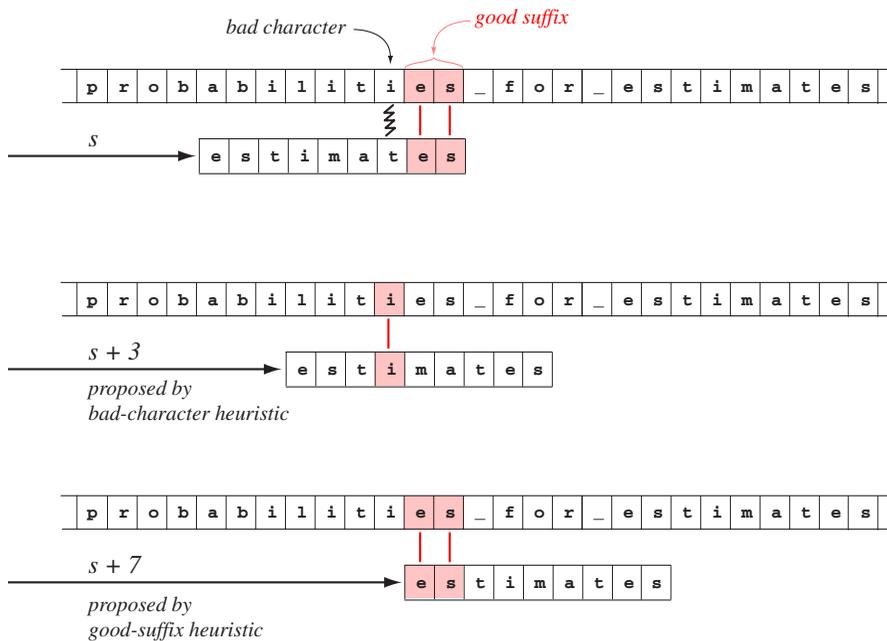


FIGURE 8.8. String matching by the Boyer-Moore algorithm takes advantage of information obtained at one shift s to propose the next shift; the algorithm is generally much less computationally expensive than naive string matching, which always increments shifts by a single character. The top figure shows the alignment of *text* and pattern x for an invalid shift s . Character comparisons proceed right to left, and the first two such comparisons are a match—the good suffix is “es.” The first (rightmost) mismatched character in *text*, here “i,” is called the *bad character*. The bad-character heuristic proposes incrementing the shift to align the rightmost “i” in x with the bad character “i” in *text*—a shift increment of 3, as shown in the middle figure. The bottom figure shows the effect of the good-suffix heuristic, which proposes incrementing the shift the least amount that will align the good suffix, “es” in x , with that in *text*—here an increment of 7. Lines 11 and 12 of the Boyer-Moore algorithm select the larger of the two proposed shift increments, i.e., 7 in this case. Although not shown in this figure, after the mismatch is detected at shift $s + 7$, both the bad-character and the good-suffix heuristics propose an increment of yet another 7 characters, thereby finding a valid shift. From: Richard O. Duda, Peter E. Hart, and David G. Stork, *Pattern Classification*. Copyright © 2001 by John Wiley & Sons, Inc.

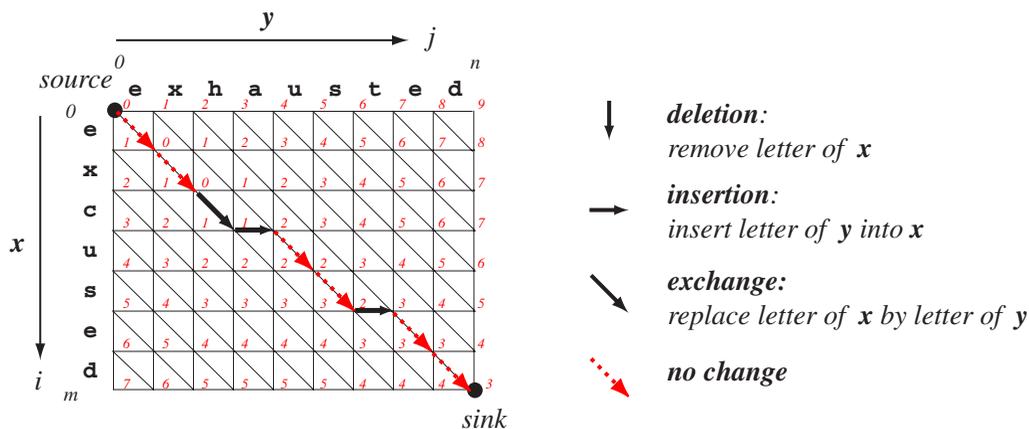


FIGURE 8.9. The edit distance calculation for strings x and y can be illustrated in a table. Algorithm 3 begins at *source*, $i = 0, j = 0$, and fills in the cost matrix C , column by column (shown in red), until the full edit distance is placed at the *sink*, $C[i = m, j = n]$. The edit distance between “excused” and “exhausted” is thus 3. From: Richard O. Duda, Peter E. Hart, and David G. Stork, *Pattern Classification*. Copyright © 2001 by John Wiley & Sons, Inc.

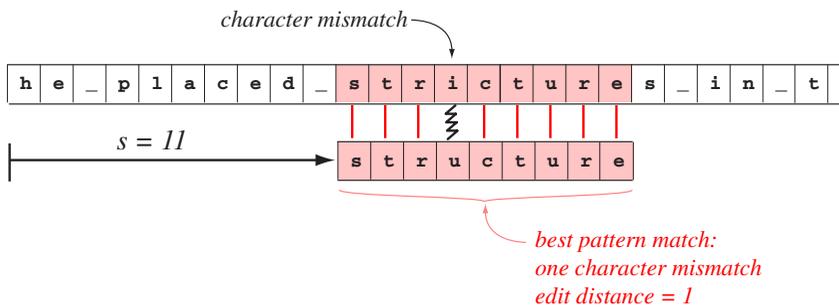


FIGURE 8.10. The string-matching-with-errors problem is to find the shift s for which the edit distance between x and an aligned factor of $text$ is minimum. In this illustration, the minimum edit distance is 1, corresponding to the character exchange $u \rightarrow i$, and the shift $s = 11$ is the location. From: Richard O. Duda, Peter E. Hart, and David G. Stork, *Pattern Classification*. Copyright © 2001 by John Wiley & Sons, Inc.

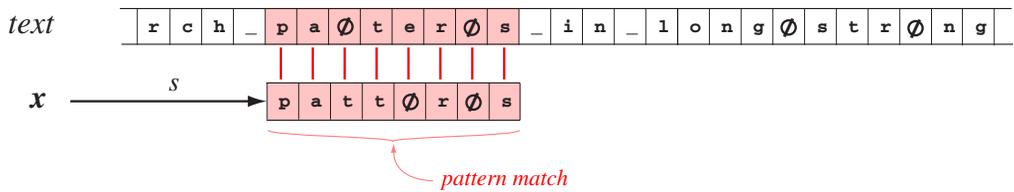


FIGURE 8.11. The problem of string matching with don't care symbol is the same as that in basic string matching except that the \emptyset symbol—in either *text* or *x*—can match any character. The figure shows the only valid shift. From: Richard O. Duda, Peter E. Hart, and David G. Stork, *Pattern Classification*. Copyright © 2001 by John Wiley & Sons, Inc.

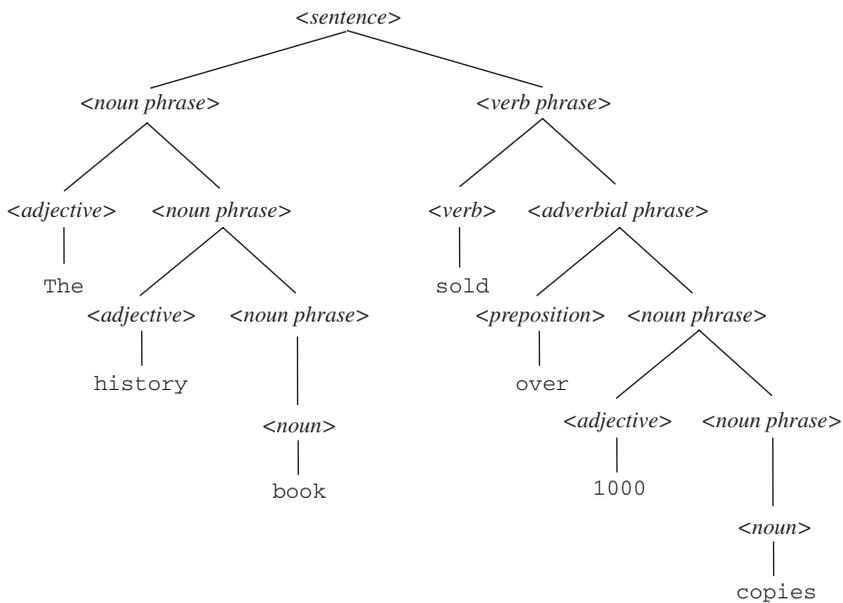


FIGURE 8.12. This derivation tree illustrates how a portion of English grammar can transform the root symbol, here (*sentence*), into a particular sentence or string of elements, here English words, which are read from left to right. From: Richard O. Duda, Peter E. Hart, and David G. Stork, *Pattern Classification*. Copyright © 2001 by John Wiley & Sons, Inc.

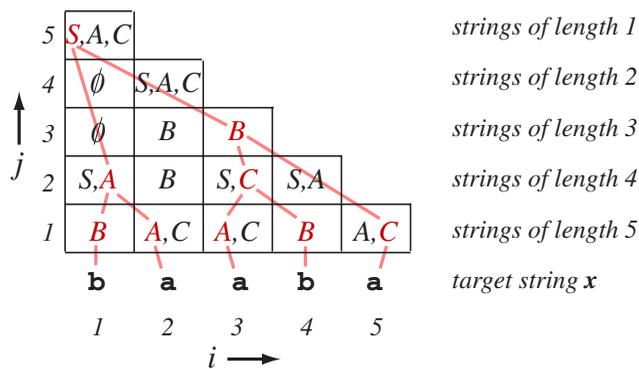


FIGURE 8.13. The bottom-up parsing algorithm fills the parse table with symbols that might be part of a valid derivation. The pink lines are not provided by the algorithm, but when read downward from the root symbol they confirm that a valid derivation exists. From: Richard O. Duda, Peter E. Hart, and David G. Stork, *Pattern Classification*. Copyright © 2001 by John Wiley & Sons, Inc.

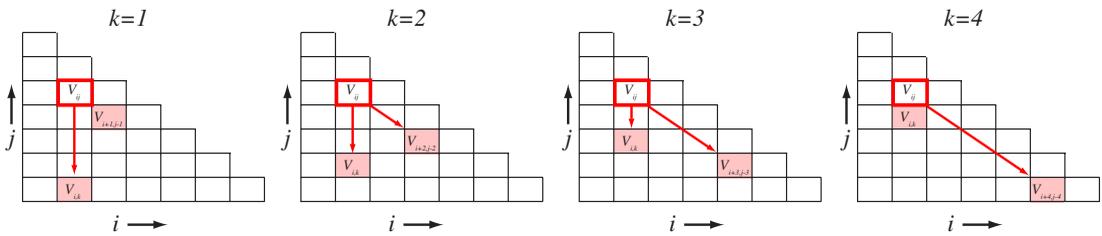


FIGURE 8.14. The innermost loop of Algorithm 4 seeks to fill a cell V_{ij} (outlined in red) by the left-hand side of any rewrite rule whose right-hand side corresponds to symbols in the two shaded cells. As k is incremented, the cells queried move vertically upward to the cell in question and move diagonally down from that cell. The shaded cells show the possible right-hand sides in a derivation, as illustrated by the pink lines in Fig. 8.13. From: Richard O. Duda, Peter E. Hart, and David G. Stork, *Pattern Classification*. Copyright © 2001 by John Wiley & Sons, Inc.

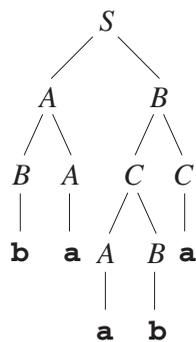


FIGURE 8.15. This valid derivation of “babaa” in G can be read from the pink lines in the parse table of Fig. 8.13 generated by the bottom-up parse algorithm. From: Richard O. Duda, Peter E. Hart, and David G. Stork, *Pattern Classification*. Copyright © 2001 by John Wiley & Sons, Inc.

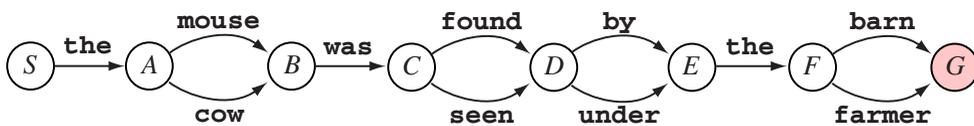


FIGURE 8.16. One type of finite-state machine consists of nodes that can emit terminal symbols (“the,” “mouse,” etc.) and transition to another node. Such operation can be described by a grammar. For instance, the rewrite rules for this finite-state machine include $S \rightarrow \text{the}A$, $A \rightarrow \text{mouse}B$ OR $\text{cow}B$, and so on. Clearly these rules imply this finite-state machine implements a type 3 grammar. The final internal node (shaded) would lead to the null symbol ϵ . From: Richard O. Duda, Peter E. Hart, and David G. Stork, *Pattern Classification*. Copyright © 2001 by John Wiley & Sons, Inc.

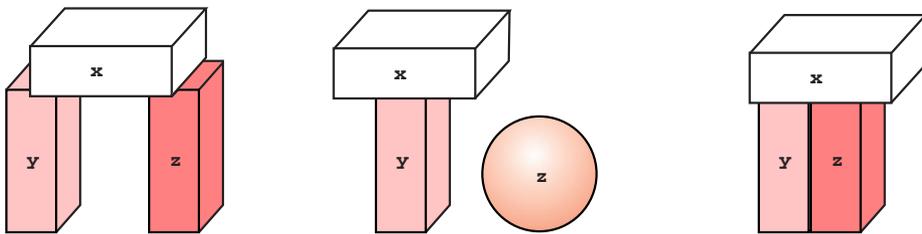


FIGURE 8.17. The rule in Eq. 11 identifies the figure on the left as an example of *Arch*, but not the other two figures. In practice, it is very difficult to develop subsystems that evaluate the propositions themselves, for instance *Touch*(x, y) and *Supports*(x, y, z). From: Richard O. Duda, Peter E. Hart, and David G. Stork, *Pattern Classification*. Copyright © 2001 by John Wiley & Sons, Inc.

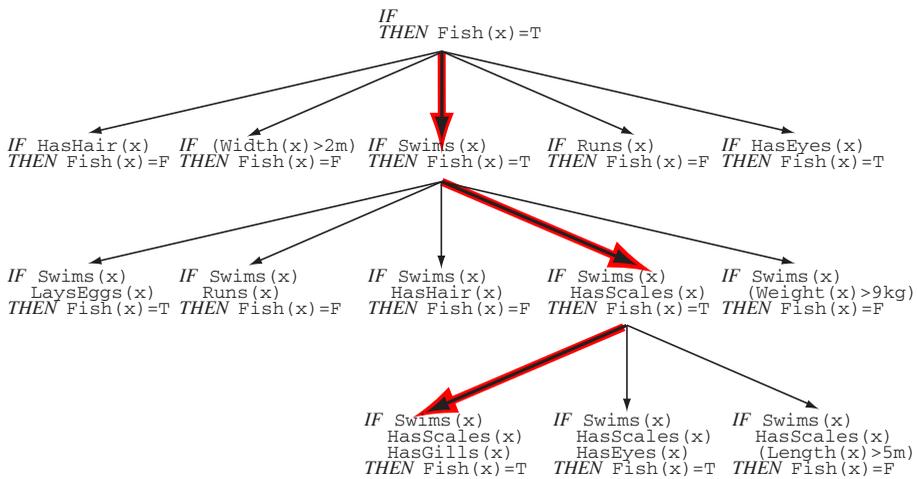


FIGURE 8.18. In sequential covering, candidate rules are searched through successive refinements. First, the “best” rule having a single conditional predicate is found—that is, the one explaining most training data. Next, other candidate predicates are added, the best compound rule is selected, and so forth. From: Richard O. Duda, Peter E. Hart, and David G. Stork, *Pattern Classification*. Copyright © 2001 by John Wiley & Sons, Inc.