19 Learning by Recording Cases

In this chapter, you learn how it is possible to deal with problem domains in which good models are impossible to build.

In particular, you see how to *learn by recording cases as is*, doing nothing to the information in those cases until that information is used. First, you learn that you are using the *consistency heuristic* whenever you attribute a feature of a previously observed thing to a new, never-before-seen thing. Next, you learn how it is possible to find nearest neighbors in a feature space quickly using *k-d trees*.

By way of illustration, you see how to find the nearest neighbors of a wooden block, measured by width and height, in time proportional to the logarithm of the number of blocks. You also see how the same ideas make it possible to move a robot's arm without developing complicated motion equations and without obtaining difficult-to-measure arm parameters.

Once you have finished this chapter, you will know how to do nearest-neighbor calculations, and you will understand some of the conditions under which nearest-neighbor calculations work well.

RECORDING AND RETRIEVING RAW EXPERIENCE

In this section, you learn about the consistency heuristic, and you learn to identify the kinds of problems you can solve by recording cases for later use.

The Consistency Heuristic Enables Remembered Cases to Supply Properties

Consider the eight blocks in figure 19.1. Each has a known color, width, and height. Next, suppose you are confronted with a new block of size 1 by 4 centimeters and of unknown color. If you had to guess its color, given nothing else to go on, you would have to guess that the color is the same as that of the block that is most similar in other respects—namely in width and height. In so guessing, you would use the consistency heuristic:

The **consistency heuristic**:

▷ Whenever you want to guess a property of something, given nothing else to go on but a set of reference cases, find the most similar case, as measured by known properties, for which the property is known. Guess that the unknown property is the same as that known property.

Plotting the widths and heights of the cases yields the feature space shown in figure 19.2 and makes it easy to apply the consistency heuristic in the color-guessing situation. No fancy reasoning is needed. Because the width and height of the unknown, labeled U, are closest to the width and height of the orange block, you have to assume that the orange block is the right case, and to guess that block U is orange.

The Consistency Heuristic Solves a Difficult Dynamics Problem

Consider the problem of moving a robot hand along a prescribed path in space at a prescribed speed, as in figure 19.3. To succeed, you clearly need to know how the joint angles should change with time, and what joint torques will produce those joint-angle changes.

Relating the joint angles to manipulator position is a **kinematics** problem. It is relatively easy to derive formulas that relate manipulator position to joint angles. For the joint angles, θ_1 and θ_2, and segment lengths, l_1 and l_2, of the two-dimensional manipulator shown in figure 19.3, the equations are as follows:

$$x = l_1 \cos\theta_1 + l_2 \cos(\theta_1 + \theta_2),$$
$$y = l_1 \sin\theta_1 + l_2 \sin(\theta_1 + \theta_2).$$

For the two-joint, two-dimensional manipulator shown in figure 19.3, it is also straightforward, albeit tedious, to derive the inverse formulas relating joint angles to manipulator position:

$$\theta_1 = \tan^{-1}\left(\frac{y}{x}\right) - \tan^{-1}\left(\frac{l_2 \sin\theta_2}{l_1 + l_2 \cos\theta_2}\right)$$
$$\theta_2 = \cos^{-1}\left(\frac{x^2 + y^2 - l_1^2 - l_2^2}{2 l_1 l_2}\right)$$

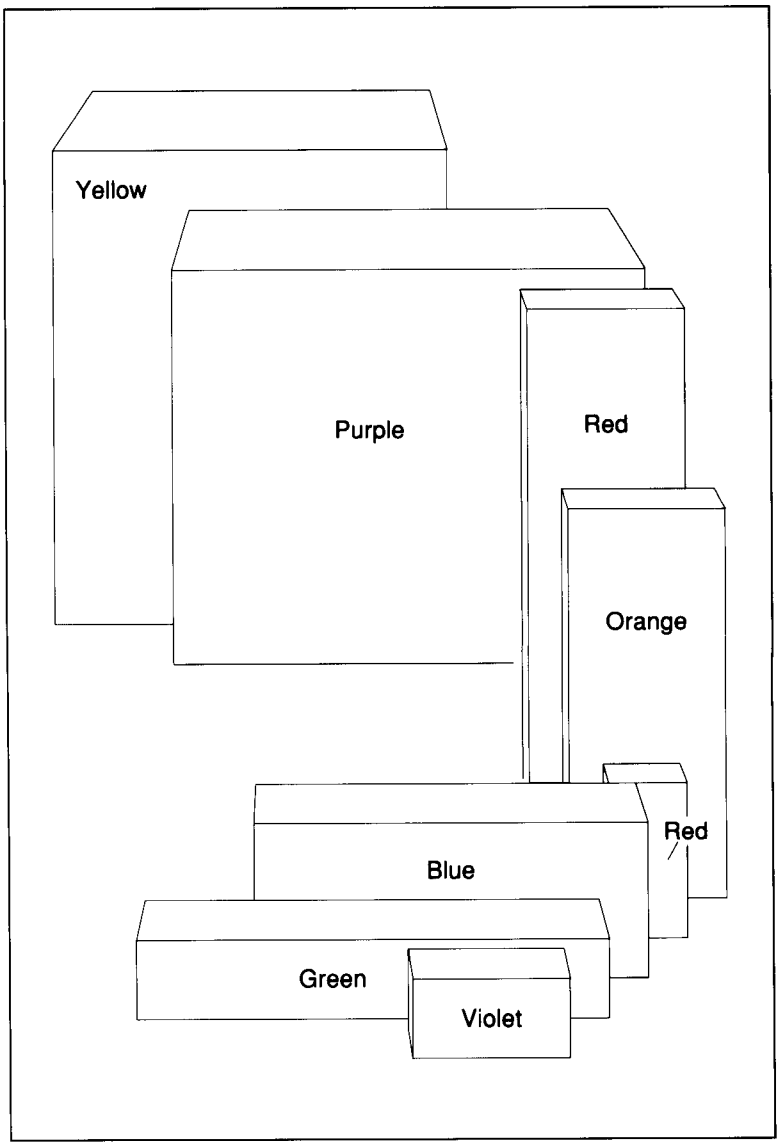

Figure 19.1 Eight blocks of known color, width, and height. These eight form a set of cases by which color can be guessed for other blocks of known width and height.

Given formulas for joint angles, you might think it would be easy to move a manipulator: Just chop up the desired trajectory into pieces, determine the necessary joint motions during each piece, and tell the motors about the results. Unfortunately, the only message that motors understand is one that tells them the torque you want them to supply.

Relating joint motions to required motor torques is a *dynamics* problem, which can be unbearably intricate mathematically, even though everything ultimately is just a matter of Newton's second law relating force to

Figure 19.2 A feature space relating the block of unknown color to eight blocks of known color. The color of the unknown, block U, is judged to be the same as the color of the orange block, the one that is closest in width and height.

the product of mass and acceleration. The following complicated-looking equations emerge, ignoring gravity, assuming cylindrical links, for the simple two-joint, two-dimensional manipulator shown in figure 19.3:

$$\begin{aligned}
\tau_1 =& \ddot{\theta}_1(I_1 + I_2 + m_2 l_1 l_2 \cos\theta_2 + \frac{m_1 l_1^2 + m_2 l_2^2}{4} + m_2 l_1^2) \\
&+ \ddot{\theta}_2(I_2 + \frac{m_2 l_2^2}{4} + \frac{m_2 l_1 l_2}{2}\cos\theta_2) \\
&- \dot{\theta}_2^2 \frac{m_2 l_1 l_2}{2}\sin\theta_2 \\
&- \dot{\theta}_1\dot{\theta}_2 m_2 l_1 l_2 \sin\theta_2, \\
\tau_2 =& \ddot{\theta}_1(I_2 + \frac{m_2 l_1 l_2}{2}\cos\theta_2 + \frac{m_2 l_2^2}{4}) \\
&+ \ddot{\theta}_2(I_2 + \frac{m_2 l_2^2}{4}) \\
&+ \dot{\theta}_1^2 \frac{m_2 l_1 l_2}{2}\sin\theta_2.
\end{aligned}$$

where each τ_i is a torque, each $\dot{\theta}_i$ is an angular velocity, each $\ddot{\theta}_i$ is an angular acceleration, each m_i is a mass, each l_i is a length, and each I_i is a moment of inertia about a center of mass.

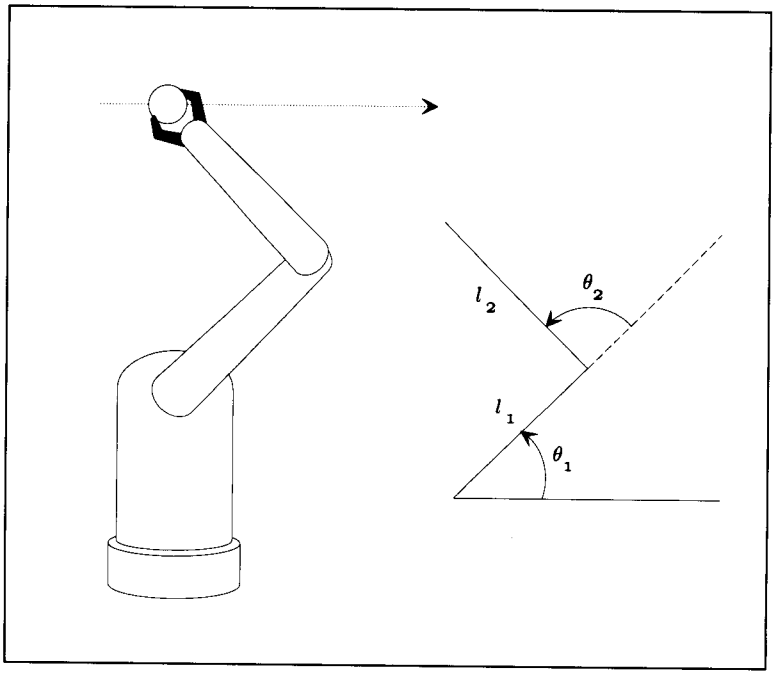

Figure 19.3 A robot arm throwing a ball. To arrange the throw, you must know the joint angles that produce the straight-line motion shown, and you must know how to apply torques that produce the desired joint angles.

The torque equations for three-dimensional manipulators are much bulkier because six joints are required to place a manipulator at a given position in space with a given orientation. Like the solution for two joints, the real-world solutions demonstrate that the required torques depend, in general, on accelerations, on velocities squared, on velocity products, and on multipliers that depend on joint position:

- Because there are velocities squared, the torques necessarily involve *centripetal forces.*
- Because there are products of different velocities, the torques involve *Coriolis forces.*
- Because there are multipliers that are functions of several angles, the torques involve *variable, cross-coupled moments of effective inertia.*

Even with all this mathematical sophistication, it remains difficult to get satisfactory results with real-world robot arms, and to explain how we can manage to move our biological arms. There are just too many factors to consider and too many measurements to be made with too much precision.

Fortunately, nearest-neighbor calculations in a feature space, together with the notion of practice, provide an alternate approach to robot-arm control. Imagine a giant table with columns for torques, positions, velocities, squared velocities, velocity products, and accelerations:

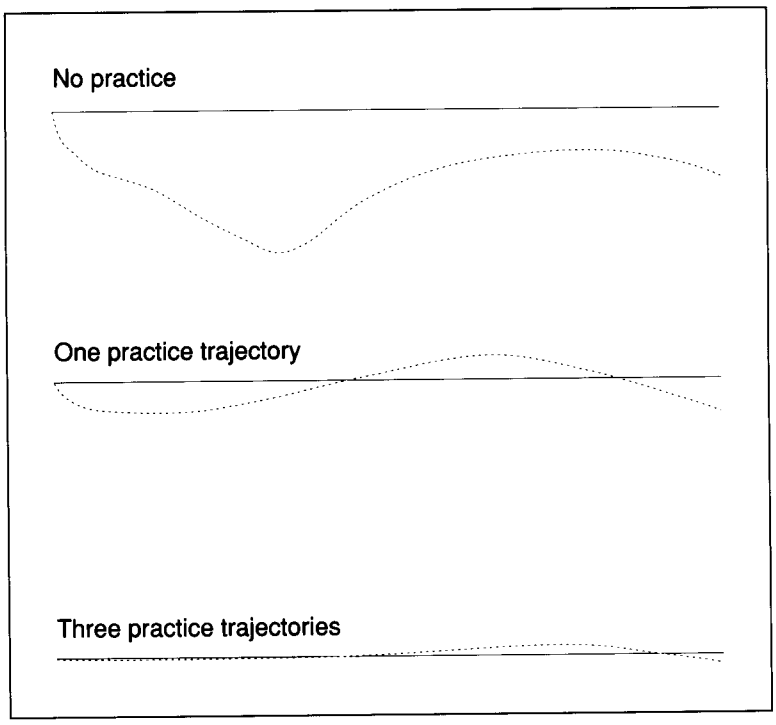

Figure 19.4 The robot arm shown in figure 19.3 tries to follow a straight line. In the first instance the robot arm does poorly because its table is sparse. In the second and third instances, a table that relates torques to desired path parameters is used, and the robot arm does much better.

τ_1	τ_2	θ_1	θ_2	$\dot{\theta}_1$	$\dot{\theta}_2$	$\dot{\theta}_1^2$	$\dot{\theta}_2^2$	$\dot{\theta}_1\dot{\theta}_2$	$\ddot{\theta}_1^2$	$\ddot{\theta}_2^2$

Next, suppose you issue a command that causes the robot arm to be waved about more or less randomly. Every so often, you measure the torques, positions, and other indicated parameters, and you record the results in the giant table.

Then, when you want to move the robot arm along a prescribed trajectory, you break up that trajectory into little pieces; treat the giant table as a feature space; look for entries with nearby positions, velocities, squared velocities, velocity products, and accelerations; and interpolate among them to find appropriate torques for the corresponding little piece of trajectory.

You might worry, legitimately, that no table could hold enough entries to fill the feature space densely enough to do a good job, even with an elaborate interpolation method. To combat this density problem, you arrange for practice. The first time that you have the robot try a particular reach or throw motion, it does miserably, because the table relating torques to positions, velocities, and accelerations is sparse. But even though the robot does miserably, it is still writing new entries into its table, and these

new entries are closer than old entries to the desired trajectory's positions, velocities, and accelerations. After a few tries, motion becomes smooth and accurate, as shown in figure 19.4, because you are interpolating among the new table entries, and they are much closer to what you want.

FINDING NEAREST NEIGHBORS

In this section, you learn about two relatively fast ways to find nearest neighbors; one is serial, and one is parallel.

A Fast Serial Procedure Finds the Nearest Neighbor in Logarithmic Time

The straightforward way to determine a block's nearest neighbor is to calculate the distance to each other block, and then to find the minimum among those distances. For n other blocks, there are n distances to compute and $n - 1$ distance comparisons to do. Thus, the straightforward approach is fine if n is 10, but it is not so fine if n is 1 million or 1 billion.

Fortunately, there is a better way, one for which the average number of calculations is proportional to $\log_2 n$, rather than to n. This better way involves the use of a special kind of **decision tree**. In general, a decision tree is an arrangement of tests that prescribes the most appropriate test at every step in an analysis:

A **decision tree** is a representation

That is a semantic tree

In which

- ▷ Each node is connected to a set of possible answers.
- ▷ Each nonleaf node is connected to a test that splits its set of possible answers into subsets corresponding to different test results.
- ▷ Each branch carries a particular test result's subset to another node.

To exploit the decision-tree idea so as to deal with the block-identification example, you divide up the cases in advance of nearest-neighbor calculation. As illustrated in figure 19.5, all cases are divided first by height alone into two sets, each with an equal number of blocks. In one set, all heights are equal to or greater than 5 centimeters; in the other, equal to or less than 2 centimeters. A 3-centimeter block-free zone separates the two sets.

Next, each of the two sets is divided by width alone. The tall set is divided into one set in which all widths are equal to or greater than 5 centimeters, and another set in which all widths are equal to or less than 2

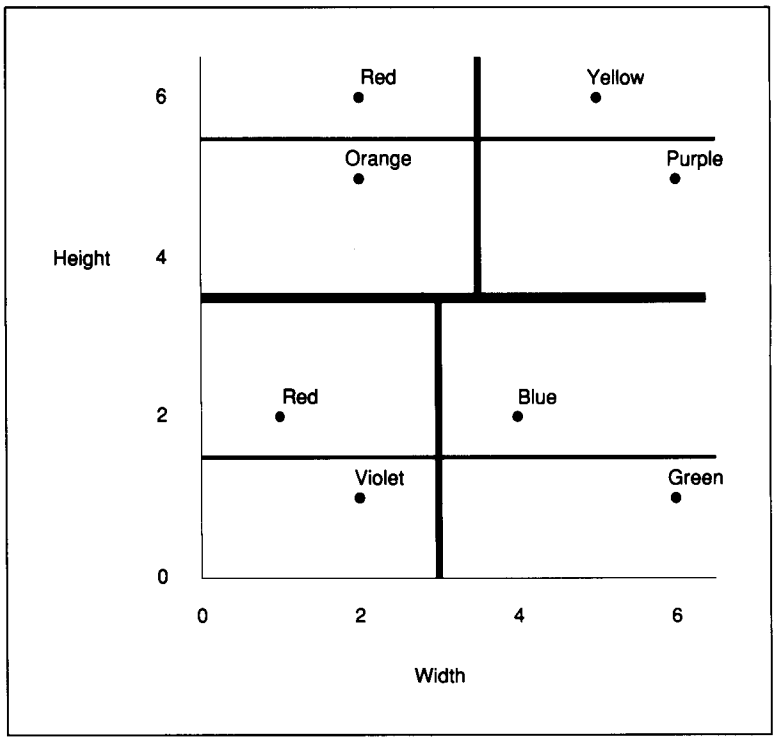

Figure 19.5 The case sets are divided horizontally and vertically until only one block remains in each set.

centimeters. Similarly, the bottom set is divided into one set of blocks 2 centimeters or less, and one set of blocks 4 centimeters or greater.

Finally, each of those four sets is divided by height alone, producing eight sets of just one block each.

The overall result is called a **k-d** tree, where the term *k-d* is used to emphasize that distances are measured in *k* dimensions:

A **k-d tree** is a representation

That is a decision tree

In which

- ▷ The set of possible answers consists of points, one of which may be the nearest neighbor to a given point.
- ▷ Each test specifies a coordinate, a threshold, and a neutral zone around the threshold containing no points.
- ▷ Each test divides a set of points into two sets, according to on which side of the threshold each point lies.

To determine the nearest neighbor of U, you first note that U's height is more than 3.5 centimeters, which is the middling height between the

shortest tall block and the tallest short block. From this observation, you conclude that U is more likely, albeit not certain, to be nearer to one of the tall blocks than to one of the short blocks. On this ground, you temporarily ignore the short blocks.

Because the tallest short block is 2 centimeters tall, the distance between U and the tallest short block is at least 2 centimeters, and maybe is more, because the difference in height alone is 2 centimeters, as is evident in the top part of figure 19.6. If U proves to be equal to or less than 2 centimeters from a tall block, your decision temporarily to ignore the short blocks will become a permanent decision. If U is more than 2 centimeters from a tall block, you will have to reconsider the short blocks eventually.

Next, you consider the tall blocks, which are themselves divided into two sets. Because U's width is less than 3.5 centimeters, U is more likely, albeit not certain, to be nearer to one of the narrow tall blocks than to one of the wide tall blocks. On this ground, you temporarily ignore the wide tall blocks.

As illustrated in the middle part of figure 19.6, if U proves to be equal to or less than 4 centimeters from a narrow tall block, your decision temporarily to ignore the wide tall blocks will become a permanent decision, because the width of U differs by 4 centimeters from the width of the narrowest wide tall block.

One more step puts the unknown with the short, narrow, tall blocks, of which there is only one, the orange block, as illustrated in the bottom part of figure 19.6. If U proves to be equal to or less than 2 centimeters from the orange block, there is no need to calculate the distance to the narrow tall red block, which differs from U by 2 centimeters in height alone.

Now it is clear that the nearest block is the orange one, at a distance of 1.41 centimeters, provided that all previous decisions turn out to be justified. In this example, those decisions are justified because 1.41 centimeters is less than 2 centimeters, justifying the rejection of the narrow tall red block; it is less than 4 centimeters, justifying the rejection of the yellow and purple blocks; and it is less than 2 centimeters, justifying the rejection of all the short blocks.

Finding the nearest block is really just a matter of following a path through a decision tree that reflects the way the objects are divided up into sets. As the decision tree in figure 19.7 shows, only three one-axis comparisons are required to guess the nearest neighbor in the example, no matter what the width and height of the unknown are. Once the distance to the guessed nearest neighbor is calculated, only three more comparisons are needed to validate the decisions that led to the guess if it is correct. If you are unlucky, and the guess is wrong, you have to look harder, working down through the sets that have been ignored previously.

In general, the decision tree with branching factor 2 and depth d will have 2^d leaves. Accordingly, if there are n objects to be identified, d will

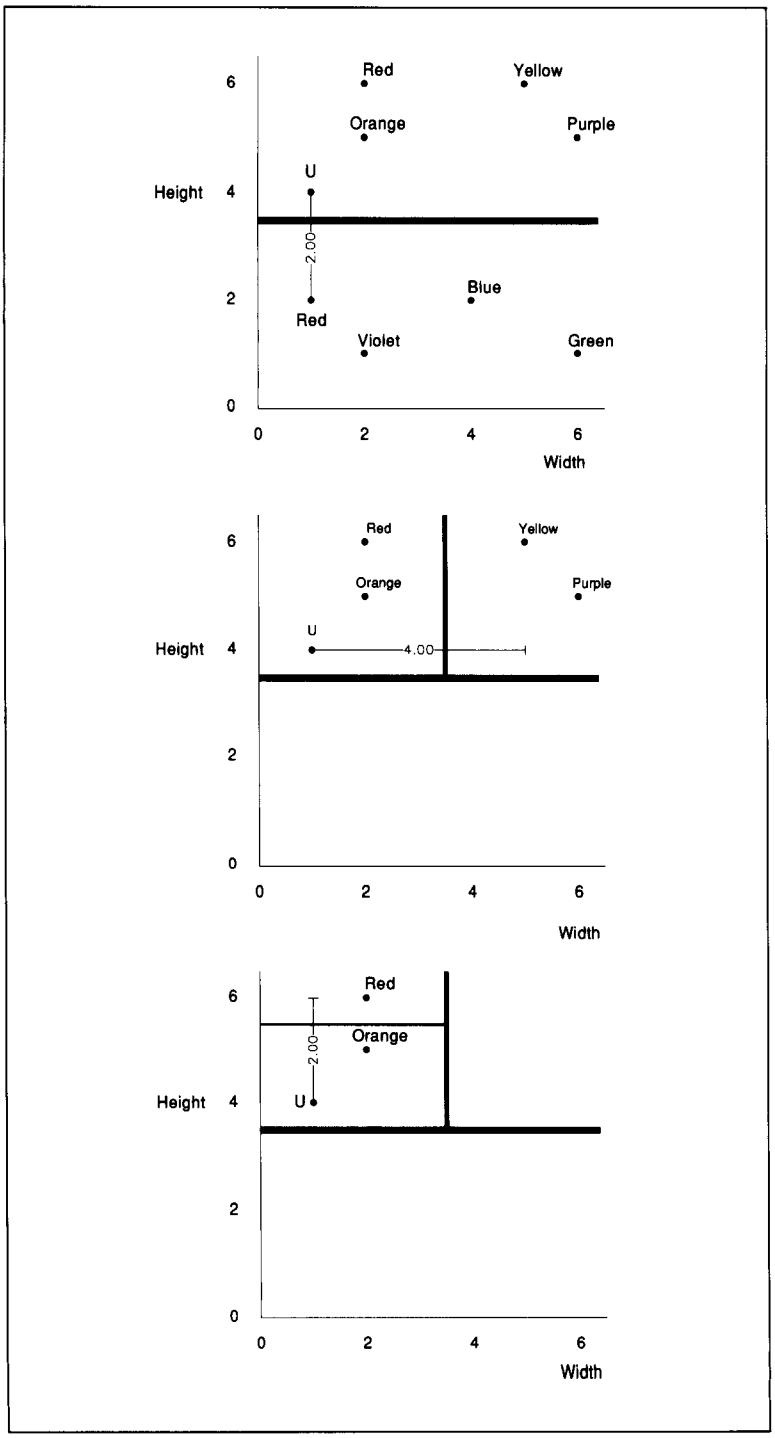

Figure 19.6 In the top part, unknown U cannot be closer than 2 centimeters to any block in the bottom set, because the height of block U is 4 centimeters and the height of the tallest block in the bottom set is 2 centimeters. In the middle part, the remaining cases are divided into two sets. In one set, all widths are greater than 3.5 centimeters; in the other, less than 3.5 centimeters. Because the width of block U is 1 centimeter and the width of the narrowest block in the right set is 5 centimeters, block U cannot be closer than 4 centimeters to any block in the right set. Finally, in the bottom part, only two cases remain. Because the height of block U is 4 centimeters and the height of the red block is 6 centimeters, block U cannot be closer than 2 centimeters to the red block.

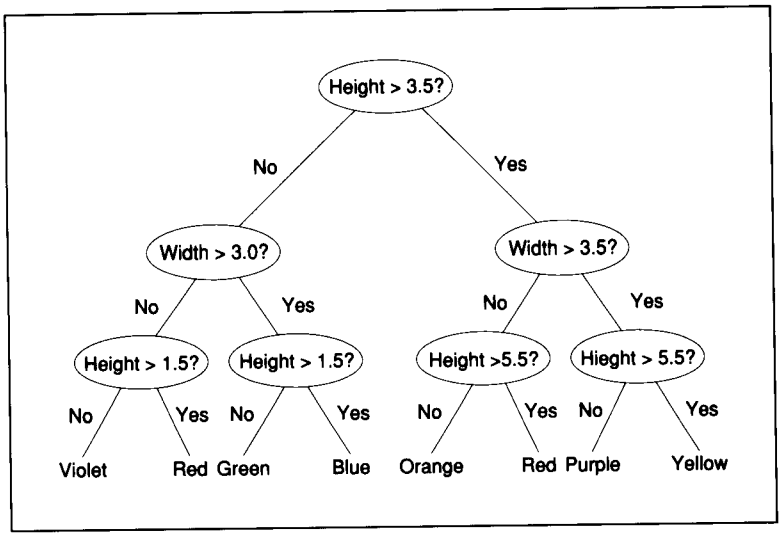

Figure 19.7 The complete k-d tree for identifying a new block's nearest neighbor.

have to be large enough to ensure that $2^d \geq n$. Taking the logarithm of both sides, it is clear that the number of comparisons required, which corresponds to the depth of the tree, will be on the order of $\log_2 n$. If there are eight objects, the saving does not amount to much. If there are 1 billion objects, however, the number of comparisons is on the order of 30, not 1 billion, which is a substantial saving.

Here is the procedure for dividing the cases into sets, building a decision tree along the way:

To divide the cases into sets,

- ▷ If there is only one case, stop.
- ▷ If this is the first division of cases, pick the vertical axis for comparison; otherwise, pick the axis that is different from the axis at the next higher level.
- ▷ Considering only the axis of comparison, find the average position of the two middle objects. Call this average position the threshold, and construct a decision-tree test that compares unknowns in the axis of comparison against the threshold. Also note the position of the two middle objects in the axis of comparison. Call these positions the upper and lower boundaries.
- ▷ Divide up all the objects into two subsets, according to on which side of the average position they lie.
- ▷ Divide up the objects in each subset, forming a subtree for each, using this procedure.

Of course, the procedure for finding the nearest neighbor must address the possibility that previously ignored choices will have to be examined. Its common name is the K-D procedure:

To find the nearest neighbor using the K-D procedure,

- ▷ Determine whether there is only one element in the set under consideration.
  - ▷ If there is only one, report it.
  - ▷ Otherwise, compare the unknown, in the axis of comparison, against the current node's threshold. The result determines the likely set.
  - ▷ Find the nearest neighbor in the likely set using this procedure.
  - ▷ Determine whether the distance to the nearest neighbor in the likely set is less than or equal to the distance to the other set's boundary in the axis of comparison:
    - ▷ If it is, then report the nearest neighbor in the likely set.
    - ▷ If it is not, check the unlikely set using this procedure; return the nearer of the nearest neighbors in the likely set and in the unlikely set.

Parallel Hardware Finds Nearest Neighbors Even Faster

If you happen to have a massively parallel computer, with so many processors that each case can have its own, then none of this fancy search is needed. Each distance measurement can proceed in parallel.

Of course, all the results have to be compared somehow to find the case with the minimum distance from the unknown. One way to do this comparison would be to have neighboring processors compare their results. Then each two-processor minimum would be compared with a neighboring two-processor minimum. Carrying on this way eventually would lead to the global minimum after on the order of $\log_2 n$ sequential steps, where n is the number of distances to be compared. There are better ways, however, that find the minimum distance in constant time on a parallel computer.

SUMMARY

- The consistency heuristic is the justification for the use of remembered cases as sources of properties for previously unseen objects.
- Remembered cases can help you to solve many difficult problems, including problems in dynamic arm control.

- The K-D procedure is a fast serial procedure that finds nearest neighbors in logarithmic time.
- Parallel hardware can find nearest neighbors even faster than the K-D procedure can.

BACKGROUND

The discussion of K-D trees is based on the work of Jerome H. Friedman and colleagues [1977].

The discussion of arm control is based on the work of Chris Atkeson [1990]. Atkeson's more recent work does not involve K-D trees, however; instead, he has developed fast hardware that essentially interpolates among all cases.