5
CHAPTER

Behaviors and Artificial Intelligence

Since the first fictional stories about robots were written, robots have been given a built-in ability to "think" for themselves and to carry out tasks specified by their creators. In fiction, robots tend to be malevolent beings, either programmed to carry out revenge on those unfortunate enough to have crossed the paths of their inventors, or transformed from gentle, useful tools into monsters bent on destruction. In most stories, the complexity of programming the robots is generally avoided, since it would bog down the story and detract from the carnage that takes place.

When you first started to experiment with the *TAB Electronics Build Your Own Robot*, I hope you discovered that the robot seemed to move with almost organiclike decision-making about where it should point itself and whether or not is should move forward. These "behaviors" are designed to carry out simple tasks and are truly the limit of artificial intelligence possible with today's understanding of computer science.

This may be surprising to you if you have ever seen robots on educational TV channels demonstrating reactions that seem very intelligent. The actions of these robots are in fact "behaviors" that are very similar to those programmed into the *TAB Electronics Build Your Own Robot Kit*.

In this chapter, I will introduce you to the concept of behaviors and describe how the behaviors that come with the *TAB Electronics Build Your Own Robot Kit* were designed. In addition, I will discuss with you some of my thoughts about artificial intelligence.

Robot Behaviors

Whenever I hear the term "behavior," I generally think of a bratty kid lying on the floor kicking and screaming because he isn't getting his way.

As unlikely as this seems, robot behaviors have many of the same features as this sort of "behavior." A behavior is a set of responses to external events (often referred to as the "stimulus") that is designed to reach an end goal. Each behavior has been designed to handle different responses to changing stimuli. Properly designed behaviors will counter all possible responses in order to carry through to the desired result.

Going to the example of the bratty child, the program that dictates the "behavior" could be

```
BrattyBehavior5:                                    '  Get Toy
'
'  Execute Actions designed to get specified toy

  output "Mom, Dad?  Can I have that toy?"
  input Response

  if (Response = "Yes") then BrattyBehavior5End     '  Response Acceptable?
                                                    '  No...  Intimidate
BrattyBehavior5Loop                                 '  Return Here
  Lie Down on Floor
  Cry
  Kick Floor
  output "You <emphasize>SAID</emphasize> I could have a toy!"

  if (input.Pending = False) then BrattyBehavior5Loop
                                                    '  No Response, Repeat
  input Response
  if (input = "Okay, okay, we'll Buy it") then BrattyBehavior5End
                                                    '  Successful Behavior
  if (input <> "If you don't stop, you'll get such a spanking") then +
   BrattyBehavior5End                               '  No Threat - Repeat

  output "Sorry Mommy and Daddy"                    '  Threat. Gone too far

BrattyBehavior5End:                                 '  End Response Reached
  Stand up
  Stop Crying
  end
```

This program probably seems whimsical, but as I will demonstrate in the following sections, robot behaviors are very similar to this. In any event, the behavior is given an end goal along with the methodology for achieving it.

Note that there are provisions for alternative stimulus. In the bratty behavior example, the end goal was getting the toy, but if the behavior went too far and was eliciting a potentially negative response (spanking), the behavior stops as a safety measure. Actual robot behaviors work in a similar fashion and can be programmed as in the example above.

Behavior 1—Random Movement

After assembling the *TAB Electronics Build Your Own Robot Kit*, you probably installed a battery and set the robot down on the floor to watch it perform a number of random motions such as reversing its motion if it encountered objects in front of it. This operation is known as "Behavior 1" and it was my initial test for the operation of the robot prototypes.

The behavior can be initiated using the remote control by pushing on the top left button, which has the icon shown in Figure 5-1. This icon (like the icons for the other behaviors) represents the movement of the behavior. In this case, the robot goes forward for a couple of seconds, turns, and then repeats the forward motion. When left for a set period of time, this behavior will place the robot at some random position within the room.

Behavior 1's actions can be listed as:

1. Move forward for 2 seconds.
   - If robot encounters an object, go to 6.
2. Stop motors for 300 msec.
3. Turn left for 760 msec.
   - If robot encounters an object, go to 6.
4. Stop motors for 300 msec.
5. Goto 1.
6. Turn on LED to indicate collision.
7. Stop motors for 400 msec.
8. Move backward for 1 second.
9. Stop motors for 400 msec.
10. Turn left for 600 msec if no object to left.
11. Turn right for 600 msec if object to left.
12. Turn off LED, indicating collision handled.
13. If collision occurred at 1, go to 2.
14. Go to 1.

This behavior can be programmed in PBASIC, but I will refrain from doing this for a number of reasons. During the robot's movement, the collision sensors are continuously polled, and the motion of the robot will stop immediately. This is difficult to reproduce in PBASIC code—especially given that the timing of the movement must be quite accurate. Another reason not to reproduce the code in PBASIC is that it is very tedious to do so.

There are two purposes of Behavior 1. The first is to provide a "demo" behavior for the robot that demonstrates its ability to move and sense objects around it. The second is to provide you with a method of moving the robot to a random location in the room. This behavior probably doesn't seem that useful at first, but there are a lot of applications (especially concerning artificial intelligence) where it will be quite useful.

After reading the above, you may be confused by the idea that Behavior 1 actually results in random motion. After all, it seems as though the set time for moving forward and turning will cause the robot to end up in the same location each time, if

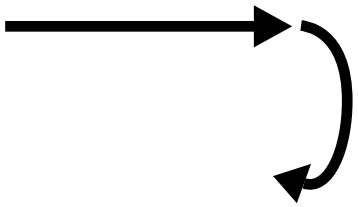

Figure 5-1 Behavior 1—"random movement" icon.

it starts at the same place. This would be true in a "perfect world," but in a real-life environment you will find that the wheels will slip on dust on the floor in a manner in which you cannot predict. The amount of torque and the speed of the wheels is dependent on the charge in the batteries. The impact of these two variables makes the final position of the robot random.

Behavior 2—Photovore

A natural behavior of living organisms is to move toward light. I'm sure you've noticed bugs of all types flying around a light in the summer. This is a natural behavior—if there's light there's food. This behavior prevents insects from dying of starvation in a cave. The remote control icon for this behavior, Figure 5-2, indicates that the robot goes toward the light.

The action of following light was one of the first behaviors investigated by roboticists, going back as far as the 1940s. These simple robots would turn toward a light and move toward it, allowing the scientists to observe a machine behavior that was not preset or predefined. The movement, as you will see when you experiment with the behavior, is quite random and is often unexpected.

The term "photovore" characterizes the robot's motion toward the light as if it were food—"photovore" translates literally into "light eater." There have been a number of experimental robots that seek out light and, when they reach it, rest as if they are eating and then continue on seeking out other light sources. These robots are used to better understand machine intelligence and responses and to model insect and animal behaviors.

The photovore behavior is best characterized by the PBASIC code:

```
Behavior2:                                          '  Go Toward Light
'
'  Go to the brightest object in the room
Behavior2Loop:                                      '  Read in the Current
  i = LeftCDS                                       '   Light Levels at
  j = RightCDS                                      '   Both CDS Cells

  if (LeftCDS = RightCDS) then Behavior2Forward     '  Straight if Equal
  if (LeftCDS > RightCDS) then Behavior2Right       '  Turn towards smaller
                                                    '   CDS Cell Value
Behavior2Left:                                      '  Left CDS Cell
  Robot(TurnLeft, 80msecs)                          '   Exposed to more
  goto Behavior2Forward                             '   Light
```

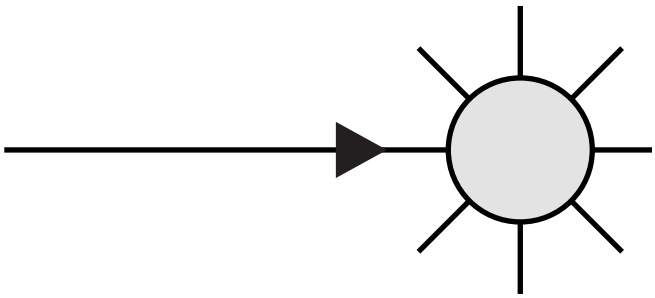

Figure 5-2 Behavior 2—"photovore" icon.

```
Behavior2Right:                                  '  Right CDS Cell
  Robot(TurnRight, 80msecs)                      '   Exposed to more
                                                 '   Light
Behavior2Forward:                                '  Finished Turn, go
  if (Collision = True) then Behavior2End        '   Forward until
  Robot(Forward, 80msecs)                        '   Collision with
  if (Collision = False) then Behavior2End       '   Light

Behavior2End:                                    '  At Light Source
  End
```

Note that when the photovore reaches an obstacle, it stops. This was done to bound the behavior to a simple set of parameters. It also allows other applications to use this as a simple component without requiring the user to program it into the robot. When I present some sample BS2 applications for the robot, I will show how this behavior can be used.

There are a few things that you should notice about how the code accesses the CDS cells on the robot. The first is that the brighter the light a CDS cell is exposed to, the smaller the value that is returned. To turn towards the light, the turn is made toward the side of the CDS cell that returns the smallest value.

The code that is used to "read" the resistance of the CDS cell is not always accurate to the least significant bit. There are a number of reasons why this is the case. You will find that the least significant bits of the result can change without any apparent reason.

This can be a confounding problem, especially if you are new to programming and are trying to get an application running. To mitigate this problem (both for new programmers as well as myself), I AND the returned values with $FC to clear the least significant two bits.

This simple "filtering" eliminates any problems with the bits seeming to change without reason and allows for a reasonable range for both 1) CDS cells to be equal and 2) the robot to go forward without any turning.

When you look at the example PBASIC code above, you may be surprised to see that the motors are turned on for only 80 msec—for Behavior 1, the motors are on for 10 times longer just for turning. This short "pulse" of power to the motors moves the robot very slightly closer to the final goal rather than attempting to turn the robot so that it is pointing exactly at the light source.

Trying to point exactly in any direction is a very difficult undertaking for any kind of robot. Dust on the floor that causes the wheels to slip or a change in the battery charge hamper the robot's ability to precisely set or determine its position.

When I first programmed this behavior I used a 200-msec motor pulse for turning and moving forward. This pulse was generally much too long and would result in the robot overshooting the desired vector to the light source. By keeping the turning and moving interval as short as possible, the robot movements were much more precise.

When you were setting up and learning about the robot, I suggested that you place the robot in a dark room with a flashlight at one point and then initiate Behavior 2 (Figure 5-3). Depending on the room that you test the application in, you may find that the robot begins to chase reflections on baseboards or furniture.

The reason that the robot goes after reflections is that it has a limited field of view (about 160°, as shown in Figure 5-3), and it will "home in" on the brightest light

within this field of view. It is important to remember that Behavior 2's programming does not search around looking for the brightest object in the room; it simply follows the brightest source of light in front of it.

Another potential problem that you may discover with the robot is that there is a "dead spot" exactly in the middle of the robot where the I/R collision detection system does not register objects in front on it. I have shown this situation in Figure 5-7. This situation is exacerbated by the operation of the flashlight bulb. The IR the flashlight produces from the heat of the bulb overpowers the weak IR signal and prevents the detectors from reading the modulated signal.

The solution to this problem (as well as minimizing the reflections the robot follows) is to use a high-intensity LED for the light source instead of a flashlight. Along with this, make sure that there are some objects (such as two books) on either side of the light source. This will ensure that the stop point is not in the "dead spot" of collision detectors.

Behavior 3—Photophobe

The obvious opposite to a behavior that searches out light is one that avoids light. This behavior is commonly called a "photophobe" because it acts as if it is afraid of light rather than being attracted to it like the "photovore" behavior. In nature, this behavior is often employed by animals to avoid predators. The remote control icon, Figure 5-5, indicates that the robot is moving *away* from light rather than toward.

The code to implement the photophobe behavior is remarkably similar to what I used to implement the photovore:

```
Behavior3:                                          '  Go Away from Light
'
'  Go to the darkest spot in the room

Behavior3Loop:                                      '  Read in the Current
  i = LeftCDS                                       '   Light Levels at
  j = RightCDS                                      '   Both CDS Cells
```

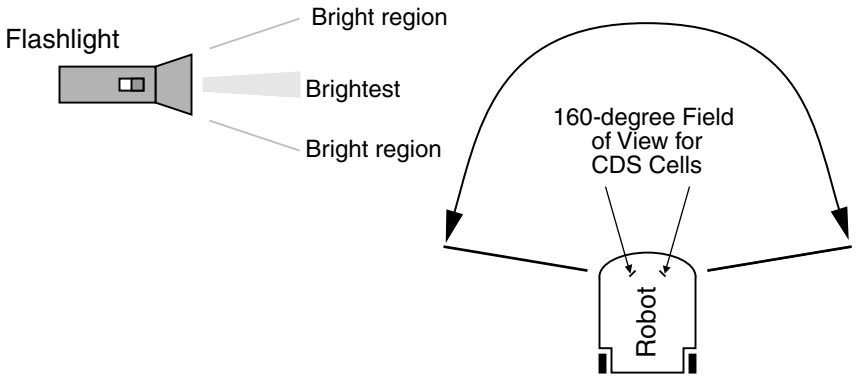

Figure 5-3 Apparatus for testing Behavior 2.

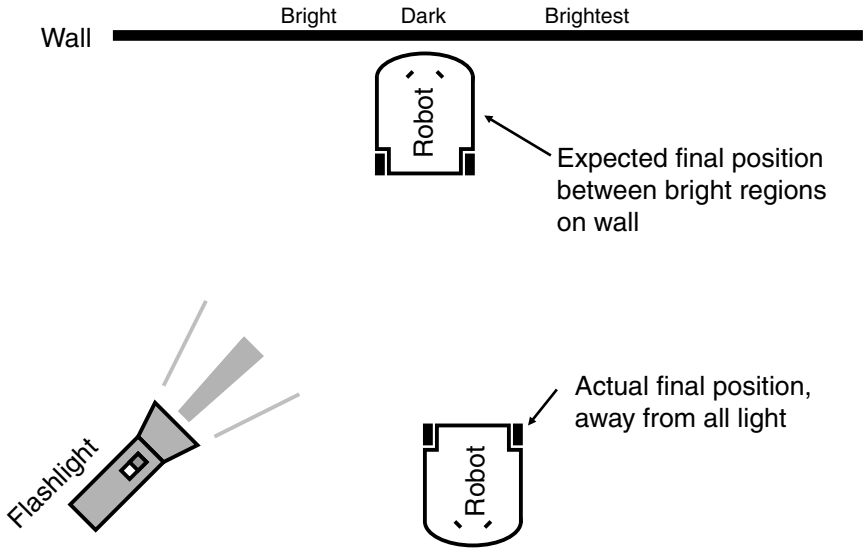

Figure 5-4 Final position of Behavior 3.

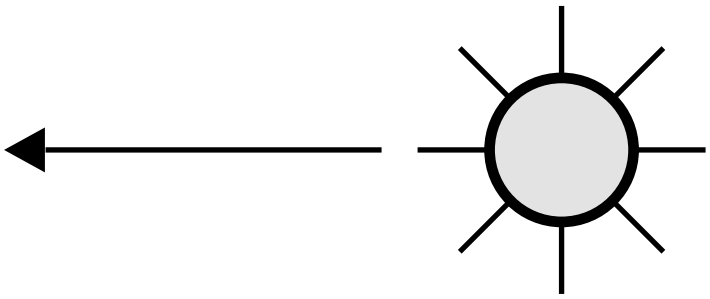

Figure 5-5 Behavior 3—"photophobe" icon.

```
  if (LeftCDS = RightCDS) then Behavior2Forward   '  Straight if Equal
  if (LeftCDS < RightCDS) then Behavior2Right     '  Turn towards larger
                                                  '   CDS Cell Value
Behavior3Left:                                    '  Left CDS Cell
  Robot(TurnLeft, 80msecs)                        '   Exposed to less
  goto Behavior2Forward                           '   Light

Behavior3Right:                                   '  Right CDS Cell
  Robot(TurnRight, 80msecs)                       '   Exposed to less
                                                  '   Light
Behavior3Forward:                                 '  Finished Turn, go
  if (Collision = True) then Behavior3End         '   Forward until
  Robot(Forward, 80msecs)                         '   Collision with
  if (Collision = False) then Behavior3End        '   object/wall

Behavior3End:                                     '  At object/wall
  End
```

The only substantial difference in the code is that the left CDS cell value is less than the right one, indicating the brighter light is to the left. As such, the robot turns right in Behavior 3. In the same situation in Behavior 2, the robot turns left, toward the brighter light source.

In fact, the code is so similar between the two behaviors that I use the same code for both behaviors. The only difference is that after reading the CDS cell values and comparing them, I check to see which behavior is active and instruct the robot to turn appropriately.

When this behavior has finished executing you will be surprised to discover that it works much better than the photovore's behavior. Whereas the robot's limited field of view caused the photovore behavior to chase reflections, the photophobe behavior is much better at finding the darkest point of the room.

The reason for the better performance of the photophobe is that it is very hard for it to find a middle ground between two bright spots. When a flashlight is put down on the floor as a light source, there is a central lighted area directly from the flashlight and secondary lighted areas caused by spherical aberrations in the bulb and the lens. These secondary lighted areas will often cause the photovore robot to move toward a lighted part of the room that is not necessarily the brightest.

One interesting aspect of Behavior 3 that you will notice is that the robot will shade itself from the light. I alluded to this earlier when I described the robot as having only a 160° field of view. Despite this, when you are running Behavior 3 with a flashlight on the floor, you will find that the battery, motors, and other features of the robot (including the Parallax AppMod) card will "shield" the light from the flashlight, creating artificial "shadows" that the robot may follow. These shadows will change as the robot moves, causing the robot to move in what seems to be a totally random fashion.

When the robot is executing Behavior 3, you may expect it to fall into the trap of being caught between two lighted areas generated by the flashlight. In practice, however, you will find that the robot will probably find the darkest part of the room. The reason for this difference is the difficulty in finding a balanced point as the robot approaches a wall. As the robot moves, the "scene" changes slightly, resulting in a different balance of light and dark. You may find that the robot will start off toward a dark area between two lighted areas (the "Expected End Point" of Figure 5-4), but it will usually move in a different direction as the aspect of the light changes.

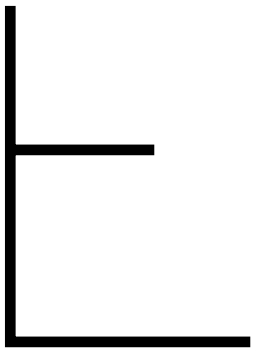

Figure 5-6 Behavior 4—"wall-hugging mode" icon.

Behavior 4—Wall-Following Mode

The origin of the last behavior is probably the least likely of the four to be built into the robot. When I had the first prototype robot working, I was trying to come up with a fourth behavior to build into the robot. Looking for a good idea, I asked my 6-year old daughter what she would like to see the robot do.

The answer was quite surprising. She said that she would like to see the robot solve a maze. She really did come up with an excellent idea. The next problem was how to program a robot that can solve a maze.

Very coincidentally, I read a "Flash" comic book in which the Flash was stranded in a giant maze (supercriminals often do that kind of thing to delay the hero long enough for them to carry out their nefarious deeds). In record time Wally West (the "Flash") got out of the maze, saved his wife, and captured the bad guy by remembering that you can always get out of a maze by keeping one hand on a wall of the maze at all times.

To illustrate how this works, consider the maze with the solution path shown in Figure 5-8. By keeping a hand on the right side wall at all times, you can move from the "Entrance" to the "Exit" without recording your previous position or guessing randomly.

This method of solving the maze is actually quite efficient. It solves the maze without going back over paths that have already been tried. The advantage of this method is that the current position of the robot does not have to be recorded, and you do not have to do something like drop "pellets" so the robot remembers where it has been.

The icon on the remote control of a partial maze (Figure 5-6) reflects that this behavior follows a wall, no matter where it leads.

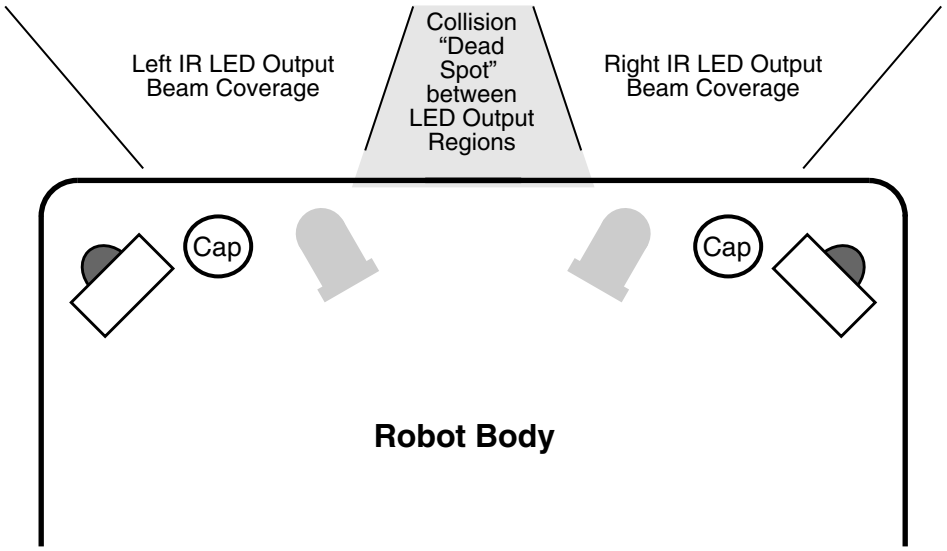

Figure 5-7 Collision "dead spot" in middle of robot.

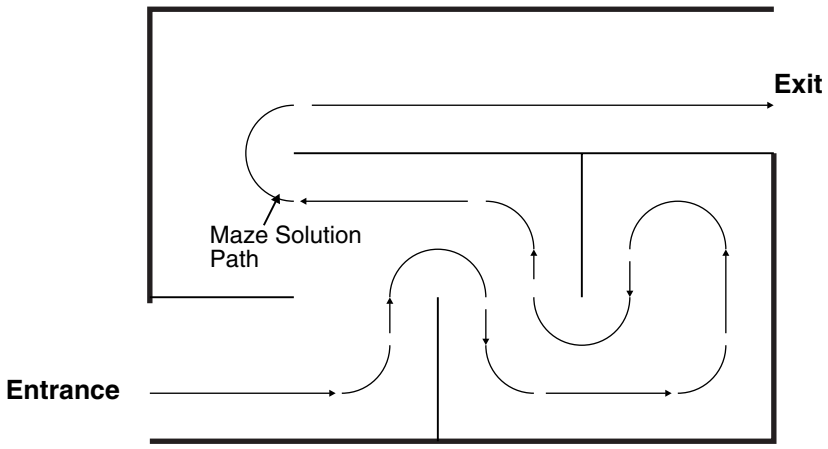

Figure 5-8 Solving a maze by keeping the right hand on a wall at all times.

The code required for this behavior is quite simple, but there are a few aspects of it that I would like to discuss with you.

```
Behavior4:                                          '  Wall Following
'
'  Follow the Wall/Work Inside the Maze
Behavior4Loop:                                      '  Repeat Here
  pause 80
  Robot(Forward, 80msecs)                           '   Move Forward

  if (LeftCollide = True) then Behavior4Left        '  If Object on
  if (RightCollide = True) then Behavior4Left       '   Left, or Object
                                                    '   on Right
  Robot(TurnRight, 80msecs)                         '   Else, Right
  goto Behavior4Loop
Behavior4Left:                                      '  Turn Left
  Robot(TurnLeft, 80msecs)
  if (LeftCollide = True) then Behavior4Left        '  Repeat while
                                                    '   Object to Left
  goto Behavior4Loop                                '  Else, Start Again
```

When this behavior executes, it does not stop on collision with an object—in fact, it is looking to be *constantly* colliding with a wall. To do this, the robot takes advantage of the IR collision detection system built into the robot to sense objects to the side.

When I originally programmed this behavior, I created two new movements. Instead of turning one wheel forward and one wheel back when executing a turn, I moved one wheel forward and left the other wheel stationary. This caused the robot to rotate around the stopped wheel and provided a much larger turn radius than turning both wheels in opposite directions.

This larger turn radius caused a few problems such as not being able to negotiate turns to the left while keeping the right sensor active. The fix to the problem was to use the traditional turning motions and then "blip" forward for 80 msec (see the "Robot(Forward, 80msecs)" instruction at the start of the application).

In the code, notice that I turn left if there is a collision to the right (turn away from the wall), or if there is a collision to the left. The collision to the left is probably surprising, but it is designed to deal with the case where the robot is following a wall, and it has a perpendicular wall in front of it. If there is a wall directly in front of the robot, it will continually turn away from it to the left. This allows the robot to make very tight turns to the left.

It is interesting to watch the robot run along a wall in your house with Behavior 4 active, but I find it gets old pretty fast. You can create your own maze for the robot to solve using a few paperback books lying on their backs. I found that if I made each maze "grid" square, the length of two books on a side, the robot could run it quite well, and the total maze would not take up a large amount of space.

If you use paperback books, I recommend that the paper edges rather than the shiny spine is detected by the robot. The paper reflects the IR light back to the IR detectors quite well, while the shiny cover material may reflect the light away from the detectors.

While the book solution is very easy to implement, you may want to try out other types of walls, such as paper, bricks, or pieces of wood (sections of 2 by 4 are ideal). If you use these different types of walls, remember that the grid square size should be at least 12 inches (30 cm) on a side.

Artificial Intelligence

For me, the most interesting and exciting aspect of working with robots is the opportunity to experiment with artificial intelligence programs and look at different ways in which robots can "think." In TV and movies, making robots think is generally a relatively simple affair—in real life, nobody has done it yet. There seem to be some basic aspects of intelligence that we have not yet discovered. Even when we do, I believe that there will be some issues that are going to make it very difficult to program. In this section, I want to discuss my thoughts on what artificial intelligence is along with some ideas on how it should be implemented.

While there are many different research thrusts in artificial intelligence, I do not know of a single definition for what "artificial intelligence" actually is. This gives me the unique opportunity to define the term myself as:

> A computer program which responds to external stimulus in a similar way to organic beings.

This definition probably seems very simple and broad, but I feel that such a definition needs to be based on the number of different areas in which artificial intelligence research is taking place. A brief search on the Internet reveals that there are projects going on in the following areas:

- Robot locomotion
- Robot limb and gripper design and control
- Artificial vision and object recognition
- Perception of nearby objects
- Speech discrimination
- Spacecraft control

- Knowledge representation
- Belief programming
- Map navigation
- Hardware diagnosis
- Neural networking
- Disease diagnosis

It is interesting (and important) to note that each one of these topics is handled by humans almost effortlessly. Most of the capabilities in this list are built into animals and humans for virtually no cost while millions (and maybe billions) of dollars have been spent trying to create machines that mimic these natural actions.

The amount of effort that has been expended and the limited results so far tells me that there is something special about life. Maybe part of the problem is that living organisms have been evolving for billion of years compared to the less than 50 years for mechanical organisms.

In my definition above, note that I do not discuss what goes on inside the artificial intelligence computer program; I say that the program *responds* in a similar manner. I do not discuss how the "thought" process is implemented in the application.

Many artificial intelligence theories and experiments are based on creating a database of different behaviors and some method of logic for selecting the appropriate behavior for a given situation. These behaviors are identical to the behaviors I have discussed at the start of this chapter. Breaking down the program into simpler and simpler behaviors and decision trees will give you an application that appears "intelligent," but it is, in fact, still just a program and in no way "artificially intelligent."

Probably the best example of this is Behavior 4—it carries out quite an intelligent task (solving a maze), but it is really just executing a program. When children solves a maze, they are using learned rules to solve the problem, when the computer does it, it is simply executing a program that a human has programmed into it.

When I was in college, I took a course in "artificial intelligence" where my final project was to come up with a program that found the most efficient route between two places in Kitchener–Waterloo, Ontario, Canada.

First I laboriously keyed in a representation of the street map for the two cities into the computer system, along with a representation of the length of the street segments, one-way situations, and stop signs or traffic lights at intersections. Next, I was able to develop a program that would figure out every possible path between the two points. To illustrate the process, I am going to use the simple example shown in Figure 5-9 in which the object is to move from point "A" to point "B."

To find the number of paths from point "A" to point "B," I started at point "A" and worked through every possible combination and halted the combination if there was a dead end or it repeated upon itself. For the Figure 5-9 example, the valid street routes are

```
1, 5
1, 4, 6, 7
2, 3, 6, 7
2, 3, 4, 5
```

Note that this list ignores invalid paths like: 1, 4, 3, 2 and 2, 3, 4, 1 because they return to point "A."

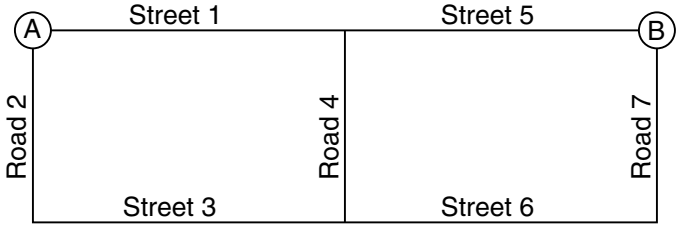

Figure 5-9 Street map showing different paths between point "A" and point "B."

This operation seems simple, but consider the situation of "twin cities" that are home to 300,000 people. This operation took anywhere from 45 minutes to one and a half hours on a VAX 870 and often produced several *hundred thousand* possible routes.

With the list of valid routes calculated, the program then went through the process of finding the shortest route by summing the time required to travel through each segment to the next one. This calculation included factors for the speed at which a car could travel on the segment and whether or not there are traffic lights at the intersection.

The calculation is very simple for the four routes of the Figure 5-9, but consider the number of calculations required for the actual application. This second part of the operation would take at least an hour.

The real failure for me in this application was that I could not come up with a way for the computer to determine which routes were obviously unacceptable. A human looking at a map can come up with a half dozen routes that would be suitable for better analysis, but the computer program had to work through each one individually.

Despite this, there was on upside to this project. When I ran it I discovered that a number of routes I had been using for a long time were actually slower than other routes selected by the computer. In testing out these routes, I discovered that the computer was actually right. The irony of the situation was that it took two hours or more of computer time to calculate the best route for a 10-minute trip.

This problem is known as the "traveling salesman problem" in computer science circles and coming up with a solution to this problem is of vital importance—particularly in the area of networking and telecommunications. As humans, we can "see" the most expedient routing for a phone call or data packet; a computer has to test every possible path to find the best solution. As in the case of my computer program, the time required to find the best path takes many times that of the actual trip.

This "brute force" approach to solving the problem of finding the best route is common to many different types of artificial intelligence applications. The long processing time is not that unusual (at the end of each term, the VAX system response went down to just about zero as the artificial intelligence class tested out their applications). This approach does, however, avoid the major advantage of organic beings: their ability to look at a problem and relate the data in such a way that the answer is produced quickly and is quite logical.

This is the reason for looking at using "behaviors" for providing responses from artificial intelligence systems. Instead of beating through all the data, a simple set of rules built into a behavior will provide a very similar response to the brute force approach but with much less computing resources required.