65
REM MAZE WALKER IN BASIC

FIXING THE MAZE
WALKING THE MAZE
TOUCHING THE MAZE
TESTING THE MAZE

Downloaded from http://direct.mit.edu/books/chapter-pdf/2025481/c001500_9780262305501.pdf by guest on 01 December 2022

10 PRINT can be appreciated purely for its visual qualities—its regular asymmetry, its determined ranging over and across the screen, and even its colors, two shades of blue that can be pleasing. But 10 PRINT can also be interpreted as a maze, a labyrinth with routes and potentially with a solution. One might even wander through the maze, tracing a path with one's eyes, a finger, or some computational procedure.

What would such a computational procedure, and a program that supports its use, look like?

To see the answer, this section uses a software studies approach, writing programs to interpret other programs. It takes this approach to the extreme and builds a large program, using 10 PRINT as the starting point. Just as literary scholars study a text by generating more texts, it is productive to study software by coding new software. In this particular case, it's possible to develop a series of hermeneutic probes in Commodore BASIC—probes of increasing complexity, programs that transform 10 PRINT's output into a stable, navigable, and testable maze.

FIXING THE MAZE

The first step in this process is to freeze the pattern so that it can be contemplated as a fixed maze. 10 PRINT, of course, produces an endlessly scrolling sequence of two symbols, an animated effect lost in the static images shown in this book. For at most an instant—after the screen has filled and the lower-right character has been drawn, but before the pattern has scrolled up to make room for the next line—is there ever a rectangular maze pattern filling the entire screen within the border.

To draw a stable rectangular maze pattern, 10 PRINT must be modified to draw a finite number of symbols, rather than an infinite sequence. As described in the chapter Regularity, the program must use a bounded rather than unbounded loop, placing characters on the screen a set number of times. To fill the forty columns and twenty-five rows, 1,000 characters must be drawn (40 × 25 = 1000).

This task can be accomplished using the FOR . . . NEXT construct discussed in the Regularity chapter. Here is a program that uses PRINT to output exactly 1,000 characters:

Downloaded from http://direct.mit.edu/books/chapter-pdf/2025481/c001500_9780262305501.pdf by guest on 01 December 2022

```
10 FOR I=1 to 1000
20 PRINT CHR$(205.5 + RND(1));
30 NEXT I
```

As might be expected from observation of 10 PRINT, the screen scrolls up when the last character is printed; in this case, there are four lines at the bottom that lack the maze pattern. Furthermore, once the program ends, the "READY." prompt appears with a blinking cursor stationed after it.

Trying to avoid this nonmaze text, one could add 40 GOTO 40 at the end of the program. This would create a continuous loop that did nothing but keep the program from terminating. This valiant attempt fails; "READY." and the blinking cursor are avoided, but a two-line gap still appears at the bottom of the screen. Changing "1000" in line 10 to "999" moves the program closer to the goal; everything but the lower-right character is drawn, and there are no blank lines at the bottom. But the program is still one character away from completely filling the screen with the maze.

As discussed in the chapter The Commodore 64, PRINT invokes the operating system's CHROUT routine with its automatic scrolling and eighty-character logical lines. When the one-thousandth character is printed (at the eightieth character of the last logical line on the display), the screen scrolls up by two physical (forty-character) lines to make room for the next eighty-character logical line. To generate a complete screen of a stable maze, it is necessary to use a mechanism other than the virtual Teletype provided by PRINT and the CHROUT routine it invokes.

To create a fixed screen-sized maze, a program can directly place PETSCII character codes into the computer's video memory. Rather than iterating from one to 1,000, the FOR loop must iterate though the 1,000 characters as locations in video memory, which begin at memory location 1024 and end 1,000 characters later at 2023. Because these invocations of POKE rely on memory locations rather than character codes, this modified program must also refer the correct screen codes for the diagonal-line characters (77 and 78), rather than the 205 and 206 values that are the PETSCII codes used in the CHR$ statement. This same use of 77 and 78 was seen in the POKE variation near the end of the Variations in BASIC remark.

```
10 FOR I=1024 TO 2023
20 POKE I,77.5+RND(1)
```


Downloaded from http://direct.mit.edu/books/chapter-pdf/2025481/c001500_9780262305501.pdf by guest on 01 December 2022

```
30 NEXT I
40 GOTO 40
```

One final nicety can be added: a standard statement at the beginning to clear the screen, `PRINT CHR$(147);`. This is not strictly necessary for this program, since the full screen will be overwritten one way or the other with a maze, but it makes the initial unfolding of the maze look a bit neater. It actually helps in the next step and in future programs, because this statement also restores color memory, cleaning up the traces of previous walks of the maze.

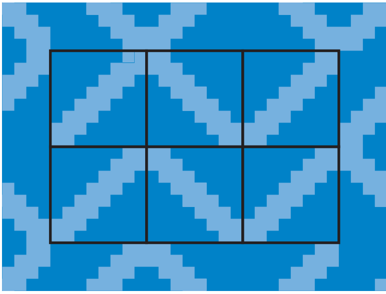
WALKING THE MAZE

Now that code has been developed to draw a stable full-screen maze pattern, work can begin on a program that treats this pattern as a maze and "walks" it, moving through it with respect for the "walls" set up by the two characters. The first step is to determine a location within the maze. Viewers will often interpret the lighter slanting characters as thin walls and the dark blue background as the floor, although the opposite interpretation is possible. The program discussed here considers the light, thinner lines to be walls.

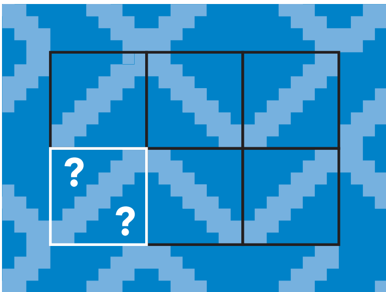
The first step in operationalizing this view of the maze—that is, in creating a computational system that functions in a way that is consistent with this interpretation—involves defining what it means to occupy a location within the maze. How can a "walker" be placed at a particular point in the maze?

The challenge is that the visual distinction between walls and floor is not explicitly represented in the program. A close-up of the maze pattern, with black outlines around the individual characters, each of which is plotted out on an 8 × 8 matrix, shows these distinctions. The dark blue is the background of characters, but positions within the dark blue "corridor" have no unique character locations. Dark-blue and light-blue areas of the screen are distinguished at the level of individual pixels, but in the graphics mode used, it is only possible to manipulate the larger 8 × 8 pixel characters:

Downloaded from http://direct.mit.edu/books/chapter-pdf/2025481/c001500_9780262305501.pdf by guest on 01 December 2022



Designating a particular screen location (such as the highlighted location in maze below) would identify one of the slanting characters (a wall segment), but would not identify which side of the wall is currently occupied:



Given a diagonal wall location, it's possible to imagine someone approaching that wall from above, below, left, or right—that is, along a particular course or heading. The walker, in this view, would ricochet off the wall along particular headings. Approaching a right-leaning diagonal from above or from the left implicitly indicates that the walker is in the corridor segment above the wall, while approaching from below or from the right suggests the walker is in the corridor segment below the wall. These relationships are reversed for the left-leaning diagonals. In this view, in addition to a particular X, Y location, a third piece of information—a heading, or particular direction of movement—can be used to uniquely identify the maze location and where the walker will go next:

Downloaded from http://direct.mit.edu/books/chapter-pdf/2025481/c001500_9780262305501.pdf by guest on 01 December 2022

Given an initial location and a heading, the walker moves through the maze in a sort of drunken (or very determined) walk, not unlike the first run of Claude Shannon's Theseus mouse through its maze of relays and switches. In the case of the "Maze Walker" program here, the walker encounters and bounces off the walls in the manner depicted:

The BASIC code for "Maze Walker" is as follows:

```
10 REM PRODUCE A STABLE MAZE
20 PRINT CHR$(147)
30 FOR I=1024 TO 2023
40 POKE I,77.5+RND(1)
50 NEXT I

100 REM SET INITIAL X AND Y WALKER LOCATION AND DIRECTION
110 REM DIRECTION IS EITHER 0 LEFT, 1 RIGHT, 2 UP, 3 DOWN
120 X=INT(RND(0) * 39) : XOLD=-1
130 Y=INT(RND(0) * 24) : YOLD=-1
140 DIR=INT(RND(0) * 3)
```


Downloaded from http://direct.mit.edu/books/chapter-pdf/2025481/c001500_9780262305501.pdf by guest on 01 December 2022

```
150 WOLD=-1
160 GOSUB 500

200 REM START WALKING MAZE USING RULES FOR BOUNCING OFF WALLS
210 REM COMPUTE NEW LOCATION BASED ON INITIAL DIRECTION
220 IF DIR=0 THEN X=X - 1 : GOTO 270
230 IF DIR=1 THEN X=X + 1 : GOTO 270
240 IF DIR=2 THEN Y=Y - 1 : GOTO 270
250 IF DIR=3 THEN Y=Y + 1

260 REM DETERMINE IF THE WALKER IS OFF THE SCREEN
270 IF X >= 0 AND X <= 39 AND Y >= 0 AND Y <= 24 THEN GOTO 300
280 GOSUB 600 : GOSUB 650
290 GOTO 10

300 REM BOUNCE OFF WALL AS FUNCTION OF DIRECTION
310 REM 77 IS \, 78 IS /
320 WALL=PEEK(1024 + X + (Y * 40))
330 IF WALL=78 THEN GOTO 380
340 IF DIR=0 THEN DIR=2 : GOTO 420
350 IF DIR=1 THEN DIR=3 : GOTO 420
360 IF DIR=2 THEN DIR=0 : GOTO 420
370 IF DIR=3 THEN DIR=1 : GOTO 420
380 IF DIR=0 THEN DIR=3 : GOTO 420
390 IF DIR=1 THEN DIR=2 : GOTO 420
400 IF DIR=2 THEN DIR=1 : GOTO 420
410 IF DIR=3 THEN DIR=0
420 GOTO 160

500 REM DRAW WALKER, RESTORING PREVIOUS WALL CHARACTER
510 GOSUB 600
520 XOLD=X : YOLD=Y
530 M=1024 + X + (Y * 40)
540 WOLD=PEEK(M)
550 C=55296 + X + (Y * 40)
560 POKE C, 1 : POKE M, 87
570 GOSUB 650
```


Downloaded from http://direct.mit.edu/books/chapter-pdf/2025481/c001500_9780262305501.pdf by guest on 01 December 2022

```
580 RETURN

600 REM RESTORE WALL AT PREVIOUS WALKER LOCATION
610 IF XOLD=-1 THEN GOTO 630
620 POKE 1024 + XOLD + (YOLD * 40), WOLD
630 RETURN

650 REM PAUSE FOR 500 LOOPS
660 FOR I=1 TO 500 : NEXT I
670 RETURN
```

Because it is written in BASIC, the code to "Maze Walker" is fairly legible, even if it is significantly longer than BASIC programs discussed so far. A line-by-line explication will highlight the process by which "Maze Walker" walks the maze. The program begins with lines 20 through 50, filling the screen with a random maze as described in the last section.

Line 120 initializes a random horizontal (X) location between 0 and 39, representing the forty columns across the screen. The range 0 to 39 is used instead of 1 to 40 because this X value indexes a location in video memory; counting from 0 more directly corresponds to memory locations.

The variable `OLDX` holds the previous X coordinate of the walker. Initially, since a new X coordinate has just been initialized, there is no old value—so the X coordinate is set to an invalid value, −1. A common technique when dealing with a variable that can take a range of values, this method allows the variable to be easily tested to determine whether it has a valid value yet. Similarly, line 130 initializes a random Y coordinate between 0 and 24 (for the twenty-five rows on the screen), and initializes `OLDY`, the previous Y location, to −1, since there is no previous Y coordinate.

Line 140 sets the initial heading to a number between 0 and 3; the program will interpret 0 as left, 1 as right, 2 as up, and 3 as down. `WOLD`, initialized in line 150, stores the value of the screen code at the given location. The program "remembers" the location, so that the maze wall can be redrawn after the walker has passed.

Line 160 jumps to a subroutine at line 500. This program has three subroutines: one to draw the current location of the walker, changing the color of walls that have been bumped into; one to redraw the wall after the walker has passed; and one that simply pauses (using a loop that does

Downloaded from http://direct.mit.edu/books/chapter-pdf/2025481/c001500_9780262305501.pdf by guest on 01 December 2022

nothing) so that the walker's movement is not too fast. The GOSUB at line 160 jumps to the first draw subroutine, pinpointing the initial location of the walker.

Lines 220 through 250 determine the next position of the walker (as an X, Y coordinate) by referring to the walker's heading. Leftward movements decrease the X value, rightward movements increase it; upward movements decrease the Y value, downward movements increase it. For the Y values, this change is the opposite of the standard Cartesian grid, in which the 0,0 coordinates rest in the lower left-hand corner. Screen coordinates commonly begin in the upper left-hand corner, just as CRT monitors scan the screen from left to right and top to bottom.

Lines 270 through 290 define what happens if the walker runs off the edge of the screen. Line 270 uses an conditional statement, an IF . . . THEN statement, to test whether the walker has a legal position on the screen; if it does, the program jumps to line 300, where a new heading for the walker is determined. Otherwise, two subroutines are called. These restore the wall at the walker's last location and wait for a short span of time. Line 290 then jumps back to the beginning of the program, drawing a new maze and re-initializing the walker at a random location.

Lines 320 through 410 determine the new heading of the walker using the current location's wall segment and the current heading. Line 320 uses the PEEK command to see what is in video memory—what character is stored at the current location. In this line, the 2D grid of the screen is rolled up into one-dimensional video memory. Screen location 0,0 in the upper left-hand corner corresponds to the first location in video memory, 1024. Each line of forty characters corresponds to a range of forty memory locations, with each group of forty following each other successively in memory. So multiplying the vertical Y coordinate by forty, and adding the horizontal X coordinate, yields the appropriate location in video memory.

Each of the four headings resolves into one of four new headings for a right-leaning diagonal character and one of four new headings for a left-leaning diagonal character. The eight IF . . . THEN statements at lines 340 to 410 handle each of these eight cases. The IF . . . THEN at line 330 jumps to the second group of four IF . . . THEN statements for a right-leaning diagonal character, allowing program execution to fall through to the first group of IF . . . THEN statements for the other character. The GOTO statements at the end of each line jump over the rest of the IF . . .

Downloaded from http://direct.mit.edu/books/chapter-pdf/2025481/c001500_9780262305501.pdf by guest on 01 December 2022

THEN statements once the correct new heading has been set.

Line 420 is the last line of the main loop. It loops back to start the process of drawing the walker at its current location, and updating location and heading, all over again.

The subroutine at line 500 draws the walker at its current location and redraws the wall in the location that it just left. At the beginning, in line 510, there is a call to the subroutine at line 600, placing the correct wall character in the old position of the walker. Then, the subroutine saves the current X, Y to the old location XOLD, YOLD. Line 540 computes the location in video memory (M) for the current X, Y location. This memory location is used twice: on line 530 to save the current character at this location, and in the second POKE on line 560 to change this character to a new character representing the walker. It would be ideal to use a character that shows the walker standing next to the wall, but there is no character in the standard character set that combines a diagonal line with a shape next to it. It is possible to define custom characters for the four combinations of walls with walkers, but this program uses the built-in character with screen code 87 to represent the walker. This has the disadvantage that from a static screen shot that walker's exact maze location is visually ambiguous. While watching the walker move as the program executes, however, the location is discernible from the pattern of movement.

Line 550 computes the memory location in color memory given the X, Y screen location. There are 1,000 bytes of color memory, as with video memory. The effect of values in color memory on the display depends on the graphics mode. In character mode (used in 10 PRINT and in this program), each location in color memory stores a color code that tells the system what color should be used to draw the character indicated by the screen code in the corresponding location in video memory. The first POKE on line 560 stores a color code of 1, which draws the corresponding screen code using the foreground color white. Finally, line 570 makes a nested call to the subroutine at 650, which adds a delay to the maze walker, making it easier to observe the details of the walker's movement.

The subroutine at line 600 redraws the wall character from the maze walker's previous location. Without this subroutine, the walker would leave a trail behind it, slowly replacing the walls of the maze. The IF . . . THEN at line 610 tests whether the previous location is a valid location, which it is not on the first call, when XOLD is initialized to −1. Although the wall is

Downloaded from http://direct.mit.edu/books/chapter-pdf/2025481/c001500_9780262305501.pdf by guest on 01 December 2022

restored as the walker passes by, the color code in color memory is not restored. This means that the redrawn wall will appear in white, leaving a trail of white walls to mark the walker's passage.

Finally, the subroutine at line 650 adds a delay between each step through the maze. The `FOR` loop contains no statements before the `NEXT`; it simply counts to 500. To increase or decrease the delay time, this value can be increased or decreased.

There are a number of observations to make about the `10 PRINT` maze, the representational properties of BASIC, and the Commodore 64 environment based on the development of "Maze Walker." First, it takes considerable effort to transform the visual perception of a maze with walls and a floor into a practical functioning model of this perception. Decisions must be made about what it means to hold a location in the maze and to move through it. This program sharpens the somewhat vague visual perception of "mazeness" into a highly detailed understanding of the local structure of the maze.

Second, the representation of movement requires repeatedly drawing and erasing a shape (the representation of the walker), with the need to remember what lies "under" the shape so that the occluded object can be correctly redrawn. This basic principle of continuously drawing and erasing static snapshots to produce the illusion of movement is a fundamental feature of modern media, seen in everything from the latest Pixar movie to the latest blockbuster Xbox game. The related principle of collision with virtual objects, when combined with the representation of movement, defines graphical logic, a representational trope that underlies the computer's ability to represent virtual spaces. In the compressed form of "Maze Walker," there are specific lines that encode the concept of collision with walls: lines 320 through 410.

Finally, the ability to observe walks through the maze brings clarity to the structure of the `10 PRINT` maze. A typical (stabilized) `10 PRINT` maze consists of loops of various lengths that are interspersed with runs connecting two locations on the edge of the maze. The pattern therefore consists of multiple, intertwined unicursal mazes; once embarked on a particular path from edge to edge, there are no choices to make. A `10 PRINT` maze might be considered multicursal if there is a choice of where to enter the maze from one of the outside "openings," but once such a choice is made, the path will lead irrevocably to its paired entrance or exit.

Downloaded from http://direct.mit.edu/books/chapter-pdf/2025481/c001500_9780262305501.pdf by guest on 01 December 2022

TOUCHING THE MAZE

While “Maze Walker” allows the user to watch a computer “other” navigate the maze, a program can turn this spectacle into an interactive environment. Here, computation acts as a prosthesis, or extension of the user’s sense of touch, presenting the user with solid walls that constrain navigation.

```
10 REM PRODUCE A STABLE MAZE
20 PRINT CHR$(147)
30 FOR I=1024 TO 2023
40 POKE I,77.5+RND(1)
50 NEXT I

100 REM SET INITIAL X AND Y WALKER LOCATION AND DIRECTION
110 REM DIRECTION IS EITHER 0 LEFT, 1 RIGHT, 2 UP, 3 DOWN
120 X=INT(RND(0) * 39) : XOLD=-1
130 Y=INT(RND(0) * 24) : YOLD=-1
140 DIR=INT(RND(0) * 3)
150 WALL=-1
160 GOSUB 500

200 REM WAIT FOR LEGAL MOVE GIVEN LOCATION AND DIRECTION
210 GET A$ : IF A$="" GOTO 210
220 IF A$=" " THEN GOSUB 600 : GOTO 120 : REM HYPERSPACE
230 REM 77=\, 78=/ 0 LEFT, 1 RIGHT, 2 UP, 3 DOWN
240 IF WALL=78 THEN GOTO 310
245 REM UP
250 IF (DIR=0 OR DIR=3) AND ASC(A$)=145 THEN DIR=2 : GOTO 400
255 REM RIGHT
260 IF (DIR=0 OR DIR=3) AND ASC(A$)=29 THEN DIR=1 : GOTO 400
265 REM DOWN
270 IF (DIR=1 OR DIR=2) AND ASC(A$)=17 THEN DIR=3 : GOTO 400
285 REM LEFT
290 IF (DIR=1 OR DIR=2) AND ASC(A$)=157 THEN DIR=0 : GOTO 400
300 GOTO 200

310 REM DOWN
```


Downloaded from http://direct.mit.edu/books/chapter-pdf/2025481/c001500_9780262305501.pdf by guest on 01 December 2022

```
320 IF (DIR=0 OR DIR=2) AND ASC(A$)=17 THEN DIR=3 : GOTO 400
330 REM RIGHT
340 IF (DIR=0 OR DIR=2) AND ASC(A$)=29 THEN DIR=1 : GOTO 400
350 REM UP
360 IF (DIR=1 OR DIR=3) AND ASC(A$)=145 THEN DIR=2 : GOTO 400
370 REM LEFT
380 IF (DIR=1 OR DIR=3) AND ASC(A$)=157 THEN DIR=0 : GOTO 400
390 GOTO 200

400 IF DIR=0 THEN X=X - 1 : GOTO 450
410 IF DIR=1 THEN X=X + 1 : GOTO 450
420 IF DIR=2 THEN Y=Y - 1 : GOTO 450
430 IF DIR=3 THEN Y=Y + 1

450 REM DETERMINE IF THE WALKER IS OFF THE SCREEN
460 IF X >= 0 AND X <= 39 AND Y >= 0 AND Y <= 24 THEN GOTO 160
470 GOSUB 600
480 GOTO 10

500 REM DRAW WALKER, RESTORING PREVIOUS WALL CHARACTER
510 GOSUB 600
520 XOLD=X : YOLD=Y
530 M=1024 + X + (Y * 40)
540 WALL=PEEK(M)
550 C=55296 + X + (Y * 40)
560 POKE C, 1 : POKE M, 87
570 RETURN

600 REM RESTORE WALL AT PREVIOUS WALKER LOCATION
610 IF XOLD=-1 THEN GOTO 630
620 POKE 1024 + XOLD + (YOLD * 40), WALL
630 RETURN
```


Downloaded from http://direct.mit.edu/books/chapter-pdf/2025481/c001500_9780262305501.pdf by guest on 01 December 2022

The change between this and the previous walker is found in lines 200 through 390. These lines replace the code that changed the walker's heading given the current heading and the wall type. Now the program reads the keyboard, looking for arrow keys. The user can use the arrow keys to move backward and forward along the current path as allowed by the wall at the current location and the current heading. Line 210 uses the `GET` statement to read a character from the keyboard. If no key has been pressed, `GET` returns the empty string. The `IF . . . GOTO` in the second statement on line 210 loops continuously until a key has been pressed.

Line 220 tests whether the spacebar has been pressed. If so, the subroutine at line 600 that redraws the wall at the current walker location is called, and the program jumps to 120, initializing the current location and heading to new random values. This allows the user to jump to a new location after exploring the current path to the edge of the screen, or after completing a loop.

Line 240 selects between the four different cases for ╲ and ╱. Consider the cases for ╲, when `WALL` is 77. If the current heading is left or down, the walker is on the left side of the slash. The valid headings to move are right and up. If the current heading is right or up, the walker is on the right side of the slash. The valid headings to move are left or down. Now consider the cases for ╱, when `WALL` is 78. If the current heading is left or up, then the walker is on the right side of the slash. The valid headings to move are down or right. If the current heading is right or down, the walker is on the left side of the slash. The valid headings to move are up or left.

The eight `IF . . . THEN` statements from 250 through 380 handle these eight cases, checking whether the user has hit an arrow key corresponding to a valid heading given the current heading. If a key other than space or an arrow key is hit, or if the arrow key is not valid given the current wall and heading, control will fall through to 300 or 390, and the program will loop back to 200 to continue scanning the keyboard. Thus, the walker only moves when the user pushes an arrow key in a valid heading, enforcing the "solidity" of the walls and responding only to valid input.

The interactive maze walker allows the user to trace a finger along the maze pattern, kinesthetically experiencing the ricocheting movement employed by the maze walker. The ability to jump randomly about the maze allows the user to explore many paths in the same maze, observing how the various loops and trails intertwine.

Downloaded from http://direct.mit.edu/books/chapter-pdf/2025481/c001500_9780262305501.pdf by guest on 01 December 2022

TESTING THE MAZE

What would it mean for the 10 PRINT maze to have a solution? Given that the only choice to be made is outside the maze, in choosing an entry point, one definition of a solution would be a path that leads all the way from one side of the maze to the other. Solving the maze would, in this case, consist of choosing the right entry point to make it all the way to the other side.

This question of solutions is just one example of the more general question of determining maze properties. One could as easily be interested in mazes that have really long loops, or as many loops as possible, or as many side-to-side paths as possible, or lots of really short paths, and so forth. Is it possible to computationally recognize such properties, so that the design space of 10 PRINT mazes can be explored and mazes can be generated with specific properties?

The perhaps-surprising answer is yes. Computer science offers a general approach to such problems called generate and test. It is based on the observation that, while directly generating a solution to a problem is generally difficult, recognizing whether a proposed solution is in fact a solution is easy. Therefore, to solve problems, or to generate artifacts with desired properties, one approach is to use a relatively simple generator to generate candidates and then test them to see if they have the desired property. For 10 PRINT, this means generating random maze patterns (as explored throughout this book), and then testing them to see if they have the desired property. In the explorations that led to this book, the authors wrote programs as a method for better understanding 10 PRINT. The generate and test paradigm provides a framework for extending this practice by writing programs to analyze the output of 10 PRINT.

To illustrate this approach, here is a program that looks for mazes with solutions, that is, with a path from one side to the other. While searching for a path, the program systematically tries every left-hand and upper entrance into the maze, testing whether this passage goes through to the other side. As paths are searched, walls are changed to white. If a solution is found, the maze is redrawn in its original color with just the solution path redrawn in white, to allow the user to behold the maze with a solution in its purity, before randomly generating a new maze to test. If every path is explored with no solution found, a new maze is generated and the search begins anew.

Downloaded from http://direct.mit.edu/books/chapter-pdf/2025481/c001500_9780262305501.pdf by guest on 01 December 2022

```
10 DIM B1(3),B2(3) : REM 'BOUNCE' ARRAYS
20 B1(0)=2 : B1(1)=3 : B1(2)=0 : B1(3)=1
30 B2(0)=3 : B2(1)=2 : B2(2)=1 : B2(3)=0

40 REM PRODUCE A STABLE MAZE
50 PRINT CHR$(147)
60 FOR I=1024 TO 2023
70 POKE I,77.5+RND(1)
80 NEXT I

90 REM TEST: SOLUTIONS MUST BE PATHS ACROSS WIDTH OR HEIGHT
100 FOR S=0 TO 24
110 X=-1 : Y=S : DIR=1 : XOLD=-1 : YOLD=-1 : WOLD=-1
120 SX=X : SY=Y: SD=DIR
130 GOSUB 410 : GOSUB 520
140 IF X > 39 THEN GOTO 290 : REM FOUND A SOLUTION
150 IF X < 0 OR Y < 0 OR Y > 24 THEN GOTO 180
160 GOSUB 610
170 GOTO 130
180 GOSUB 710 : NEXT S
190 FOR S=0 TO 39
200 X=S : Y=-1 : DIR=3 : XOLD=-1 : YOLD=-1 : WOLD=-1
210 SX=X : SY=Y : SD=DIR
220 GOSUB 410 : GOSUB 520
230 IF Y > 24 THEN GOTO 290 : REM FOUND A SOLUTION
240 IF X < 0 OR Y < 0 OR X > 39 THEN GOTO 270
250 GOSUB 610
260 GOTO 220
270 GOSUB 710 : NEXT S
280 GOTO 50
290 FOR I=55296 TO 56295 : POKE I,14 : NEXT I
300 X=SX : Y=SY : XOLD=-1 : YOLD=-1 : WOLD=-1 : DIR=SD
310 GOSUB 410 : GOSUB 520 : GOSUB 610
320 REM DETERMINE IF WE’RE OFF THE SCREEN
330 IF X >= 0 AND X <= 39 AND Y >= 0 AND Y <= 24 THEN GOTO 360
340 GOSUB 710 : GOSUB 800
350 GOTO 50
```


Downloaded from http://direct.mit.edu/books/chapter-pdf/2025481/c001500_9780262305501.pdf by guest on 01 December 2022

```
360 GOTO 310

400 REM COMPUTE NEW LOCATION BASED ON INITIAL DIRECTION
410 IF DIR=0 THEN X=X - 1 : GOTO 450
420 IF DIR=1 THEN X=X + 1 : GOTO 450
430 IF DIR=2 THEN Y=Y - 1 : GOTO 450
440 IF DIR=3 THEN Y=Y + 1
450 RETURN

500 REM BOUNCE OFF CURRENT WALL AS FUNCTION OF DIRECTION
510 REM 77=\, 78=/
520 WALL=PEEK(1024 + X + (Y * 40))
530 IF WALL=77 THEN DIR=B1(DIR) : GOTO 550
540 IF WALL=78 THEN DIR=B2(DIR)
550 RETURN

600 REM DRAW WALKER, RESTORING PREVIOUS WALL CHARACTER
610 GOSUB 710
620 XOLD=X : YOLD=Y : I=X + (Y * 40)
630 M=1024 + I
640 WOLD=PEEK(M)
650 C=55296 + I
660 POKE C, 1 : POKE M, 87
670 RETURN

700 REM RESTORE WALL AT PREVIOUS WALKER LOCATION
710 IF XOLD=-1 THEN GOTO 730
720 POKE 1024 + XOLD + (YOLD * 40), WOLD
730 RETURN

800 FOR I=1 TO 2000 : NEXT I
810 RETURN
```

The two biggest differences from the initial “Maze Walker” are the line blocks 100–180 and 190–270. Lines 100–180 systematically set the initial position to a character on the left-most side of the maze, and the heading to right. A solution is detected if the walker runs out the right-hand side of

Downloaded from http://direct.mit.edu/books/chapter-pdf/2025481/c001500_9780262305501.pdf by guest on 01 December 2022

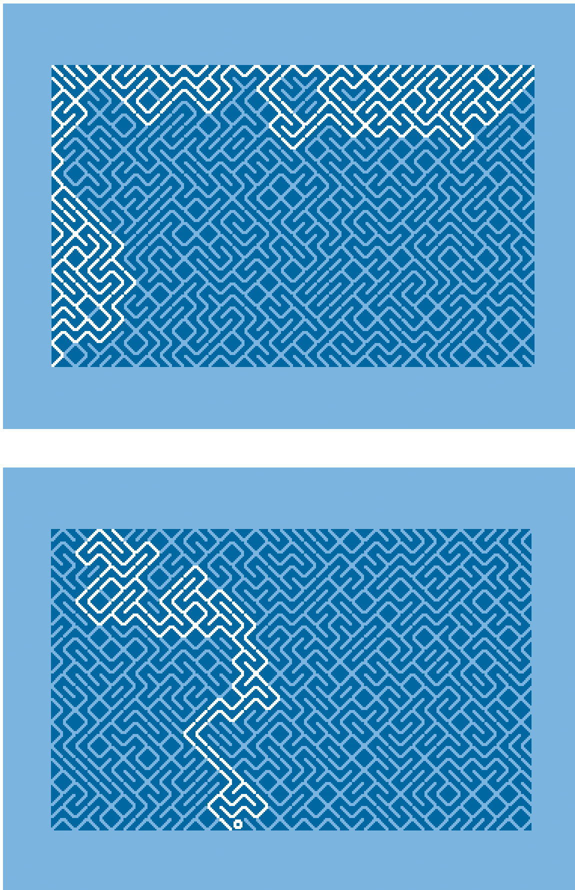


Figure 65.1

“Maze Walker” can determine whether a maze has solution (top) or not (bottom).

the maze. Lines 190–270 systematically set the initial position to a character on the top of the maze, and the heading to down (entering the maze). A solution is detected if the walker runs out the bottom of the maze. Figure 65.1 provides an example of a maze with no solutions and an example of a maze that has a solution.

Downloaded from http://direct.mit.edu/books/chapter-pdf/2025481/c001500_9780262305501.pdf by guest on 01 December 2022