Lesson

LEARN THE RED PATH - STACK DATA STRUCTURES

CREATED BY

Richard Born
Associate Professor Emeritus
Northern Illinois University
rborn2@niu.edu

TOPICS

Programming

GRADES

7-12

METHOD

OzoBlockly

DURATION

60 minutes

Learning a Red Path in a Binary Tree Maze
A Lesson on the Stack Data Structure

By Richard Born
Associate Professor Emeritus
Northern Illinois University
rborn2@niu.edu

Introduction

In this lesson, students will study how to make Ozobot Bit learn and then follow a red path that begins at the root of a binary tree maze. The map that is used is from the OzoBlockly *Intermediate Mode Examples* at ozoblockly.com entitled "Simple Maze Search". This maze is a "standard" maze—it has no loops. It is "binary" because at every intersection there are exactly *two* directions that Ozobot can move—either right or left.

Figure 1 shows an example of this maze with a red path. Ozobot starts at the root, making random decisions at each intersection to go either right or left. If the decision leads him away from the red path, he makes adjustments to the sequence of right/left turns that he stores in a stack as he moves. Upon reaching the red leaf (line-end) near the upper right of the maze, he stops. The user then has several seconds to lift Ozobot off of the maze and move him back to the start. Ozobot then follows the red path perfectly while consulting the turns as stored in his stack. The student is now encouraged to view the video *Learn the Red Path,* found here https://youtu.be/jv9WHfL1QSU. Doing so will clarify what the OzoBlockly lesson accomplishes.

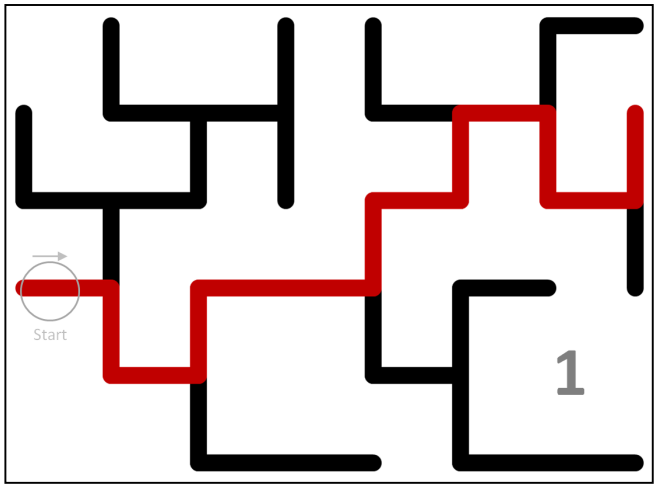


Figure 1

This lesson assumes that the user is familiar with the concept of a stack and the stack operations *push* and *pop*. If the user is not familiar with stacks, then he or she is encouraged to study the Ozobot Bit lesson entitled "Path Reversal with a Stack" at ozobot.com/stem-education prior to doing this lesson. "Path Reversal with a Stack" assumes no prior knowledge of stacks, so it can be used as a practical an independent learning experience for students.

The Main OzoBlockly Program

Now that we have a good understanding of the purpose of this lesson and have viewed Ozobot running the OzoBlockly program, let's start to study the program blocks. The program was written in a modularized manner with several subroutines (OzoBlockly functions) so that it becomes much easier to follow. Figure 2 shows the main program.

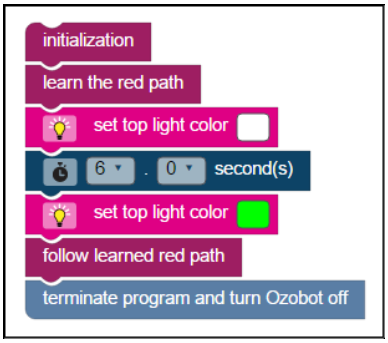


Figure 2

Subroutines are shown in the purple blocks. An initialization subroutine takes care of some "housekeeping" things, to be discussed in the next section of this paper. The "learn the red path" subroutine does just that—Ozobot follows the binary tree making random right/left decisions at each intersection. After he has learned the path, Ozobot sets the top LED to white and gives the user six seconds to move Ozobot back to the start location. Ozobot then follows the learned red path perfectly. After reaching the end of the red path, the program is terminated and Ozobot turns off.

The "Initialization" Subroutine

Figure 3 shows the blocks making up the 'initialization" subroutine. This subroutine does a few things just once before Ozobot begins moving.

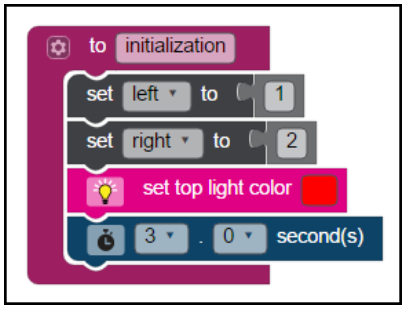


Figure 3

We need some way to distinguish between left and right when storing directions in our stack. The variable *left* will contain a 1, signifying a left turn. The variable *right* will contain a 2, signifying a right turn. Using carefully named variables rather than the numbers 1 and 2 throughout the program helps to self-document what is happening. The user is then given three seconds, after starting the stored program with a double-press of the start button, to place Ozobot on the start location on the binary tree maze.

The "Learn the Red Path" Subroutine

This subroutine is the nuts and bolts of this program, as it involves the most complex logic and makes heavy use of the stack operations of push and pop. Figure 4 shows the OzoBlockly code for this subroutine.

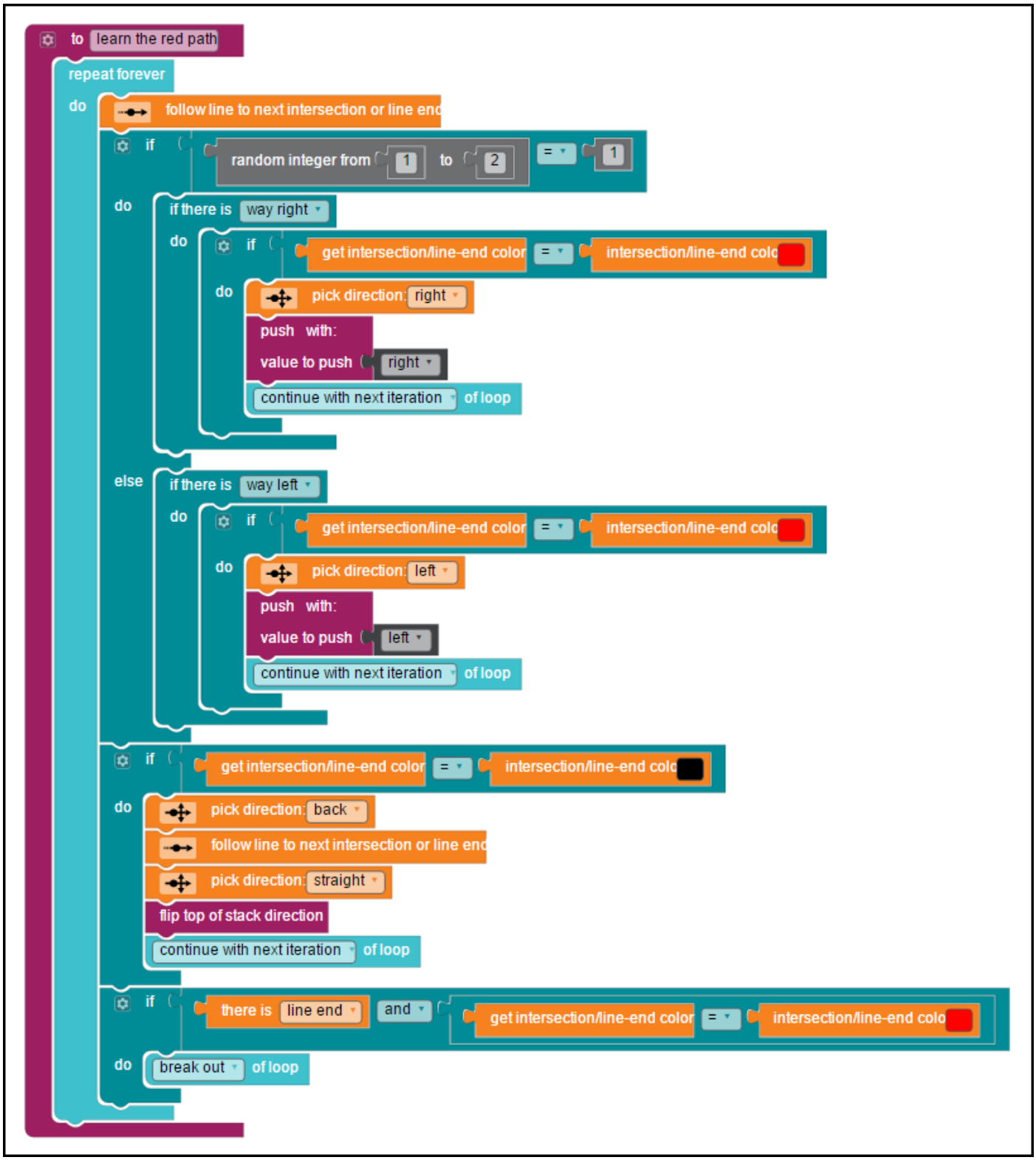


Figure 4

The outermost level of this subroutine is a *repeat forever* loop, which is actually terminated by a *break out of loop* block when Ozobot has reached the end of the red path. Within the loop are three consecutive *if* blocks. These *if* blocks deal with what happens when Ozobot reaches three kinds of points in the binary maze. Figure 5 identifies these three kinds of points by labeling them with the letters A, B, and C.

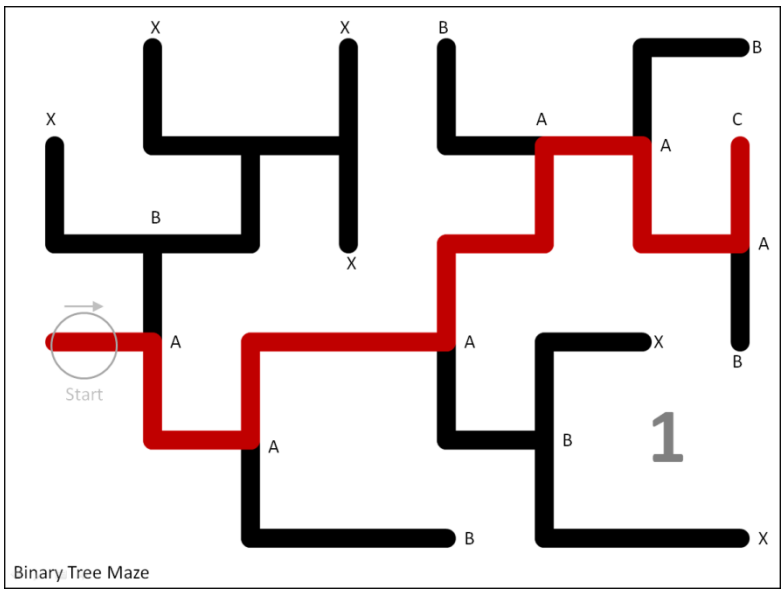


Figure 5

The first *if* block deals with what happens when Ozobot reaches the points labeled with the letter A. At these points, the intersection is red, and Ozobot must decide whether to make a right or left turn. He does this with uniform probability based upon whether a random integer is 1 or 2, respectively. If the random integer is 1, there is a way right, and the intersection is red, he picks the direction right, pushes *right* to the top of the stack, and then continues with the next iteration of the loop. Similar code is shown for a left turn in the event that the random integer is 2. It should be noted that in the event that Ozobot randomly lucks out and turns onto the red branch at every such intersection, then he will follow the path perfectly during his learning phase.

The second *if* block deals with what happens when Ozobot reaches the points labeled with the letter B. At these points, the intersection or line end color is black. Therefore, he has erred and must make a U-turn by picking the direction back. Then he must follow the line to the intersection where he erred. He then goes straight through the intersection and flips the top of the stack from right to left or left to right, to correct his err.

The third *if* statement deals with what happens when Ozobot reaches the point labeled with the letter C. This is a line-end and the end of the red path. In this case Ozobot has completed the learning phase and *breaks out of the loop*.

With the red path shown in Figure 5, Ozobot will never reach any of the points labeled with the letter X. This is because he will be making a U-turn at the points labeled with the letter B. Therefore, even in the learning phase, Ozobot always keeps relatively close to the red path.

The "Push" and "Pop" Subroutines

OzoBlockly provides a single list. Two of its list-processing blocks, shown in blue in Figure 6, can be used to construct push and pop subroutines for a stack. Figure 6 also shows the push and pop subroutines along with typical calling blocks.

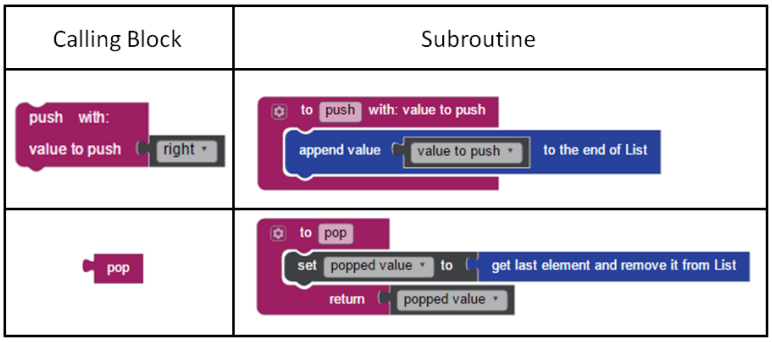


Figure 6

The "append value to the end of the List" block pushes a new value to the end of the list (or to the top of our stack), increasing the length of the list by 1. The example calling block shows that the value of the variable *right* is being pushed to the top of our stack. As indicated in the OzoBlockly *Master Mode Reference*, the List can hold at most 127 elements, and if the capacity is full, the append operation will fail and the user program will terminate with a blue-red LED animation.

The "get last element and remove it from List" block retrieves and removes the last element in the list (or the top item in our stack), decreasing the length of the list by 1. As indicated in the OzoBlockly *Master Mode Reference*, this block should not be used on an empty list, as doing so will cause a runtime error and terminate the user program. The example calling block is simply a "pop" block for a user-defined subprogram that returns the value popped from the top of the stack.

The "Flip Top of Stack Direction" Subroutine

Figure 7 shows the blocks for the "flip top of stack direction" subroutine. If the top of the stack is *left*, then it is changed to *right*, and vice-versa. The value at the top of the stack is popped and saved in the variable *temp*. If *temp* equals *right*, then the value of the variable *left* is pushed to the top of the stack. Otherwise, the value of the variable *right* is pushed to the top of the stack.

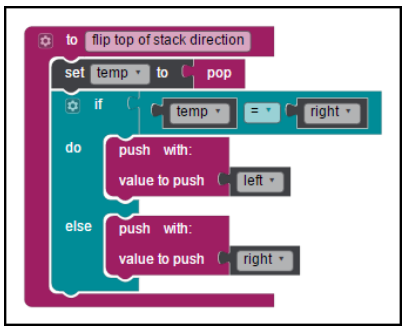


Figure 7

The "Follow Learned Red Path" Subroutine

Figure 8 shows the blocks making up the "follow learned red path" subroutine, the final subroutine in our program. We need to grab the intersection turn directions from the stack starting with the first element in the list. Since list elements are indexed starting at [0], our loop counter variable *i* goes from 0 to one less than the length of the list. Each time through the loop, Ozobot follows the line to the next intersection or line end and sets the variable *direction* to the value of the *i*'th element in the list. If *direction* equals *right*, then Ozobot picks to turn right, else he picks to turn left.

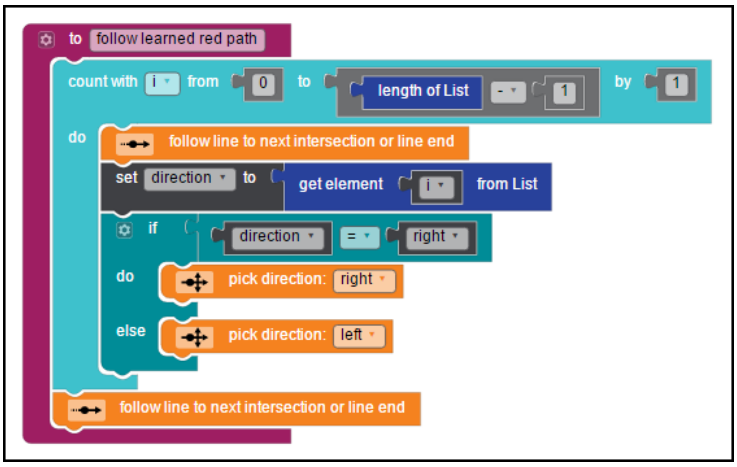


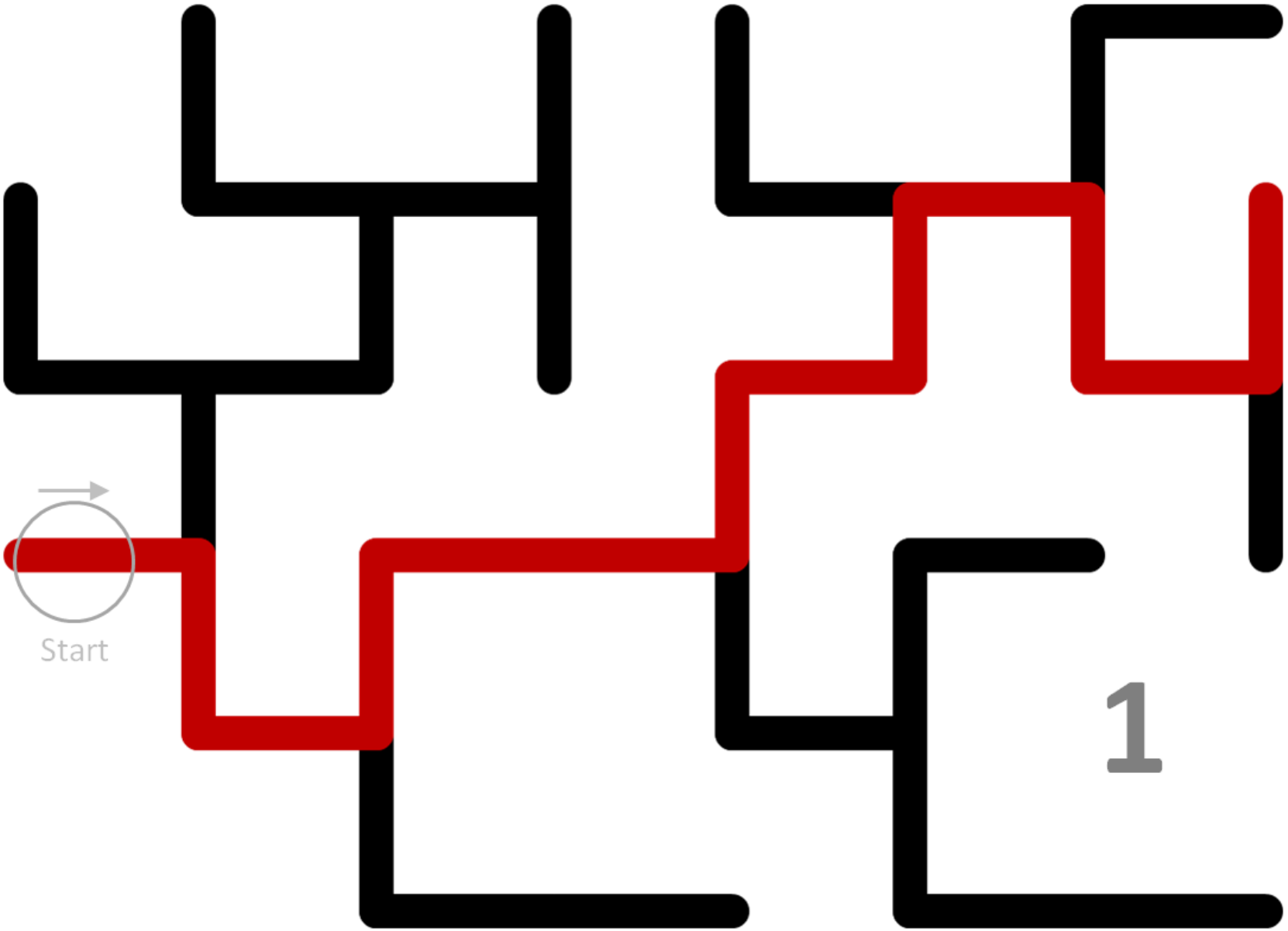
Figure 8

Classroom Activity

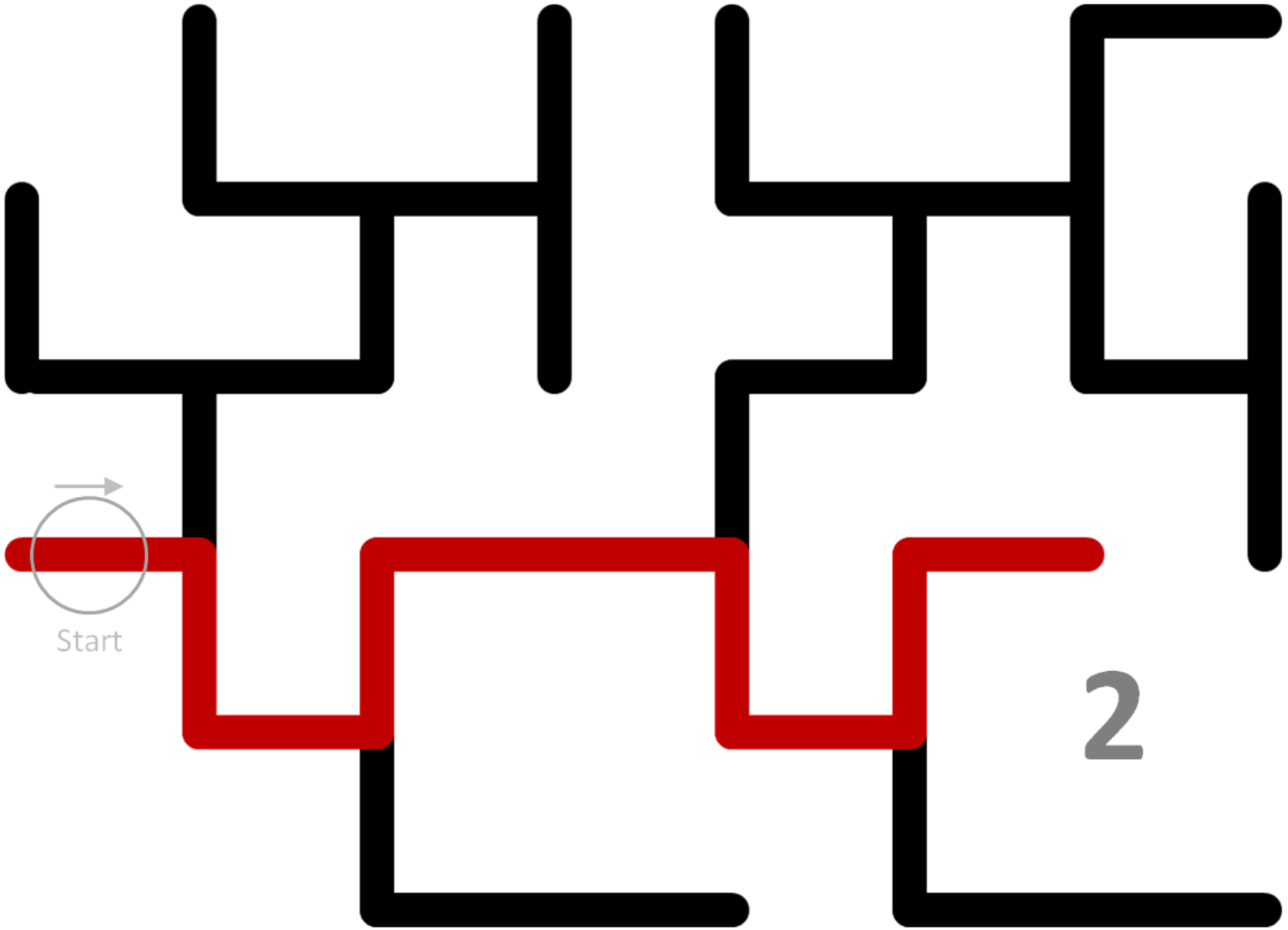
After discussing the OzoBlockly program with the class, provide each student or student group with a copy of the *LearnRedPath.ozocode* file and let them play with the program for a while. Then give them the following program maintenance task:

> *Rather than moving Ozobot back to the start and having him follow the learned red path, it has been requested by users that he simply trace the learned red path* ***backwards from finish to start.*** *Modify the OzoBlockly program to do this and make sure that it works correctly on the five binary tree maze maps provided.*

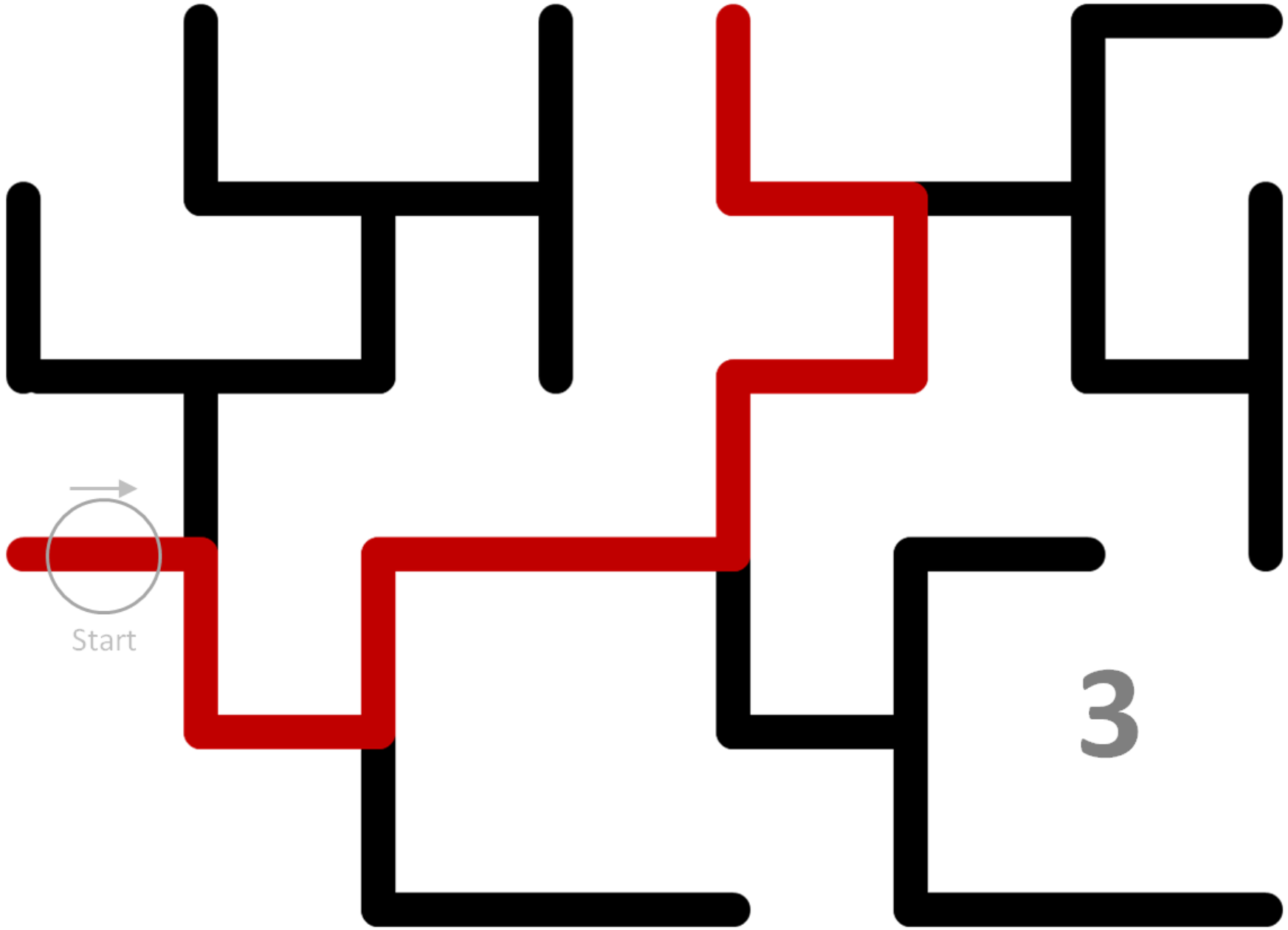
(Note that the last five pages of this document contain five binary tree mazes, each with a different red path. These can be shared by the student groups.)



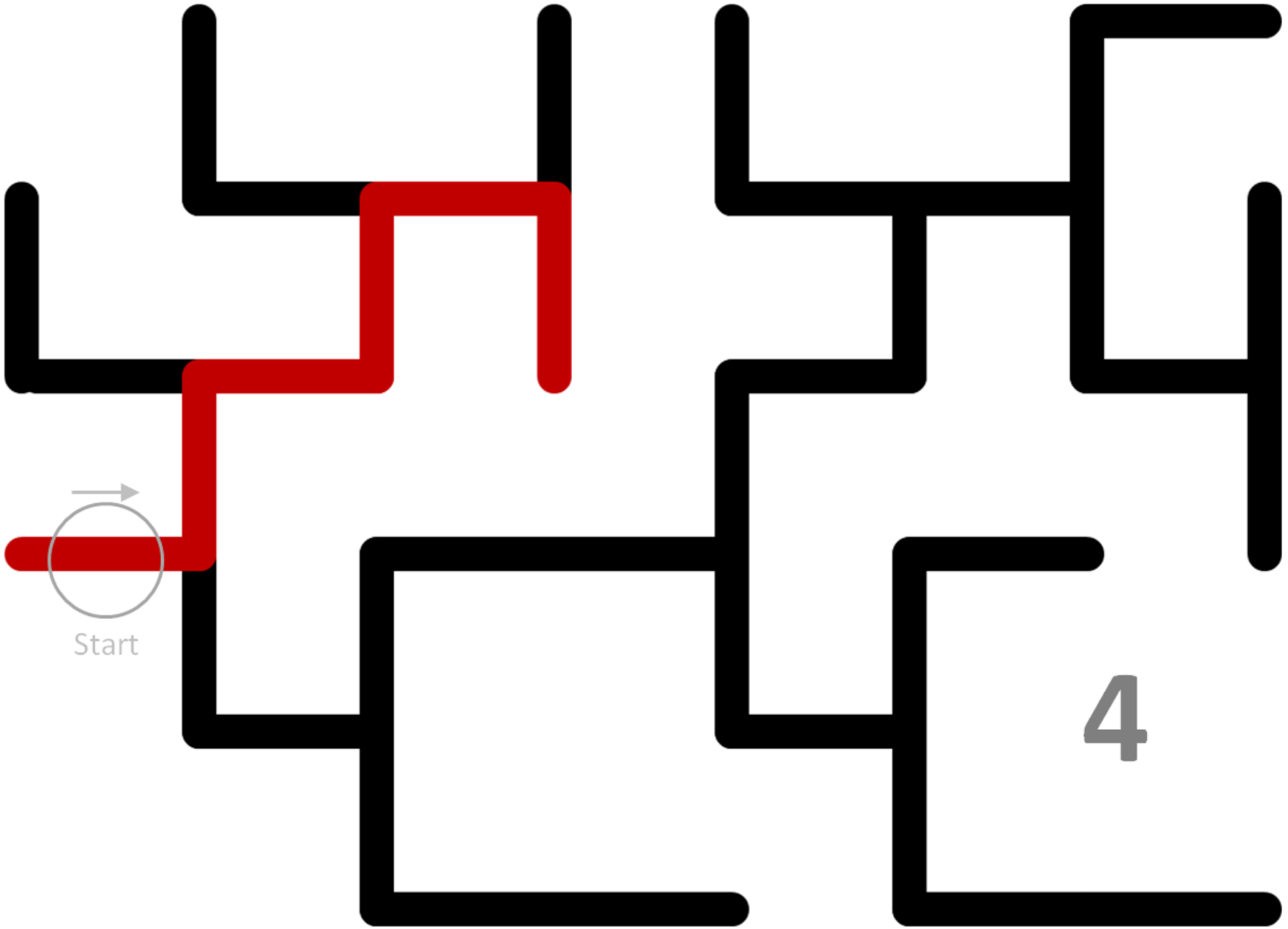
Binary Tree Maze



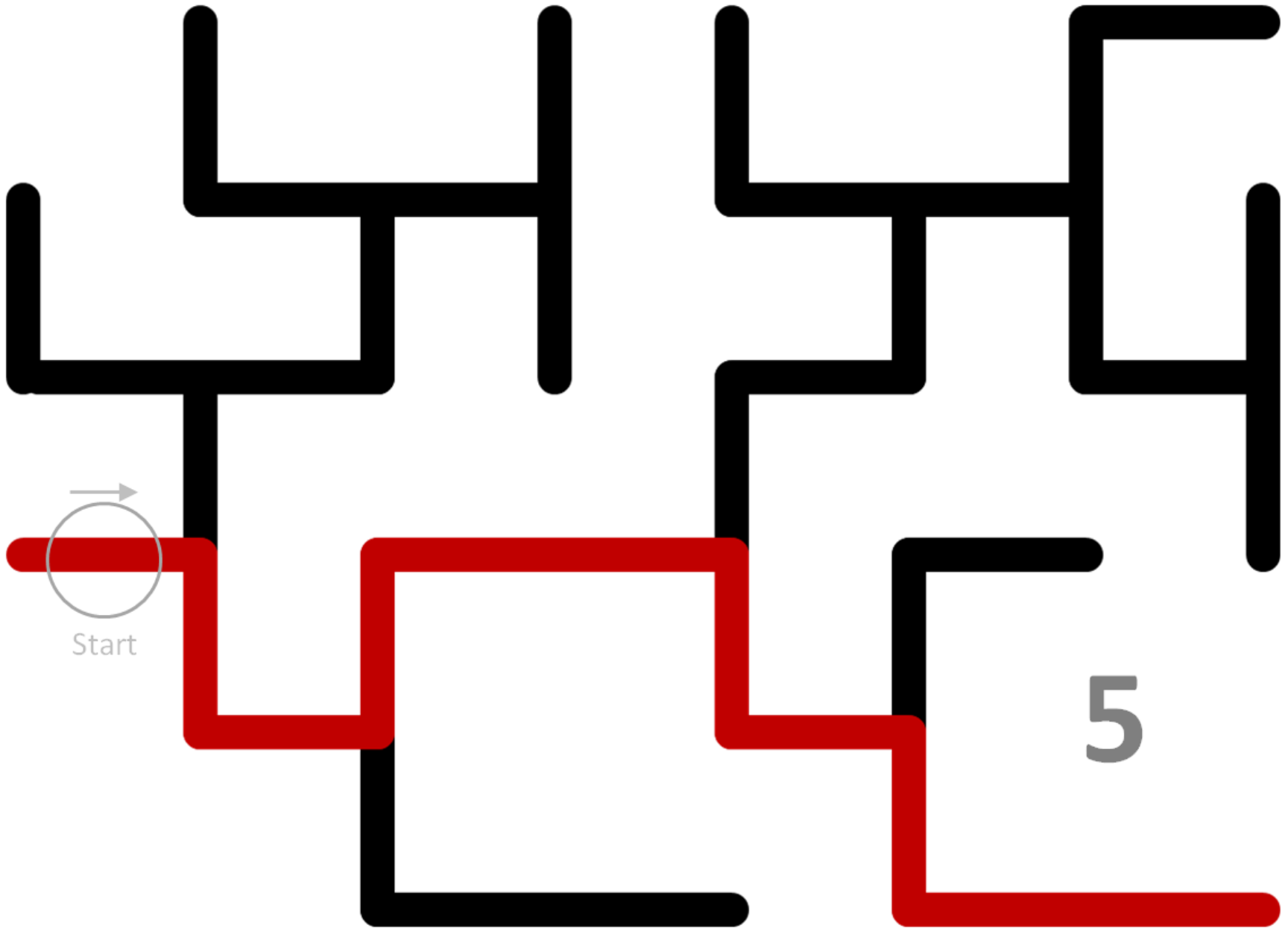
Binary Tree Maze



Binary Tree Maze



Binary Tree Maze



Binary Tree Maze