Extracted from:

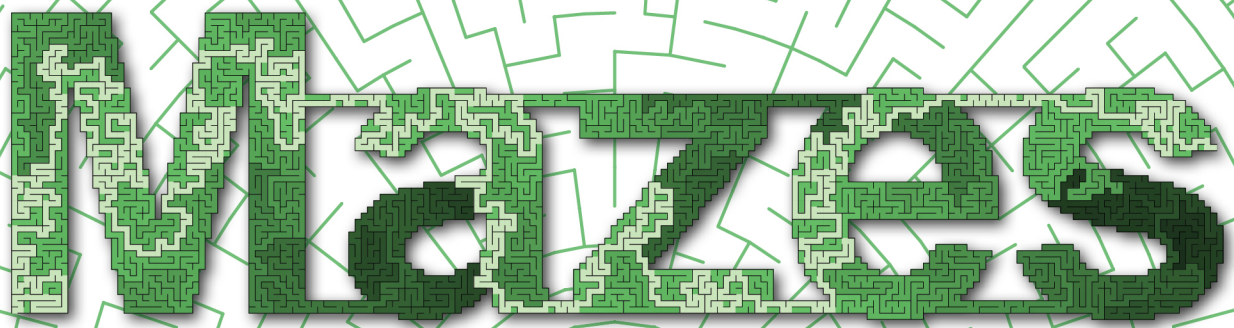
Mazes for Programmers

Code Your Own Twisty Little Passages

The Pragmatic Bookshelf

Dallas, Texas • Raleigh, North Carolina

The Pragmatic Programmers



for Programmers

Code Your Own Twisty Little Passages

Jamis Buck

Edited by Jacquelyn Carter

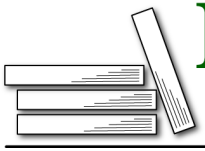
Mazes for Programmers

Code Your Own Twisty Little Passages

Jamis Buck

The Pragmatic Bookshelf

Dallas, Texas • Raleigh, North Carolina



Pragmatic Bookshelf

Our Pragmatic courses, workshops, and other products can help you and your team create better software and have more fun. For more information, as well as the latest Pragmatic titles, please visit us at *https://pragprog.com*.

The team that produced this book includes:

Jacquelyn Carter (editor)
Potomac Indexing, LLC (indexer)
Liz Welch (copyeditor)
Dave Thomas (typesetter)
Janet Furlow (producer)
Ellie Callahan (support)

For international rights, please contact *rights@pragprog.com*.

Printed in the United States of America.
ISBN-13: 978-1-68050-055-4
Encoded using the finest acid-free high-entropy binary digits.
Book version: P1.0—July 2015

So far, we've looked at two different maze algorithms, and while both were straightforward to understand and implement, they also had some pretty significant biases. They could be worked around, sure, but maybe there's a better way. In this chapter we're going to try to balance the scales a bit by exploring two new algorithms, Aldous-Broder and Wilson's, both of which are guaranteed to be *absolutely unbiased*.

If that sounds too good to be true, that's because it almost is! Nothing comes without a price, so we'll also see where these two algorithms—despite their mathematical flawlessness—ultimately end up with their own set of disappointments.

To get there, though, we need to have a better understanding of what biases really are, and what it means for an algorithm to have them. Let's take a short step back and look at biases from another angle, so we can see more precisely what their absence really means.

Understanding Biases

It's easy to say "this algorithm has a bias" and "that algorithm has no bias," but so far we haven't been very specific about what we mean by that. It's more than simply identifying a long passage or a slantwise tendency in a maze. A visible texture or pattern in a maze—in the output of an algorithm—is only evidence of an algorithm's bias if a significant number of mazes generated by that algorithm exhibit that same texture.

But even that definition doesn't quite cover it. Unlike Sidewinder and Binary Tree, biases may not always be blatantly obvious. They may not even produce visible artifacts at all. To understand how that can be, let's consider an example.

Joe asks:

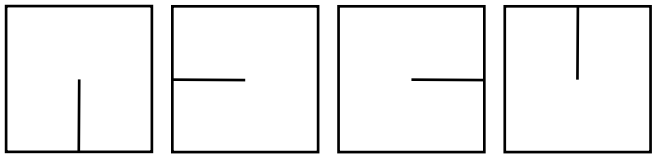
Can't Mazes Have Biases, Too?

Walter Pullen, on his "Maze Classification" page,[a] uses the term "bias" to describe a specific class of texture involving passage direction. A maze with horizontal bias, for instance, will have longer east-to-west passages. So, yes, in some contexts, the word bias may be applied to mazes, as well as algorithms.

However, to avoid confusion, in this book "bias" will refer strictly to algorithms, while the more general term *texture* will be used to describe mazes. That is to say, a *bias of the algorithm* may produce a *texture in the maze.*

a. http://www.astrolog.org/labyrnth/algrithm.htm

Let's say we wanted to generate a perfect 2×2 maze. (Recall that "perfect" here means one with no loops in it.) The following figure shows all four of the possible perfect mazes that can fit in a 2×2 grid. By generating a 2×2 maze randomly, we're effectively choosing between these four possibilities.



This means that, conceptually, randomly generating a maze is the same as putting all of the possible mazes into a big bag, giving them a good shake (you know—to make sure they're well shuffled), and then reaching into that bag and blindly picking one of them.

The Binary Tree algorithm is one way to do this. It lets us effectively reach into that big bag full of all possible mazes and pick one at random, but it does so *with bias*. In terms of our big bag of mazes, this means that the algorithm doesn't actually choose evenly from among all the possibilities. *It cheats.*

Consider the four perfect mazes in the preceding figure again. Ideally, we'd like the Binary Tree algorithm to be able to generate all of them, but it can't. (Or won't!) Recall what we know about its biases, revealed in the unbroken passages it always leaves on the north and east. We see right away that C is impossible because the eastern passage is broken in half by a wall. Binary Tree would never give us that one. D is similarly impossible—that passage across the north is also split by a wall. In other words, Binary Tree flatly refuses to give us anything but A and B, 50% of the possible 2×2 mazes. *It is biased against those other possibilities.*

What about Sidewinder? When we recall that Sidewinder always generates mazes with an unbroken northern passage, we immediately see that D is out, because of that wall on the north. That gives us A, B, and C, or 75% of the possible 2×2 mazes, which is better. But still—what we *want* is an algorithm that will choose from *all* of them!

And as long as we're making demands, we might as well add that we'd like to pick them *uniformly*. That is to say, we want to make sure that every maze

in the bag has an equal chance of being selected. Some algorithms are *nonuniform*, which is to say that they might eventually give us every possible maze but tend to produce certain types more often (perhaps preferring longer passages, for instance). These may be subtle biases, difficult or impossible to spot by eye, but we don't want those kinds, either.

Put simply, we'd like to see what happens when we choose our mazes *uniformly* and *at random* from the set of all possible mazes. The good news is that there *are* ways to do it, and they're pretty straightforward, but, like everything, there are trade-offs.

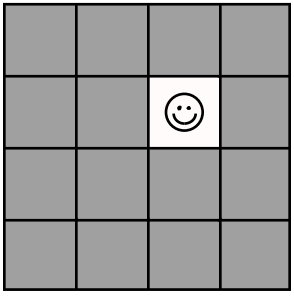
One way is called the Aldous-Broder algorithm. Let's start there.

The Aldous-Broder Algorithm

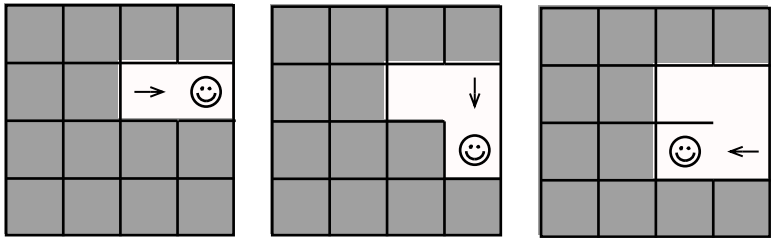
The Aldous-Broder algorithm was developed independently by both David Aldous, a professor at UC Berkeley, and Andrei Broder, currently a Distinguished Scientist at Google. It is almost as simple to implement as the Binary Tree algorithm. The idea is just this: Start anywhere in the grid you want, and choose a random neighbor. Move to that neighbor, and if it hasn't previously been visited, link it to the prior cell. Repeat until every cell has been visited.

Easy, right? It's that *random walk*, the aimless, directionless meandering from cell to cell, that is at the root of this algorithm's ability to avoid being biased. Sadly, as we'll see it also means that it can take a long time to run.

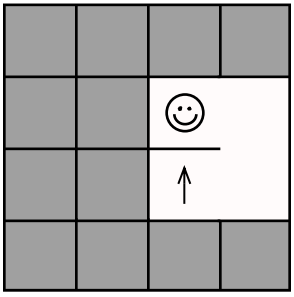
Let's walk through it once so we can see it in action. We'll start by picking a cell at random. We'll color unvisited cells gray, and our old friend, the smiley face, will indicate which cell is current.



We need to pick a random neighbor, so let's choose east. That neighbor hasn't been visited yet, so we link the two cells together, and then we do the process again from that new cell. The following figure shows three steps in a row, each taking us to a new, unvisited cell.

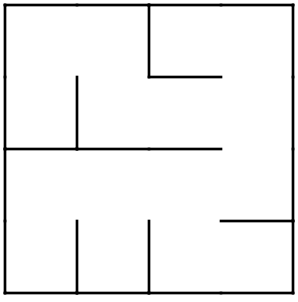


Check out this next move, though. We choose a random neighbor, going north this time, but that cell has already been visited. That's okay! It's how the algorithm works. The only difference is, this time, we don't link the two cells. We simply make the neighbor the current cell, and carry on.



The process continues until every cell has been visited, which, for large mazes, can take a while. Go ahead and finish out this maze yourself. Assuming you're picking the neighbors randomly, you'll see that the random walk will tend to meander, visiting and revisiting some cells multiple times. By the time you're down to just one or two unvisited cells remaining, it can be downright infuriating to watch the algorithm cluelessly walk right past them!

So, have a go at it. When you're done, you should have something like this.



Not bad! It certainly takes a bit of time to reach those last few unvisited cells, but the process itself almost couldn't be simpler.

Next let's see how to make it real in code.

Implementing Aldous-Broder

As might be expected, that random walk forms the core of our implementation, repeatedly visiting neighboring cells until no unvisited cells remain. It comes together without any surprises, just as described.

Put the following code in a file named aldous_broder.rb. As before, we'll put the algorithm in its own class so we can reuse it more easily.

aldous_broder.rb

```
Line 1  class AldousBroder
        
        
          def self.on(grid)
            cell = grid.random_cell
        5   unvisited = grid.size - 1
        
            while unvisited > 0
              neighbor = cell.neighbors.sample
        
       10     if neighbor.links.empty?
                cell.link(neighbor)
                unvisited -= 1
              end
        
       15     cell = neighbor
            end
        
            grid
          end
       20 
        end
```

Line 4 starts everything off by choosing one of the cells at random. The random walk will begin at that cell. To make sure the algorithm knows when everything has been visited, line 5 computes the number of unvisited cells in the grid. (We subtract one, because we treat the starting cell as having been visited already.) Each time a new cell is visited, that value will be decremented (line 12), and the loop continues until that value is zero (line 7).

On each pass through the loop, we choose a neighbor of the current cell at random (line 8) and make it the new current cell (line 15). If that cell hasn't yet been linked to any other cells (which implies that it hasn't been visited before), we link it to the current cell (line 11) before going around again.

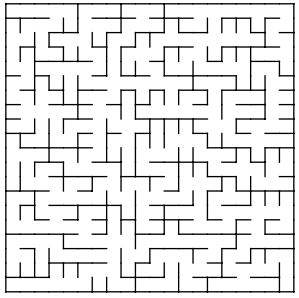
A simple demo program will suffice for testing this. Put the following in aldous_broder_demo.rb.

aldous_broder_demo.rb
```
require 'grid'
require 'aldous_broder'

grid = Grid.new(20, 20)
AldousBroder.on(grid)

filename = "aldous_broder.png"
grid.to_png.save(filename)
puts "saved to #{filename}"
```

Feel free to display the maze using the terminal, if you prefer; this code writes the maze to a PNG simply so that we can make it larger, to better see the structure of the finished maze. Running it will save the maze to a file, like this:

```
$ ruby -I. aldous_broder_demo.rb
saved to aldous_broder.png
$
```

Opening up aldous_broder.png, we ought to see something like this figure.



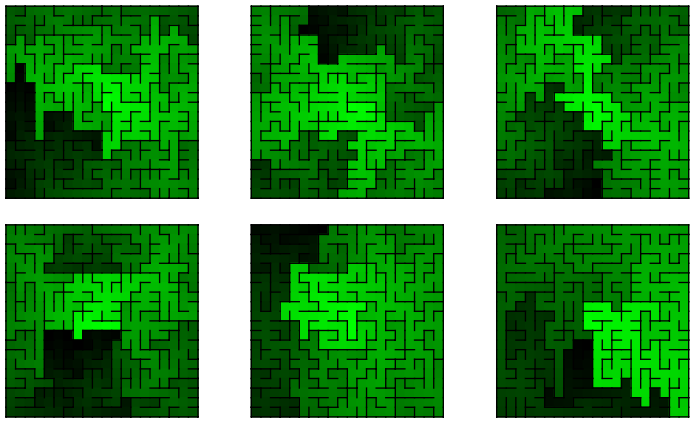
Very nice. Let's go one step further and see if this algorithm produces any kind of texture that its mazes all have in common. Using our coloring.rb program from the previous chapter as a template, let's modify aldous_broder.rb to emit a colored version of the maze. In fact, let's have it generate and color several mazes at once, so we can open them all up side-by-side to compare. Put the following code in aldous_broder_colored.rb.

aldous_broder_colored.rb
```
require 'colored_grid'
require 'aldous_broder'

6.times do |n|
  grid = ColoredGrid.new(20, 20)
  AldousBroder.on(grid)

  middle = grid[grid.rows / 2, grid.columns / 2]
  grid.distances = middle.distances

  filename = "aldous_broder_%02d.png" % n
  grid.to_png.save(filename)
  puts "saved to #{filename}"
end
```

When we run it, it will generate six different images.

```
$ ruby -I. aldous_broder_colored.rb
saved to aldous_broder_00.png
saved to aldous_broder_01.png
saved to aldous_broder_02.png
saved to aldous_broder_03.png
saved to aldous_broder_04.png
saved to aldous_broder_05.png
```

If we look at those images now, we ought to see something like these.



Remember that in general, any texture that we see is only evidence of a bias if every maze generated by the algorithm has that texture in common. Looking at this spread, you may see a few that look like they want to stretch vertically, and a couple seem to have a horizontal bridge in the middle, but there aren't any characteristics that they all seem to share. This definitely supports the claim that Aldous-Broder has no bias!

As compelling as it is, though, it doesn't mask the largest drawback of a purely random walk. All is not rosy in Aldous-Broder Land. Each individual step of the algorithm may execute quickly, but the algorithm itself can run for a long time, especially on large mazes. That aimless meandering over previously visited cells feels awfully wasteful, but we can't add a *heuristic*, or a way to give the process some intelligence, without taking away the very property of uniformity that we said we wanted.

So, we can't change Aldous-Broder itself, but maybe we can try a *different* algorithm instead. Let's see if Wilson's does any better.